2 *FISH* REFERENCE

2.1 Introduction and Overview

This section contains a detailed reference to the *FISH* language. Following the introduction, Section 2.2 describes the rules of the language and how variables and functions are used. Section 2.3 explains *FISH* statements, and Section 2.4 describes how the *FISH* language links with *UDEC*. Predefined *FISH* variables, functions and arrays are described in Section 2.5. Section 2.6 discusses extensions to *FISH* for file manipulation, the use of which generally requires a reasonable understanding of programming techniques and constructs (although *FISH* can be used without reference to these extensions).

FISH is a programming language embedded within *UDEC* that enables the user to define new variables and functions. These functions may be used to extend *UDEC*'s usefulness or add user-defined features. For example, new variables may be plotted or printed, special block generators may be implemented, servo control may be applied to a numerical test, unusual distributions of properties may be specified and parameter studies may be automated.

FISH is a "compiler" (rather than an "interpreter"). Programs entered via a *UDEC* data file are translated into a list of instructions (in "pseudo-code") stored in *UDEC*'s memory space; the original source program is not retained by *UDEC*. Whenever a *FISH* function is invoked, its compiled pseudo-code is executed. The use of compiled code – rather than interpreted source code – enables programs to run much faster. However, unlike a compiler, variable names and values are available for printing at any time; values may be modified by the user by using *UDEC*'s **SET** command.

FISH programs are simply embedded in a normal *UDEC* data file: lines following the **DEFINE** command are processed as a *FISH* function; the function terminates when the **END** command is encountered. Functions may invoke other functions, which may invoke others, and so on. The order in which functions are defined does not matter as long as they are all defined before they are used (e.g., invoked by a *UDEC* command). Since the compiled form of a *FISH* function is stored in *UDEC*'s memory space, the **SAVE** command saves the function and the current values of associated variables.

2.2 *FISH* Language Rules, Variables and Functions

2.2.1 Lines

FISH programs can either be embedded in a normal *UDEC* data file or they may be entered directly from the keyboard. Lines following the **DEFINE** command are taken to be statements of a *FISH* function; the function terminates when the **END** command is encountered. A valid line of *FISH* code must take one of the following forms.

1. The line starts with a statement, such as **IF**, **LOOP**, etc. (see Section 2.3).

2. The line contains one or more names of user-defined *FISH* functions, separated by spaces – e.g.,

   ```
   fun_1    fun_2    fun_3
   ```

   where the names correspond to functions written by the user; these functions are executed in order. The functions need not be defined prior to their reference on a line of *FISH* code (i.e., forward references are allowed).

3. The line consists of an assignment statement (i.e., the expression on the right of the **=** sign is evaluated and the value given to the variable or function name on the left of the **=** sign).

4. The line consists of a *UDEC* command, provided that the line is embedded in a section of *FISH* code delimited by the **COMMAND** – **ENDCOMMAND** statements (see Section 2.3.3).

5. The line is blank or starts with a semicolon.

FISH variables, function names and statements must be spelled out in full; they cannot be truncated as in *UDEC* commands. Lines must contain fewer than 80 characters. No continuation lines are allowed; intermediate variables may be used to split complex expressions. *FISH* is "case-insensitive" by default (i.e., it makes no distinction between uppercase and lowercase letters); all names are converted to lowercase letters. Spaces are significant (unlike in FORTRAN) and serve to separate variables, keywords, and so on; no embedded blanks are allowed in variable or function names. Extra spaces may be used to improve readability (for example, by indenting loops and conditional clauses). Any characters following a semicolon (;) are ignored; comments may be embedded in a *FISH* program by preceding them with a semicolon. Blank lines may be embedded in a *FISH* program.

2.2.2 Reserved Names for Functions and Variables

Variable or function names must start with a non-number, and must not contain any of the following symbols.

. , * / + - ^ = < > # ( ) [ ] @ ; ’ "

User-defined names can be any length but they are truncated in printout and in plot captions, due to line-length limitations. In general, names may be chosen arbitrarily, although they cannot be the same as a *FISH* statement (see Section 2.3) or a predefined variable or function (see Section 2.5). There are also many other words used in *UDEC* input that should be avoided. The list contained in Table 2.1 shows all words that could give rise to a conflict if used to name a *FISH* variable or function. However, the potential conflict depends on the way the chosen name is used. For example, the word **gravity** could be used as a *FISH* variable, provided that it is simply referred to inside a *FISH* function; a conflict would arise only if it is necessary to use the **SET** command to set its value, since **gravity** is a valid argument to the **SET** command. Similarly, it may be impossible to print the value of a *FISH* variable if its name is the same as a parameter for the **PRINT** command. If in doubt, avoid any of the names listed in Table 2.1, or contractions of the names (since *UDEC* allows truncation of keywords and commands). A simple way to avoid name conflicts would be to begin all *FISH* variable and function names with an underscore character.

Table 2.1 List of words in UDEC and FISH that may conflict with chosen names

Name	Name	Name	Name	Name	Name
-nwppressure	atan	barton	border	c_sdis	cell
-ppressure	atan2	base	both	c_sforce	cf_axi
a_us	atblock	bb	bou_gp	c_type	cf_cell
a_wipp	atcontact	bbdamage	bou_head	c_x	cf_creep
a3	atdomain	bbdil	bou_near	c_y	cf_fluid
a4	atgridpoint	bbdmax	bou_xreaction	ca85	cf_pstress
abc	atzone	bbdmin	bou_yreaction	cable	cf_tflow
above	automatic	bbdplateau	bound	cable_elem_head	cf_thermal
abs	autoname	bbdtable	boundary	cable_node_head	cforce
absolute	avelocity	bbjcsn	box	call	cfriction
acap	average	bbjrcn	bracket	cap_pressure	change
acos	axial	bbkni	brown	cap85	char
act_energy	azero	bbnc	brvelocity	capmin	checkesc
add	b_area	bbsc	bvelocity	capratio	chkesc
add_dil	b_bex	bbunc	btol	case	circular
added_mass	b_cons	bbunl	bulk	case_of	clear
address	b_corner	bbunm	bulk_mod	caseof	clemin
add_stress	b_dsf	bbunpc	bulkw	cave	clipboard
adev	b_extra	bbusc	bw	cavi	clock
afailure	b_fix	bbusm	bxload	cb_area	close
age	b_gp	bbusmb	bxvelocity	cb_density	closure
alias	b_group	bbvirr	byload	cb_fstrain	code_name
alp11	b_mass	bbvmi	byvelocity	cb_kbond	cohesion
alp12	b_mat	bcap	c_b1	cb_sbond	cohw
alp22	b_moi	be	c_b2	cb_spacing	color
alp33	b_mom	beam	c_cons	cb_thexp	com
and	b_near	beboundary	c_d1	cb_tol	combined
angle	b_next	begin	c_d2	cb_ycomp	command
anis	b_rvel	beinterface	c_extra	cb_yield	compressible
annulus	b_type	below	c_group	cb_ymod	conductivity
aperture	b_wipp	beta	c_jex	cb85	config
apmode	b_x	bfill	c_length	cc85	connect
apply	b_xforce	bfm	c_link1	ccdifference	constant
aptable	b_xload	bfx	c_link2	ccfail	constitutive
arc	b_xvel	bfy	c_mat	ccmean	constitutive_model
area	b_y	biot_c	c_ndis	ccohesion	contact_head
ares	b_yforce	black	c_near	ccs1	contacts
array	b_yload	block_head	c_next	ccs2	continue
arrow	b_yvel	blocks	c_nforce	cd	convection
asin	b_zone	blue	c_nx	cd85	copy
associated	back	bminw	c_ny	cdilation	cor_block
at	background	bmp	c_obj_type	cdisplacement	cor_bou

***Table 2.1** List of words in UDEC and FISH that may conflict with chosen names (cont.)*

Name	Name	Name	Name	Name	Name
cor_extra	ctable	directory	end	ff_density	fractures
cor_gp	ctensile	displacement	end_case	ff_nu	fracz
cor_link	ctension	dist	end_command	ff_shearmod	free
cor_near	cust1	div	end_factor	ff_ymod	freeze_con
cor_obj_type	cust2	dlist	end_if	ffield	friction
cor_rlink	cutting	dll	end_loop	ffsxx	fstrain
cor_x	cw85	dmagnitude	end_section	ffsxy	ftable
cor_xdis	cy	dmaterial	endcase	ffsyy	ftemp
cor_xvel	cyan	dnumber	endcommand	ffxacceleration	ftime
cor_y	cycle	domain_head	endif	ffxdisplacement	fullpalette
cor_ydis	cycles	domains	endloop	ffxvelocity	function
cor_yvel	cydamage	druck	endsection	ffyacceleration	fupd
corners	cysub	druck0	energy	ffydisplacement	fvelocity
cos	d_contact	dscan	ep11	ffyvelocity	g
cosine	d_extra	dshear	ep12	field	g12
coulomb	d_fix	dtable	ep22	filcolor	gap
cperm	d_near	dtflow	ep33	file	gas
cppudm	d_next	dump	eps	filename	gas_alpha
cptable	d_obj_type	dxf	erase	fill	gas_bulkmin
cr85	d_pp	dy	error	filtable	gas_c
crack	d_temp	dy_state	et_plastic	find	gas_constant
crack_flow	d_vol	e	ev_plastic	first_node	gas_densitymin
crack_store	d_wipp	e_dot_star	evol	fish	gas_flow
crdt	d_x	e_plastic	exclude	fix	gcap
creep	d_y	e_primary	exit	flies	gen
creeptime	dami	e_tension	exp	float	generate
cross	damping	e1	expa	flow	geom
crtdel	debug	e2	extend	flowrate	geps
crtime	def	echo	exx	flowtime	get_mem
cs_cftable	define	edge	exy	fluid	gfailure
cs_ncohesion	deformable	edgemax	eyy	fluid_bulk	giic
cs_nfriction	degrad	edxx	f_prop	fluid_density	giicpath
cs_nstiffness	degrees	edxy	failure	fluid_dt_type	ginterface
cs_scohesion	delc	edyy	fang	flux	gp_addxmass
cs_sctable	delete	eff	fast_read	fmem	gp_bou
cs_sfriction	delt	elastic	fbcontour	fobl	gp_corner
cs_sftable	density	element	fbflow	fobu	gp_dsf
cs_spacing	deterministic	elsemin	fboundary	force	gp_extra
cs_stiffness	development	else	fc_arg	fos	gp_mass
cscan	diamond	emf	fcut	fp	gp_near
cscint	dilation	emod	fdilation	fracb	gp_next
cstate	dim	empd	ff_bulkmod	fraction	gp_x

Table 2.1 List of words in UDEC and FISH that may conflict with chosen names (cont.)

Name	Name	Name	Name	Name	Name
gp_xdis	hmaximum	inverse	join_jks	list	max_length
gp_xforce	hoek	iset	join_ratio	lmagenta	maxdt
gp_xvel	hold	ishear	joined	lmul	maxgiicmem
gp_y	hp7550a	istrain	joint	ln	maxima
gp_ydis	hpgen	it	jointid	lnslope	maximum
gp_yforcel	hpgl	iterations	jperm	lo	maxjkn
gp_yvel	hpp	itmax	jpg	loads	maxjks
gpeq	hread	iwhite	jplot	local	maxmech
gpforces	hydraulic	j_model	jpmode	location	maxpsi
gpneg	i	j_nstress	jproperty	log	mech
gpsource	ibou_head	j_prop	jptable	logfile	mem
gradient	id	j5flow	jrco	loop	memory
grand	idilation	jangle	jregion	lose_mem	message
grav_x	if	jc4	jrepath	lred	mevol
grav_y	if_cohesion	jc5	jrescohesion	lsactiv	min
gravity	if_dilation	jcoh	jrfriction	ltype	min_length
gray	if_friction	jcohesion	jrough	m_bulk	mindt
grayscale	if_kn	jcondf	jrtensile	m_cohesion	minimum
grdexp	if_ks	jconstitutive	jset	m_density	minjkn
green	if_tensile	jcso	jsr	m_dilation	minjks
grhist	ifailure	jdelete	jtensile	m_friction	mink
grid	imem	jdilation	jtension	m_jcohesion	minkjkn
gridpoint	impermeable	jdisplacement	k	m_jdilation	minkjks
groups	implicit	jen	k11	m_jfriction	minterface
grout	impulse	jes	k12	m_jkn	mixed
grtable	in	jfriction	k22	m_jks	model
gui	include	jhistory	key	m_jrescoh	mohr
ggvel	incompressible	jif	kn	m_jrfric	moment
gvelocity	increment	jkn	ks	m_jrtens	moment_thrust
gwflow	inertia	jks	ktable	m_jrtension	mov
half	inexca	jline	label	m_jtension	movie
hard	information	jmat	large	m_shear	mptable
hbm	initemperatur	jmatdf	last_node	m_tension	mscale
hbs	initial	jmaterial	latency	magenta	mtable
hd	inormal	jmodel	lblue	magnify	mtype
head	insert	jmp	lcyan	manual	mubex
heading	inside	jmtable	left	maperture	mubim
help	insitu	jndisp	legend	mass	multi
hide	int	join	lgreen	mat	multiplier
hinterface	interface	join_block	line	match	multiply
history	interior	join_contact	link	material	n
hl	interval	join_jkn	linked	max	n_wipp

***Table 2.1** List of words in UDEC and FISH that may conflict with chosen names (cont.)*

Name	Name	Name	Name	Name	Name
name	nstable	pcxfile	quit	resolution	section
ncharres	nstep	pen	quitonerror	restart	security
ncycle	nstress	percent	r_aexp	restore	seed
ndisplacement	ntcyc	permeable	r_astiff	return	seek
neg	nther	perturb	r_head	rezone	seepage
neighbor	null	pfix	r_length	rfriction	segment
nerr	number	pfree	r_prop	right	separation
netserver	nvelocity	pgradient	r_rfac	rigid	set
new	nwbulk	phir	r_sexp	rlength	set_ref
newconstitutive	nxcperm	pi	r_spacing	rockbolt	sforce
newmaterial	nwdensity	pin	r_sstiff	rot	sgn
nflow	nwflow	plastic	r_str	rotate	sgrid
nfmech	nwfluid	pline	r_type	rotation	shape
nforce	nwimpermeable	plot	r_uaxial	round	shear
ngicap	nwjperm	pmoment	r_shear	rrfac	shear_mod
nhistory	nwpermeable	png	radiation	rset	shistory
njangle	nwppressure	point	radius	resxp	show
nmech	nwpxgradient	postscript	raexp	rshear	sig1
nmultiply	nwpygradient	pp	random	rsstiff	sig2
no_layer	off	ppressure	range	rstr	sigmac
no_restore	offset	pr12	rastiff	rstrain	sigz
noage	omega	pratio	rat	rtable	sin
nocon	on	pre_parse	rate_dependent	rtension	sine
nodal	only	pressure	ratio	rtol	single
node	oorc	principal	rcohesion	ruzxial	size
nodis	open	print	reaction	run	skip
nofix	or	processors	read	rushear	slave
noheading	origin	projected	red	rvel	slip
nojit	out	propcontour	ref	s3t	small
nonwetting	outer_domain	property	ref_loc	salztyp	smat
noop	output	ps	reftime	sampled	smaximum
noopen	outside	psrotate	region	satmax	sminimum
normal	overlap	pstatic	reinforce	satmin	smultiply
noscale	overwrite	psxscal	rel_version	saturation	snode
not	ovyol	psxshift	relax	satxgrad	sntable
nowhite	p_stress	psyscal	reldisplacement	satygrad	sol_fmag
npens	paginate	psyshift	relstress	save	sol_fob
nphi	parse	ptable	remove	scale	sol_ratio
nphir	partial	ptol	rename	sclin	sol_rloc
npoints	pause	pxgradient	replot	sclose	sol_rmax
npsi	pcmax	pygradient	reset	sdifference	solve
nshift	pcx	quad	residual	sdisplacement	solve_lcal

Table 2.1 List of words in UDEC and FISH that may conflict with chosen names (cont.)

Name	Name	Name	Name	Name	Name
solve_ratio	steps	tcut	udm	xacceleration	ytable
sopen	stiff	tdel	ufriction	xc	yvelocity
source	stiffness	temp_wipp	umul	xcond	yviscosity
spacing	stld	temperature	unbal	xcondition	ywindow
specheat	stol	tension	unbalanced	xdis	ywtable
sphi	stop	test	unbvolume	xdisplacement	z_biot
sphi0	str_elem_head	text	unslave	xform	z_block
split	str_int_head	tf	upco	xfree	z_bulk
spsi	str_node_head	tfix	upcon	xgradient	z_density
sqrt	strain	tflow	updo	xgrav	z_extra
square	stress	tfree	uplen	xhistory	z_fsi
sratio	stresses	tfres	upne	xload	z_fsr
sread	string	tfstrain	urand	xmultiply	z_gp
ss	structural	thapp	ucohesion	xntn	z_group
ssi	structure	thdt	urfriction	xrange	z_inside
ssr	sub	then	urtension	xreverse	z_mass
sstress	sub_version	thermal	utension	xtable	z_mat
st_area	sup_alfa	theta	varz	xvelocity	z_model
st_density	sup_alpha	thexpansion	velocity	xviscosity	z_near
st_inertia	sup_constant	thickness	version	xwindow	z_next
st_pmtable	sup_delete	thistory	vflow	y	z_pp
st_prat	sup_fmax	thrust_shear	vmagnitude	yacceleration	z_prop
st_rcrack	sup_head	thtime	vmaximum	yc	z_rot
st_scresid	sup_kn	time	voltol	ycomp	z_shear
st_shape	sup_spacing	title	von_mises	ycompression	z_state
st_shear	sup_tmax	tol	voronoi	ycond	z_sxx
st_spacing	sup_ycomp	top	vrat	ycondition	z_sxy
st_thexp	support	trace	vs	ydisplacement	z_syy
st_thickness	svelocity	transient	vsig	yellow	z_szz
st_width	swrite	transparency	water	yfree	z_x
st_ycomp	sxx	trigon	well	ygradient	z_y
st_yield	sxy	ttable	wetting	ygrav	z_zex
st_ymod	symfail	ttyp	while	yhistory	zdilation
st_yresid	system	tunnel	while_stepping	yield	zero
standard	syy	twophase	whilestepping	yielderr	zgradient
start	szz	type	white	yload	zone
state	table	ub_angle	width	ymodulus	zone_pp
status	table_head	ub_spacing	windows	ymultiply	zones
stcbmat	table_size	ubiquitous	wipp	yrange	zpres
stcon	tadd	ucohesion	write	yresid	
steady	tan	ucs	x	yreverse	
step	tcontour	udilation	x0	ystr	

By default, user-defined variables represent single numbers or strings. Multidimensional arrays of numbers or strings may be stored if the **ARRAY** statement is used. Section 2.3.1 defines the way arrays are created and used. At present, there is no explicit printout or input facility for arrays, but functions may be written in *FISH* to perform these operations. For example, the contents of a two-dimensional array (or matrix) may be initialized and printed, as shown in Example 2.1:

Example 2.1 Initializing and printing FISH arrays

```
def afill           ; fill matrix with random numbers
  array var(4,3)
  loop m (1,4)
    loop n (1,3)
      var(m,n) = urand
    end_loop
  end_loop
end
def ashow           ; display contents of matrix
  loop m (1,4)
    hed = '   '
    msg = '  '+string(m)
    loop n (1,3)
      hed = hed + '            '+string(n)
      msg = msg + '  '+string(var(m,n))
    end_loop
    if m = 1
      dum = out(hed)
    end_if
    dum = out(msg)
  end_loop
end
afill
ashow
```

Upon execution, the output is

```
             1            2            3
1  5.7713E-001  6.2307E-001  7.6974E-001
2  8.3807E-001  3.3640E-001  8.5697E-001
3  6.3214E-001  5.4165E-002  1.8227E-001
4  8.5974E-001  9.2797E-001  9.6332E-001
```

2.2.3 Scope of Variables

Variable and function names are recognized globally (as in the BASIC language). As soon as a name is mentioned in a valid *FISH* program line, it is thereafter recognized globally, both in *FISH* code and in *UDEC* commands (for example, in place of a number); it also appears in the list of variables displayed when the **PRINT fish** command is given. A variable may be given a value in one *FISH* function, and used in another function or in a *UDEC* command. The value is retained until it is changed. The values of all variables are also saved by the **SAVE** command and restored by the **RESTORE** command.

2.2.4 Functions: Structure, Evaluation and Calling Scheme

The only object in the *FISH* language that can be executed is the "function." Functions have no arguments; communication of parameters is through the setting of variables prior to invoking the function. (Recall that variables have global scope.) The name of a function follows the **DEFINE** command, and its scope terminates with the **END** command. The **END** command also serves to return control to the caller when the function is executed. (Note that the **EXIT** statement also returns control – see Section 2.3.2.) Consider Example 2.2, which shows function construction and use.

Example 2.2 Construction of a function

```
new
def xxx
  aa  = 2 * 3
  xxx = aa + bb
end
```

The value of **xxx** is changed when the function is executed. The variable **aa** is computed locally, but the existing value of **bb** is used in the computation of **xxx**. If values are not explicitly given to variables, they default to zero (integer). It is not necessary for a function to assign a value to the variable corresponding to its name. The function **xxx** may be invoked in one of several ways:

(1) as a single word **xxx** on a *FISH* input line;

(2) as the variable **xxx** in a *FISH* formula – e.g.,

```
new_var = (sqrt(xxx) / 5.6)^4;
```

(3) as a single word **xxx** on a *UDEC* input line;

(4) as a symbolic replacement for a number on an input line (see Section 2.4.1); and

(5) as a parameter to the **SET**, **PRINT** or **HISTORY** command of *UDEC*.

A function may be referred to in another function before it is defined; the *FISH* compiler simply creates a symbol at the time of first mention, and then links all references to the function when it is defined by a **DEFINE** command.

Function calls may be nested to any level (i.e., functions may refer to other functions, which may refer to others, and so on). However, recursive function calls are not allowed (i.e., execution of a function must not invoke that same function). Example 2.3 shows a recursive function call, which is *not* allowed, because the name of the defining function is used in such a way that the function will try to call itself. The example will produce an error on execution.

Example 2.3 A recursive function call

```
new
def load_sum
  load_sum = 0.0
  bi = block_head
  loop while bi # 0
    load_sum = load_sum + b_yforce(bi)
    bi = b_next(bi)
  end_loop
end
```

The same function should be coded as shown in Example 2.4:

Example 2.4 Removing recursion from the function shown in Example 2.3

```
new
def load_sum
  sum = 0.0
  bi = block_head
  loop while bi # 0
    sum = sum + b_yforce(bi)
    bi = b_next(bi)
  end_loop
  load_sum = sum
end
```

The difference between a variable and a function is that a function is always executed whenever its name is mentioned; a variable simply conveys its current value. However, the execution of a function may cause other variables (as opposed to functions) to be evaluated. This effect is useful, for example, when several histories of *FISH* variables are required: only one function is necessary in order to evaluate several quantities, as in Example 2.5.

Example 2.5 Evaluation of variables

```
new
def h_var_1
  zi      = z_near(1, 2)
  h_var_1 = z_sxx(zi)
  h_var_2 = z_sxy(zi)
  h_var_3 = z_syy(zi)
end
```

The *UDEC* commands to request histories might be

```
hist h_var_1
hist h_var_2
hist h_var_3
```

The function **h_var_1** would be executed by the *UDEC*'s history logic every few steps but, as a side effect, the values of **h_var_2** and **h_var_3** would also be computed and used as history variables.

2.2.5 Data Types

There are four data types used for *FISH* variables or function values:

1. ***Integer*** exact numbers in the range -2,147,483,648 to +2,147,483,647;
2. ***Floating-point*** approximate numbers with about six decimal digits of precision, with a range of approximately 10^{-300} to 10^{300}; and
3. ***String*** packed sequence of any printable characters; the sequence may be any length, but it will be truncated in the printout. Strings are denoted in *FISH* and *UDEC* by a sequence of characters enclosed by single quotes (e.g., `'Have a nice day'`). Note that the use of strings in *UDEC* is restricted to titles and file names. See Section 2.4.1.
4. ***Pointer*** (machine address – used for scanning through linked lists. They have an associated type from the object to which the pointer refers, except for the **null** pointer.)

A variable in *FISH* can change its *type* dynamically, depending on the type of the expression to which it is set. To make this clear, consider the assignment statement

```
var1 = var2
```

If **var1** and **var2** are of different types, then two things are done: first, **var1**'s type is converted to **var2**'s type; second, **var2**'s data are transferred to **var1**. In other languages, such as FORTRAN or C, the *type* of **var1** is not changed, although data conversion is done. By default, all variables in *FISH* start their life as integers. However, a statement such as

```
var1 = 3.4
```

causes **var1** to become a floating-point variable when it is executed. The current type of all variables may be determined by giving the *UDEC* command **PRINT fish** – the types are denoted in the printout.

The dynamic typing mechanism in *FISH* was devised to make programming easier for non-programmers. In languages such as BASIC, numbers are stored in floating-point format, which can cause difficulties when integers are needed for, say, loop counters. In *FISH*, the type of the variable adjusts naturally to the context in which it is used. For example, in the code fragment

```
n  = n + 2
xx = xx + 3.5
```

the variable **n** will be an integer and will be incremented by exactly 2, and the variable **xx** will be a floating-point number, subject to the usual truncation error but capable of handling a much bigger dynamic range. The rules governing type conversion in arithmetic operations are explained in Section 2.2.6. The type of a variable is determined by the type of the object on the right-hand side of an assignment statement; this applies both to *FISH* statements and to assignments done with the *UDEC* **SET** command. Both types of assignment may be used to change the type of a variable according to the value specified:

1. An integer assignment (digits 0-9 only) will cause the variable to become an integer (e.g., **var1** = `334`).
2. If the assigned number has a decimal point or an exponent denoted by "e" or "E," then the variable will become a floating-point number (e.g., **var1** = `3e5`; **var2** = `-1.2`).
3. If the assignment is delimited by single quotes, the variable becomes a string, with the "value" taken to be the list of characters inside the quotes (e.g., **var1** = `'Have a nice day'`).

Type conversion is also done in assignments involving predefined variables or functions; these rules are presented in Section 2.5.

2.2.6 Arithmetic: Expressions and Type Conversions

Arithmetic follows the conventions used in most languages. The symbols

```
^ / * - +
```

denote exponentiation, division, multiplication, subtraction and addition, respectively, and are applied in the order of precedence given. Arbitrary numbers of parentheses may be used to render explicit the order of evaluation; expressions within parentheses are evaluated before anything else. Inner parentheses are evaluated first. As an example, *FISH* evaluates the following variable **xx** as 133.

```
xx = 6/3*4^3+5
```

The expression is equivalent to

```
xx = ( (6/3) * (4^3) ) + 5
```

If there is any doubt about the order in which arithmetic operators are applied, then parentheses should be used for clarification.

If *either* of the two arguments in an arithmetic operation is of floating-point type, then the result will be floating-point. If *both* of the arguments are integers, then the result will be integer. It is important to note that the division of one integer by another causes truncation of the result (for example, 5/2 produces the result 2, and 5/6 produces the result 0).

2.2.7 Strings

There are two main *FISH* intrinsic functions that are available to manipulate strings:

in(*var*) prints out variable ***var*** if it is a string, or the message "`Input?`" if it is not, and then waits for input from the keyboard. The returned value depends on the characters typed. *FISH* tries to decode the input first as an integer and then as a floating-point number. The returned value will be of type *int* or *float* if a *single* number that can be decoded as integer or floating-point, respectively, has been entered.

If the characters entered by the user cannot be interpreted as a single number, then the returned value will be a string containing the sequence of characters. The user's *FISH* function can determine what has been returned by using the function **type( )**.)

string(*var*) converts ***var*** to type *string*.

One use of these functions is to control interactive input and output. Example 2.6 demonstrates this for user-supplied input parameters for Young's modulus and Poisson's ratio.

Example 2.6 Control of interactive input

```
def in_def
  xx = in(msg+'('+'default:'+string(default)+'):')
  if type(xx) = 3
    in_def = default
  else
    in_def = xx
  end_if
end
def moduli_data
  default = 1.0e9
```

```
  msg='Input Young's modulus '
  y_mod = in_def
;
  default = 0.25
  msg='Input Poisson's ratio '
  p_ratio = in_def
  if p_ratio = 0.5 then
    ii = out(' Bulk mod is undefined at Poisson's ratio = 0.5')
    ii = out(' Select a different value --')
    p_ratio = in_def
  end_if
;
  s_mod = y_mod / (2.0 * (1.0 + p_ratio))
  b_mod = y_mod / (3.0 * (1.0 - 2.0 * p_ratio))
end
moduli_data
;
block 0,0  0,10  10,10  10,0
gen edge 10
zone model elastic
zone bulk=b_mod shear=s_mod
print b_mod s_mod
print prop bulk
print prop shear
```

The only arithmetic operation that is valid for string variables is addition; as demonstrated in Example 2.6, this causes two strings to be concatenated.

It is invalid for only one argument in an arithmetic operation to be a string variable. The intrinsic function **string( )** must be used if a number is to be included as part of a string variable (see variable **xx** in Example 2.6). Also note the use of intrinsic function **type( )**, which identifies the type of argument (see Section 2.5.4).

2.2.8 Deleting and Redefining FISH Functions

A *FISH* function in *UDEC* with the same name as a previously defined *FISH* function will overwrite the previous function; local variables defined by the previous function will still exist. *FISH* functions which call the old function will not call the new function.

Example 2.7 Attempting to redefine a FISH function

```
def joe
  ii = out(' a function')
end
```

```
def fred
  joe
end
fred                  ; ...  old message will appear
def joe
  ii = out(' a new function')
end
joe
; however joe as called by fred no longer exists
fred
```

2.3 *FISH* Statements

There are a number of reserved words in the *FISH* language; they must not be used for user-defined variable or function names. The reserved words, or statements, fall into three categories, as explained below.

2.3.1 Specification Statements

The following words are normally placed at the beginning of a *FISH* function. They alter the characteristics of the function or its variables, but do not affect the flow of control within the function. They are only interpreted during compilation.

ARRAY ***var1(n1, n2 . . .) <var2(m1, m2 . . .)> <var3(p1, p2 . . .)> . . .***

This statement permits arrays of any dimension and size to be included in *FISH* code.

In the above specification, ***var1*** is any valid variable name and ***n1, n2*** . . . are *either* actual integers *or* single user-defined variables (not expressions) that have integer values at the time the **ARRAY** statement is processed. There may be several arrays specified on the same line (e.g., ***var2***, above); the number of dimensions may be different for each array. The **ARRAY** statement is a specification and is acted on during compilation, not execution (it is ignored during execution).

1. The given name may be an *existing* single variable. If so, it is converted to an array and its value is lost. If the name does not already exist, it is created.
2. The given name may not be that of a function or the name of an existing array (i.e., arrays cannot be redefined).
3. The given dimensions (***n1, n2,*** . . .) must be positive integers, or evaluate to positive integers (i.e., indices start at 1, not 0).
4. There is no limit to the number and size of the array dimensions, except memory capacity and the maximum line length.

Array variables take any type (integer, float or string), according to the same rules governing single variables. They are used exactly like single variables, except that they are always followed by an argument (or index) list enclosed by parentheses. In use (as opposed to in specification), array indices may be integer expressions. For example,

```
var1 = (abc(3,nn+3,max(5,6)) + qqq) / 3.4
```

is a valid statement if **abc** is the name of a three-dimensional array. Arrays may appear on both sides of an assignment, and arrays may be used as indices of other arrays.

Some restrictions apply to the use of array names in *UDEC* command lines:

(1) The command **PRINT fish** prints the legend (array) if the corresponding symbol is an array, together with the array dimensions.

(2) **PRINT `name`** (where **`name`** is a *FISH* array name) simply prints out the maximum array indices.

(3) The use of a *FISH* array name as the source or destination for a number in the **SET** command is *not* allowed, and prompts an error message (e.g., **SET grav = `name`**, where **`name`** is a *FISH* array name).

(4) **PRINT `name` *index*** prints the value stored in **`name` *(index)***.

WHILESTEPPING

If this statement appears anywhere within a user-defined function, then the function is always executed automatically at the start of every *UDEC* step. The **WHILESTEPPING** attribute can later be disabled with the use of the **SET fishcall 0 remove** command (see Section 2.4.4).

The **fishcall** (see the **SET fishcall** command) statement provides more flexibility and control than the **WHILESTEPPING** command, and its use is preferred.

Synonym: **WHILE_STEPPING**

2.3.2 Control Statements

The following statements serve to direct the flow of control during execution of a *FISH* function. Their position in the function is of critical importance, unlike the specification statements described above.

DEFINE function-name

END The *FISH* program between the **DEFINE** and **END** commands is compiled and stored in *UDEC*'s memory space. The compiled version of the function is executed whenever its name is mentioned, as explained in Section 2.2.4. The function name (which should be chosen according to the rules in Section 2.2.2) does not need to be assigned a value in the program section that follows. Defining a *FISH* function within another function is not allowable.

CASEOF ***expr***

CASE ***n***

ENDCASE The action of these control statements is similar to the FORTRAN-computed GOTO or C's SWITCH statement. It allows control to be passed rapidly to one of several code segments, depending on the value of an index. The use of the keywords is illustrated in Example 2.8.

Synonym: **CASE_OF** **END_CASE**

Example 2.8 Usage of the* CASE *construct

```
caseof expr
    ;..................... default code here
  case i1
    ;..................... case i1 code here
  case i2
    ;..................... case i2 code here
  case i2
    ;..................... case i3 code here
endcase
```

The object ***expr*** following **CASEOF** can be any valid algebraic expression; when evaluated, it will be converted to an integer. The items ***i1***, ***i2***, ***i3***, . . . must be integers (not symbols) in the range 0 to 255. If the value of ***expr*** equals ***i1***, then control jumps to the statements following the ***CASE i1*** statement; execution then continues until the next **CASE** statement is encountered. Control then jumps to the code following the **ENDCASE** statement; there is no "fall-through," as in the C language. Similar jumps are executed if the value of ***expr*** equals ***i2***, ***i3***, and so on. If the value of ***expr*** does not equal the numbers associated with any of the **CASE** statements, then any code immediately following the **CASEOF** statement is executed, with a jump to **ENDCASE** when the first **CASE** is encountered. If the value of ***expr*** is less than zero or greater than the greatest number associated with any of the **CASE**s, then an execution error is signaled, and processing stops. The numbers ***n*** (e.g., ***i1***, ***i2***, ***i3***) need not be sequential or contiguous, but no duplicate numbers may exist.

CASEOF . . . **ENDCASE** sections may be nested to any degree; there will be no conflict between **CASE** numbers in the different levels of nesting (e.g., several instances of **CASE 5** may appear, provided that they are all associated with different nesting levels). The use of **CASE** statements allows rapid decisions to be made (much more quickly than for a series of **IF** . . . **ENDIF** statements). However, the penalty is that some memory is consumed; the amount of memory depends on the maximum numerical value associated with the **CASE** statements. The memory consumed is one plus the maximum **CASE** number in double-words (four-byte units).

IF ***expr1 test expr2*** **THEN**

ELSE

ENDIF These statements allow conditional execution of *FISH* code segments; **ELSE** is optional and the word **THEN** may be omitted if desired. The item ***test*** consists of one symbol or symbol pair:

= # > < >= <=

The meanings are standard, except for #, which means "not equal." The items ***expr1*** and ***expr2*** are any valid algebraic expressions (which can involve functions, *UDEC* variables, etc.). If the test is true, then the statements immediately following **IF** are executed until **ELSE** or **ENDIF** is encountered. If the test is false, the statements between **ELSE** and **ENDIF** are executed if the **ELSE** statement exists; otherwise, control jumps to the first line after **ENDIF**. All the given ***test*** symbols may be applied when expressions ***expr1*** and ***expr2*** evaluate to integers or floating-point values (or a mixture). If *both* expressions evaluate to strings, then only two tests are valid: = and #. All other operations are invalid for strings. Strings must match exactly, for equality. Similarly, both expressions may evaluate to pointers, but only = and # tests are valid.

IF . . . **ELSE** . . . **ENDIF** clauses can be nested to any depth.

Synonym: **END_IF**

EXIT This statement causes an unconditional jump to the end of the current function.

EXIT SECTION

This statement causes an unconditional jump to the end of a **SECTION**; *FISH* program sections are explained below.

LOOP ***var (expr1, expr2)***

ENDLOOP or

LOOP **WHILE** ***expr1 test expr2***

ENDLOOP

The *FISH* program lines between **LOOP** and **ENDLOOP** are executed repeatedly until certain conditions are met. In the first form, which uses an integer counter, ***var*** is given the value of ***expr1*** initially, and is incremented by 1 at the end of each loop execution until it obtains the value of ***expr2***. Note that ***expr1*** and ***expr2*** (which may be arbitrary algebraic expressions) are evaluated at the start of the loop; redefinition of their component variables within the loop has no effect on the number of loop executions. ***var*** is a single integer variable; it may be used in expressions within the loop (even in functions called from within the loop), and may even be redefined.

In the second form of the **LOOP** structure, the loop body is executed while the test condition is true; otherwise, control passes to the next line after the **ENDLOOP** statement. The form of ***test*** is identical to that described for the **IF** statement. The expressions may involve floating-point variables as well as integers; the use of strings and pointers is also permitted under the same conditions that apply to the **IF** statement.

The two forms of the **LOOP** structure may be contrasted. In the first, the test is done at the end of the loop (so there will be at least one pass through the loop); in the second, the test is done at the start of the loop (so the loop will be bypassed if the test is false initially). Loops may be nested to any depth.

Synonym: **END_LOOP**

SECTION

ENDSECTION

The *FISH* language does not have a "GO TO" statement. The **SECTION** construct allows control to jump forward in a controlled manner. The statements **SECTION** . . . **ENDSECTION** may enclose any number of lines of *FISH* code; they do not affect the operation in any way. However, an **EXIT SECTION** statement within the scope of the section so defined will cause control to jump directly to the end of the section. Any number of these jumps may be embedded within the section. The **ENDSECTION** statement acts as a label, similar to the target of a GO TO statement in C or FORTRAN. The logic is cleaner, however, because control may not pass to anywhere outside the defined section, and flow is always "downward." *Sections may not be nested;* there may be many sections in a function, but they must not overlap or be contained within each other.

Synonym: **END_SECTION**

2.3.3 UDEC Command Execution

COMMAND

ENDCOMMAND

UDEC commands may be inserted between this pair of *FISH* statements; the commands will be interpreted when the *FISH* function is executed. There are a number of restrictions concerning the embedding of *UDEC* commands within a *FISH* function. The **NEW** and **RESTORE** commands are not permitted to be invoked from within a *FISH* function. The lines found inside a **COMMAND** – **ENDCOMMAND** pair are simply stored by *FISH* as a list of symbols; they are not checked at all, and the function must be executed before any errors can be detected. There is an 80 character limit to any command in this section.

Comment lines (starting with ;) are taken as *UDEC* comments instead of *FISH* comments. It may be useful to embed an explanatory message within a function, to be printed out when the function is invoked. If the echo mode is off (**SET echo = off**), then any *UDEC* commands coming from the function are not displayed to the screen or recorded to the log file.

A *FISH* function may not be defined within a **COMMAND** – **ENDCOMMAND** section (or within another *FISH* function). A **CALL** command may not be used within a **COMMAND** – **ENDCOMMAND** section.

Synonym: **END_COMMAND**

2.4 Linkages to *UDEC*

2.4.1 Modified UDEC Commands

The following list contains all of the *UDEC* commands that refer *directly* to *FISH* variables or entities. There are many other ways that *UDEC* and *FISH* may interact (described in Section 2.4.2).

HISTORY *FISH* symbol

causes a history of the *FISH* variable or function to be taken during stepping. If *FISH* symbol is a function, then it will be evaluated every time histories are stored (controlled by **HISTORY ncyc** command); it is *not* necessary to register the function with a **fishcall**. If *FISH* symbol is a *FISH* variable, then its *current* value will be taken, so be careful when using variables (rather than functions) for histories. The history may be plotted in the usual way.

PRINT **fish**

prints out a list of *FISH* symbols, and either their current values or an indication of their type. Variables with names that start with a dollar sign ($) are not printed with the **PRINT fish** command, but they may be printed with the command **PRINT $fish**.

PRINT **fishcall**

prints the current associations between **fishcall** ID numbers and *FISH* functions (see Section 2.4.4).

SET **fishcall** ***n*** <**rem**ove> ***name***

The *FISH* function ***name*** will be called in *UDEC* from a location determined by the value of the **fishcall** ID number ***n***. The currently assigned ID numbers are listed in Table 2.2. When placed before the *FISH* function ***name***, the optional keyword **remove** causes the *FISH* function to be removed from the list.

TITLE <'string'>

changes the stored title (used on plots, for example) to the value of the *FISH* string variable ***str***. Note that the variable name must *not* be in single quotes.

2.4.2 Execution of FISH Functions

In general, *UDEC* and *FISH* operate as separate entities: *FISH* statements cannot be given as *UDEC* commands, and *UDEC* commands do not work directly as statements in a *FISH* program. However, the two systems may interact in many ways; some of the more common ways are listed below.

1. *Direct use of function* – A *FISH* function is executed at the user's request by giving its name on an input line. Some typical uses are to generate geometry, set up a particular profile of material properties, or initialize stresses in some fashion.

2. *Use as a history variable* – When used as the parameter to a **HISTORY** command, a *FISH* function is executed at regular times throughout a run (whenever histories are stored).

3. *Automatic execution during stepping* – If a *FISH* function makes use of the generalized **fishcall** capability (or contains the **WHILESTEPPING** statement), then it is executed automatically at every step in *UDEC*'s calculation cycle, or whenever a particular event occurs. (See Section 2.4.4 for a discussion on **fishcall**.)

4. *Use of function to control a run* – Since a *FISH* function may issue *UDEC* commands (via the **COMMAND** statement), the function can be used to "drive" *UDEC* in a way that is similar to that of a controlling data file. However, the use of a *FISH* function to control operation is much more powerful, since parameters to commands may be changed by the function.

The primary way of executing a *FISH* function from *UDEC* is to give its name as *UDEC* input. In this way, *FISH* function names act just like regular commands in *UDEC*. However, no parameters may follow the function name so given. If parameters are to be passed to the function, then they must be set beforehand with the **SET** command.

There is another important link between *FISH* and *UDEC*: a *FISH* symbol (variable or function name) may be substituted *anywhere* a number is expected in a *UDEC* command. This is a very powerful feature because data files can be set up with symbols instead of actual numbers. Example 2.9 shows how a data file that is independent of problem geometry can be constructed. Parameters that are used in *FISH* functions to generate geometry variables can be specified.

Example 2.9 FISH function with generic zone handling capability

```
new

def setup
   length   = x_right - x_left
   height   = y_top - y_bottom
   x_centre = x_left + length / 2.0
   y_centre = y_bottom + height / 2.0
   radius   = (diam_frac * min(length, height)) / 2.0
   edge_len = edge_frac * radius

   SMOOTH   = 24
   MEDIUM   = 16
   ROUGH    =  8
end

; function input ---
set x_left = -15.0  x_right = 25.0  y_bottom = -50.0 y_top = 0.0
set diam_frac = 0.7 edge_frac = 0.5
setup

; create model
block x_left, y_bottom  x_left, y_top  x_right, y_top  x_right, y_bottom
crack x_centre, y_bottom  x_centre, y_top
crack x_left, y_centre  x_right, y_centre
tunnel x_centre, y_centre, radius, MEDIUM
gen edge edge_len
delete annulus x_centre, y_centre 0 radius
plot block zone hold
```

Example 2.9 illustrates several of the points made above: the function **`setup`** is invoked by giving its name on a line; the parameters controlling the function are given beforehand with the **SET** command; there are no significant numerical values in the *UDEC* input – they are all replaced by symbols.

String variables may be used in a similar way, but their use is much more restricted than the use of numerical variables. A *FISH* string variable may be substituted (1) wherever a file name is required, or (2) as a parameter to the **TITLE** command. In these cases, single quotes are not placed around the string, so that *UDEC* can distinguish between a literal name and a variable standing for a name. Example 2.10 illustrates the syntax.

Example 2.10 Using string variables

```
new
def xxx
  name1 = 'abc.log'
  name2 = 'This is run number ' + string(n_run)
  name3 = 'abc' + string(n_run) + '.sav'
end
set n_run = 3
xxx
set log = name1
title name2
save name3
```

The intrinsic function **string( )** is described in Sections 2.2.7 and 2.5.4; it converts a number to a string.

Another important way to use a *FISH* function is to control a *UDEC* run or a series of *UDEC* operations. *UDEC* commands are placed within a **COMMAND** . . . **ENDCOMMAND** section in the function. The whole section may be within a loop, and parameters may be passed to *UDEC* commands. This approach is illustrated in Example 2.11, in which 10 complete runs are done, each with a different value of joint friction angle.

Example 2.11 Controlling a series of UDEC runs

```
new
def series
  new_fric = 40.0
  step_lim = 3000
  i_gp = gp_near(0,20)
  loop nn (1, 10)
     ii = out('friction angle = ' + string(new_fric))
     command
       prop mat = 1 jfric = new_fric
       reset vel
       reset disp jdisp rot time
       solve step = step_lim
     endcommand
     xtable(1,nn) = new_fric
     ytable(1,nn) = log(abs(gp_ydis(i_gp)))
     new_fric = new_fric - inc_fric
  endloop
end
```

```
block 0 0 0 20 20 20 20 0
crack 0 3 20 13
gen edge 2.5
bound yvel 0.0 range 0 20 -.1 .1
bound xvel 0.0 range -.1 .1 0 2
prop  mat 1 dens 2700  bulk 1e9 shear .7e9
prop jmat 1 jkn 1.33e10 jks 1.33e10 jfric 45
set grav 0 -10
set inc_fric = 2
hist solve_rat type 1
solve
series
plot table 1 both hold
```

For each run (i.e., execution of the loop), all model variables are reset and the friction angle is redefined. The results are summarized in a table in which the log of incremental displacement is plotted against friction angle (Figure 2.1). The stability limit is seen to be about 26°. The table functions **xtable** and **ytable** are described in Section 2.5.5.1.

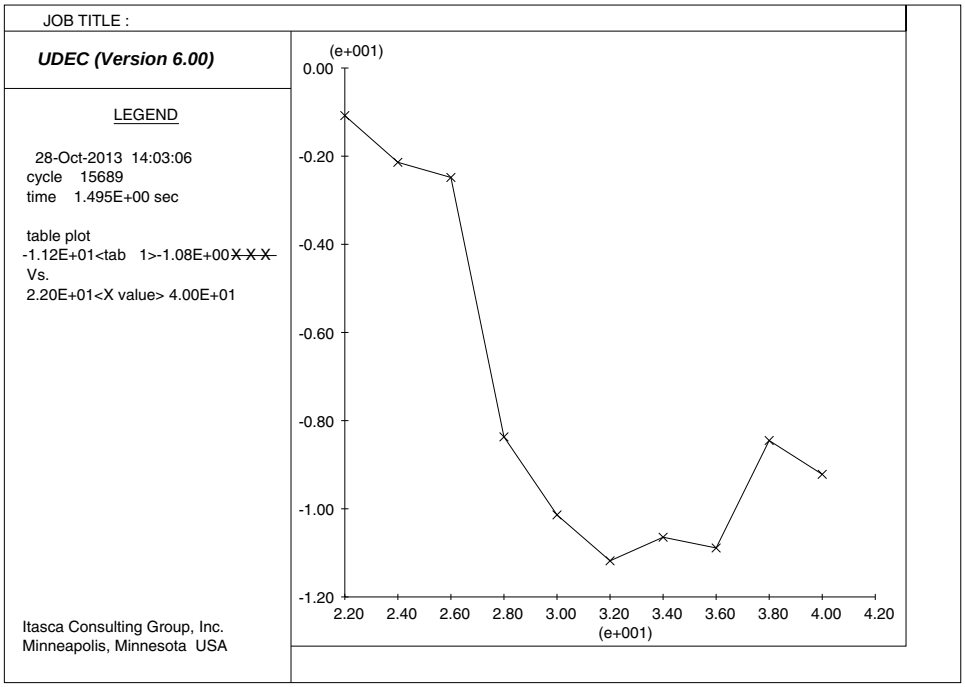

Figure 2.1 Plot of log of incremental displacement versus friction angle

2.4.3 Error Handling

UDEC has a built-in error-handling facility that is invoked when some part of the program detects an error. There is a scheme for returning control to the user in an orderly fashion, no matter where the error may have been detected.

2.4.4 Fishcall

FISH functions may be called from several places in the *UDEC* program. The form of the command is

SET **fishcall** ***n*** <**remove**> ***name***

Setting a **fishcall** causes the *FISH* function ***name*** to be called from *UDEC*, from a location determined by the value of ID number ***n***. Currently, the ID numbers shown in Table 2.2 are assigned.

The ID number 0 also corresponds to functions that contain the **WHILE_STEPPING** statement (i.e., these functions are automatically mapped to ID 0). Any number of functions may be associated with the same ID number (although the order in which they are called is undefined; if the order is important, then one master function should be called, which then calls a series of sub-functions). Also, any number of ID numbers may be associated with one *FISH* function. In this case, the same function will be invoked from several places in the host code.

Parameters may be passed to *FISH* functions called by using the intrinsic function **FC_ARG(*n*)**, where ***n*** is an argument number. The meanings of the parameters (if any) are listed in Table 2.2. For example, for **fishcall** 2, the value passed as **FC_ARG(*0*)** is the index of the contact about to be deleted.

The printout keyword **fishcall** (the abbreviation is **fishc**) lists the current associations between ID numbers and *FISH* functions (i.e., **PRINT fishcall**).

The **SET fishcall** command normally adds the given name to the list already associated with the given ID number. However, the keyword **remove**, placed before the *FISH* name, causes the *FISH* function to be removed from the list. For example,

```
set fishcall 2 remove xxx
```

will remove the association between function **xxx** and ID number 2. Note that a *FISH* function may be associated twice (or more) with the same ID number. In this case, it will be called twice (or more). The **remove** keyword will remove only one instance of the function name. Functions that are associated with a **fishcall** cannot be redefined without using the **remove** keyword first.

Table 2.2 Assigned **fishcall** *IDs*

ID	Location	Argument 0
0	beginning of calculation cycle	
1	when contact created	contact index
2	when contact deleted	contact index
3	when Jcons5 contacts crack	contact index

These numbers are given symbolic macro names in the "FISHCALL.FIS" file, the contents of which are listed in Example 2.12. The symbolic names should be used instead of actual numbers, so that assignments may be changed in the future without the need to change existing *FISH* functions.

Example 2.12 Listing of "FISHCALL.FIS"

```
def fcall_toks
; Tokens for FishCall numbers ...
   FC_CYC_MOT     = 0
   FC_CONT_CREATE = 1
   FC_CONT_DEL    = 2
   FC_JCON5_CRACK = 3
   FC_RESID_CRACK = 3
   FC_Local_rein  = 4
   FC_thermal     = 5
end
fcall_toks
```

The data file in Example 2.13 illustrates the use of a **fishcall**. Two blocks are moved toward each other at a relatively high velocity. When the blocks touch, the applied velocity is removed; this prevents a contact overlap error from occurring. The *FISH* function **stop_loading** is invoked with a **fishcall** that is triggered when a contact is created, and sets the block velocities to zero.

Example 2.13 Illustration of* fishcall *use

```
new
call fishcall.fis
round 0.001
bl (-0.05,-0.1)  (-0.05,0.1)  (0.25,0.1)  (0.25,-0.1)
crack -1 0 1 0
crack  0 0.1  0 0
crack  0.2 0.1  0.2 0
crack  0.1 0.1  0.1 0
crack  0.105 0.1 0.105 0
```

```
del  -0.05 0  0 0.1
del  0.2 0.25  0 0.1
del  0.1 0.105 0 0.1
;
gen  0 1  -1 0  quad 0.4 0.11
gen  0 1   0 1  quad 0.07 0.11
;
prop  mat=1   d=2.60e-3  k=45000    g=30000
;
joint model area
set jcondf  area
joint  jkn=40000  jks=40000 jfric=30
;
bound  xvel=0  range  -0.06,-0.04 -1,1
bound  xvel=0  range   0.24,0.26  -1,1
bound  yvel=0  range     -1,1  -0.11,-0.09
bound stress (0,0,-10) range  -1,1  0.09,0.11
;
hist solve_rat type 1
solve
;
; apply shear load by imposing x-velocity on top blocks
bou  xvel=0.1  range -.01,.101 -.01,.11
bou xvel =-0.1 range .104,.21  -.01,.11
def stop_loading
   ii=out(' ')
   ii=out(' Contact created! Address = '+string(fc_arg(0)))
   command
      bou xvel=0.0  range atblock .025 .05
      bou xvel=0.0  range atblock .150 .05
   endcommand
end
set fishcall FC_CONT_CREATE stop_loading
hist xvel 0.1,0.1
hist xdis 0.2,0.1
step 10000
```

2.5 Predefined Functions, Variables and Arrays

There are certain functions and variables that are built into *FISH*, and the names of these entities must be avoided when naming user-defined variables or functions. This section describes all predefined entities. The entities are organized in several categories: scalars, model variables, general intrinsic functions, table functions and memory-access functions. In some cases, an entity is listed under more than one category, as appropriate.

2.5.1 UDEC-Specific Scalar Variables

The variables listed in this category have a single value, and are specifically related to internal *UDEC* data structures or the solution process. An asterisk (*) denotes that the variable may be assigned a value within a user-written function; otherwise, the variable's value may only be tested, not set.

1. Indices to Data Arrays (integer type)

block_head	index to list of blocks
bou_head	index to list of boundary corners
cable_elem_head	index to list of cable elements
cable_node_head	index to list of cable nodes
contact_head	index to list of contacts
domain_head	index to list of domains
ibou_head	index to list of interior boundary corners
outer_domain	index of outer domain
r_head	index to list of reinforcement elements
str_elem_head	index to list of structural elements
str_int_head	index to list of structural node interfaces
str_node_head	index to list of structural nodes
sup_head	index to list of support elements
table_head(*i*)	index to start of table ***i***
tgps_head	start of linked list of gridpoint thermal sources

2. General Variables (floating point type unless declared otherwise)

crtdel	creep timestep
crtime	creep time
cycle	synonym for **step** – integer
fluid_bulk	* fluid bulk modulus
fluid_density	* fluid density
fracb	* fraction of critical block timestep
fracz	* fraction of critical zone timestep
ftime	fluid flow time
grav_x	* synonym for **xgrav**
grav_y	* synonym for **ygrav**
sol_fmag	sum of gridpoint forces
sol_fob	sum of out-of-balance gridpoint forces
sol_ratio	value of current out-of-balance force ratio limit
sol_rloc	maximum ratio of out-of-balance total forces for a single gridpoint
sol_rmax	current solve ratio
step	step (cycle) number – integer
tdel	timestep
thdt	thermal timestep
thtime	thermal time
time	mechanical time
unbal	maximum out-of-balance force in model
xgrav	* x-component of gravity
ygrav	* y-component of gravity

2.5.2 General Scalar Variables

The variables listed in this category have a single value, and are not specifically related to *UDEC*; they are general-purpose scalars. An asterisk (*****) denotes that a variable may be assigned a value within a user-written function. Otherwise, the variable's value may only be tested, not set. The variables listed below are of *floating-point* type unless declared otherwise.

clock	number of hundredths-of-a-second from midnight (integer)
cycle	current cycle (step) number (integer)
degrad	$\pi/180$ (used to convert degrees to radians – for example, `a = cos(30*degrad)` gives the cosine of 30°)
grand	random number drawn from normal distribution, with a mean of 0.0 and standard deviation of 1.0. The mean and standard deviation may be modified by multiplying the returned number by a factor and adding an offset.
null	link-list terminator; it is of type *pointer*
pi	π
step	current step (cycle) number (integer)
unbal	maximum unbalanced force
urand	random number drawn from uniform distribution between 0.0 and 1.0

2.5.3 *UDEC-Specific Model Variables*

The following reserved names refer to *UDEC*-specific variables that require the integer indices to be specified in parentheses immediately following the name. For example, the x-coordinate of a block with block index ***bi*** is obtained from **b_x(*bi*)**, where ***bi*** is an integer representing a block index number.

The index value for any *UDEC* block, contact, corner, domain, gridpoint or zone may be found by executing the appropriate function, given a position specified by x, y global coordinates. The functions are listed below.

bi = **b_near(*x, y*)**	index of block closest to (x, y)
bci = **bou_near(*x, y*)**	index of boundary corner closest to (x, y)
ci = **c_near(*x, y*)**	index of contact closest to (x, y)
cri = **cor_near(*x, y*)**	index of corner closest to (x, y)
di = **d_near(*x, y*)**	index of domain closest to (x, y). Note: This function will not return the address of the outer domain (use **outer_domain**).
gi = **gp_near(*x, y*)**	index of gridpoint closest to (x, y)
zi = **z_near(*x, y*)**	index of zone closest to (x, y)

Alternatively, the list of objects may be scanned by using headers (e.g., **block_head** and **contact_head**) listed in Section 2.5.1, and the next-item pointers (e.g., **gp_next( )** and **z_next( )**) listed below. The determination of index values can be time-consuming, as the entire model must be scanned. It is recommended that, if possible, the functions above not be used directly within *FISH* functions that are executed during stepping.

The following variable names must be spelled out in full in *FISH* statements; they cannot be truncated, as in *UDEC* commands. An asterisk (*) denotes that the variable can be modified by a *FISH* function; otherwise, its value may only be tested.

Block Variables

b_area(*bi*)	block area
b_bex(*bi*)	index pointing to FDEF data structure
b_cons(*bi*)	* constitutive model number
b_corner(*bi*)	first corner of block
b_dsf(*bi*)	density scaling factor

b_extra(*bi*)	* extra variable available to user
b_fix(*bi*)	* rigid block fix condition (= 1 fixed, = 0 free)
b_gp(*bi*)	index of gridpoint list
b_group(*bi*)	* block group name
b_mass(*bi*)	block mass
b_mat(*bi*)	* material property number
b_moi(*bi*)	moment of inertia
b_mom(*bi*)	* moment
b_next(*bi*)	index of next block in main list from block
b_rvel(*bi*)	* angular velocity of rigid block
b_type(*bi*)	block type: 1 = rigid, 3 = deformable
b_x(*bi*)	x-coordinate of centroid of block
b_xforce(*bi*)	* x-force
b_xload(*bi*)	* applied x-force
b_xvel(*bi*)	* x-velocity of rigid block
b_y(*bi*)	y-coordinate of centroid of block
b_yforce(*bi*)	* y-force
b_yload(*bi*)	* applied y-force
b_yvel(*bi*)	* y-velocity of rigid block
b_zone(*bi*)	index of zone list

Block Material Property Variables

m_bulk(*mn*)	* property **bulk** of property number ***mn***
m_cohesion(*mn*)	* property **cohesion** of property number ***mn***
m_density(*mn*)	* property **density** of property number ***mn***
m_dilation(*mn*)	* property tan (**dilation**) of property number ***mn***
m_friction(*mn*)	* property coefficient of **friction** (not angle) of property number ***mn***
m_shear(*mn*)	* property **shear** of property number ***mn***
m_tension(*mn*)	* property **tension** of property number ***mn***

Boundary Corner Variables

bou_gp(*bci*)	index of gridpoint associated with the boundary corner if > 0, gridpoint is on outer boundary if < 0, gridpoint is on an interior boundary
bou_xreaction(*bci*)	*x*-direction reaction force at boundary corner for fixed velocity boundary
bou_yreaction(*bci*)	*y*-direction reaction force at boundary corner for fixed velocity boundary

Configuration Variables

cf_axi(*ci*)	set =1 if **CONFIG axi** was used
cf_cell(*ci*)	set = 1 if **CONFIG cell** was used
cf_creep(*ci*)	set = 1 if **CONFIG creep** was used
cf_fluid(*ci*)	set = 1 if **CONFIG fluid** was used
cf_p_stress(*ci*)	set = 1 if **CONFIG p_stress** was used
cf_thermal(*ci*)	set = 1 if **CONFIG thermal** was used

Contact Variables

c_b1(*ci*)	block 1 at contact

c_b2(*ci*)	block 2 at contact
c_cons(*ci*)	* constitutive model number
c_d1(*ci*)	index of domain 1 at contact
c_d2(*ci*)	index of domain 2 at contact
c_extra(*ci*)	* extra variable available to user
c_group(*ci*)	* contact group name
c_jex(*ci*)	extension pointer for local properties (used for Barton-Bandis and joint models). 0th offset of extension array is model #. 1 = point 2 = area 3 = cy 4 = residual 7 = Barton-Bandis
c_length(*ci*)	contact length
c_link1(*ci*)	next contact/corner on block 1
c_link2(*ci*)	next contact/corner on block 2
c_mat(*ci*)	* material property number
c_ndis(*ci*)	* normal displacement
c_next(*ci*)	index of next contact in main list from contact ***ci***
c_nforce(*ci*)	* normal force
c_nx(*ci*)	*x*-component of unit normal
c_ny(*ci*)	*y*-component of unit normal
c_sdis(*ci*)	* shear displacement
c_sforce(*ci*)	* shear force

c_thsource(*ci*) Define a thermal source at the location of contact ***ci***. The function returns the link index (***ti***) to the gridpoint thermal source data structure. If a source already exists for the specified contact, the function merely returns the link index (***ti***). The thermal source strength, decay exponent and start time need to be defined using the appropriate gridpoint thermal source *FISH* functions:

tgps_decay(*ti*)	exponent for decaying sources
tgps_strength(*ti*)	strength of the source
tgps_timeth(*ti*)	thermal time at which source was defined

c_type(*ci*) contact type:

1	corner/corner
2	block 1 corner/block 2 edge
3	block 1 edge/block 2 corner

c_x(*ci*) *x*-coordinate of contact

c_y(*ci*) *y*-coordinate of contact

j_model(*ci*) * model of contact for extended and user-defined joint constitutive DLL models. For built-in models, use **c_cons(*ci*)**.

j_prop(*ci, name*) * value of named property. This is valid only for extended and user-defined joint constitutive models. Use **c_mat(*ci*)** for built-in models.

Contact Material Property Variables

m_jcohesion(*mn*) * property **jcohesion** of property number ***mn***

m_jdilation(*mn*) * property **jdilation** of property number ***mn***

m_jfriction(*mn*) * property coefficient of **jfriction** of property number ***mn***

m_jkn(*mn*) * property **jkn** of property number ***mn***

m_jks(*mn*) * property **jks** of property number ***mn***

m_jrescoh(*mn*) * property **jrescoh** of property number ***mn***

m_jrfriction(*mn*) * property coefficient of **jrfriction** of property number ***mn***

m_jrtension(*mn*) * property **jrtension** of property number ***mn***

m_jtension(*mn*) * property **jtension** of property number ***mn***

Corner Variables

cor_block(*cri*) index of host block

cor_bou(*cri*) index of boundary corner array

cor_extra(*cri*) * extra variable available to user

cor_gp(*cri*) index of gridpoint associated with corner

cor_link(*cri*) next corner/contact in clockwise direction

cor_rlink(*cri*) next corner in counterclockwise direction

cor_x(*cri*) x-coordinate

cor_xdis(*cri*) * x-displacement

cor_xvel(*cri*) x-velocity

cor_y(*cri*) y-coordinate

cor_ydis(*cri*) * y-displacement

cor_yvel(*cri*) y-velocity

Domain Variables

d_contact(*di, ic*) If ***ic*** is 0, the function returns the first contact in counterclockwise list around the domain. If ***ic*** is a contact, the function returns the next contact in the domain.

d_extra(*di*) * extra variable available to user

d_fix(*di*) * fixed pressure condition (= 1 fixed, = 0 free)

d_next(*di*) index of next domain in main list from ***di***

d_pp(*di*) * domain fluid pressure

d_vol(*di*) domain volume

d_x(*di*) x-coordinate of domain center

d_y(*di*) y-coordinate of domain center

Gridpoint Variables

gp_bou(*gi*)	index of boundary corner associated with gridpoint
gp_corner(*gi*)	index of corner associated with gridpoint
gp_dsf(*gi*)	density scaling factor
gp_extra(*gi*)	* extra variable available to user
gp_mass(*gi*)	gridpoint mass
gp_next(*gi*)	index of next gridpoint in main list from ***gi***
gp_thmass(*gi*)	returns thermal mass for a gridpoint. Note: *UDEC* stores the inverse of the thermal mass. The function returns the actual thermal mass, which will be different from what would be obtained using an FMEM() function.
gp_x(*gi*)	x-coordinate
gp_xdis(*gi*)	* x-displacement
gp_xforce(*gi*)	* x-force
gp_xvel(*gi*)	* x-velocity
gp_y(*gi*)	y-coordinate
gp_ydis(*gi*)	* y-displacement
gp_yforce(*gi*)	* y-force
gp_yvel(*gi*)	* y-velocity

Gridpoint Thermal Source Functions

tgps_cor(*ti*)	Corner at which the thermal mass is stored. This is the coupled corner, and may not be the corner directly connected to the gridpoint or contact for which the thermal source is specified.
tgps_decay(*ti*)	exponent for decaying sources
tgps_gp(*ti*)	gridpoint to which thermal source is to be applied
tgps_next(*ti*)	next source in list
tgps_strength(*ti*)	strength of the source

tgps_timeth(*ti*) thermal time at which source was defined

tgps_type(*ti*) thermal source type (currently, only 1)

Local Reinforcement Material Properties

r_aexp(*mn*) * axial stiffness exponent of property number ***mn***

r_astiff(*mn*) * axial stiffness of property number ***mn***

r_length(*mn*) * active length of property number ***mn***

r_rfac(*mn*) * reversal factor of property number ***mn***

r_sexp(*mn*) * shear stiffness exponent of property number ***mn***

r_sstif(*mn*) * shear stiffness of property number ***mn***

r_str(*mn*) * ultimate axial failure strain of property number ***mn***

r_uaxial(*mn*) * ultimate axial force limit of property number ***mn***

r_ushear(*mn*) * ultimate shear force of property number ***mn***

Zone Variables

z_biot_c(*zi*) * Biot's constant for zone

z_block(*zi*) index of block associated with zone

z_bulk(*zi*) zone bulk modulus (returns 0.0 if properties are assigned by the **ZONE** command)

z_density(*zi*) * zone density. Note: gridpoint masses are recalculated when the next **CYCLE** command is given.

z_extra(*zi*) * extra variable available to user

z_fsi(*zi, arr*) fills array (4) ***arr*** with the strain increment components. ($1 = \epsilon xx$, $2 = \epsilon xy$, $3 = \epsilon xy$, $4 = \epsilon yy$) Strains are calculated from the current gridpoint displacements.

z_fsr(*zi, arr*) fills array (4) ***arr*** with the strain rate components. ($1 = \dot{\epsilon} xx$, $2 = \dot{\epsilon} xy$, $3 = \dot{\epsilon} xy$, $4 = \dot{\epsilon} yy$) Strain rates are calculated from the current gridpoint velocities.

z_gp(*zi, i*) index of zone gridpoint (***i*** = 1, 2, 3 for three surrounding gridpoints)

z_group(*zi*) * zone group name

z_mass(*zi*) zone mass

z_mat(*zi*) material property number for zone (returns −1 if the properties are assigned by the **ZONE** command)

z_model(*zi*) * model of zone

z_next(*zi*) index of next zone in main list

z_pp(*zi*) * zone pore pressure

z_prop(*zi, name*) * value of named property for zone. This is valid for any models assigned using the **ZONE** command. (For density and Biot's constant, use specific functions.)

z_rot(*zi*) * zone rotation

z_shear(*zi*) zone shear modulus (returns 0.0 if the properties are assigned by the **ZONE** command)

z_state(*zi*) * plasticity state indicator:

0	elastic
1	currently at yield in shear and/or volume
2	currently at yield in tension
4	currently not at yield but has been in the past, in shear
16	ubiquitous joints at yield in shear
32	ubiquitous joints at yield in tension
64	ubiquitous joints currently not at yield but have been in the past, in shear
128	ubiquitous joints previously at yield in tension

z_sxx(*zi*) * xx-stress

z_sxy(*zi*) * xy-stress

z_syy(*zi*) * yy-stress

z_szz(*zi*) * zz-stress

z_x(*zi*)	*x*-coordinate of zone centroid
z_y(*zi*)	*y*-coordinate of zone centroid
z_zex(*zi*)	index of zone extension array extension offsets: 0 – model # (if < 0 = fill material; if > 1000 = cpp model) 1 – plasticity state . . . – property values for zone models

2.5.4 Intrinsic Functions

All functions return floating-point values except for **and**, **or**, **not**, **int** and **type**, which return integers, and **get_mem**, which returns a pointer. The functions **max**, **min**, **abs** and **sgn** return integers if their argument(s) are *all* integer; otherwise, they return as floating-point. All functions must be placed on the right-hand side of an assignment statement, even if the function's return value is of no interest. For example,

```
ii = out(' Hi there!')
```

is a valid way to use the **out** function. In this case, **ii** is not used.

abs(*a*)	absolute value of ***a***
and(*a,b*)	bit-wise logical **and** of ***a*** and ***b***
atan(*a*)	arc-tangent of ***a*** (result is in radians)
atan2(*a,b*)	arc-tangent of ***a*/*b*** (result is in radians). NOTE: ***b*** may be zero.
cos(*a*)	cosine of ***a*** (***a*** is in radians) This function causes an error condition. *FISH*-function processing (and command processing) stops immediately. The message reported is ***string***. This function can be used for assignment only. (***string*** = **error** is not allowed.)
exp(*a*)	exponential of ***a***
fc_arg(*n*)	passes arguments to *FISH* functions where ***n*** is an argument number.
float(*a*)	converts ***a*** to a floating-point number. If it cannot be converted (e.g., if ***a*** is a string), then 0.0 is returned.
get_mem(*nw*)	gets ***nw*** *FISH*-variable objects from *UDEC*'s memory space, and returns the address of the start of the contiguous array of objects (see Section 2.5.5.2).

grand	random number drawn from normal distribution: mean = 0.0; standard deviation = 1.0
in(*s*)	prints out the message contained in string variable ***s***, and then waits for input from the keyboard. The returned value will be of type *int* or *float* if a single number that can be decoded as integer or floating-point, respectively, has been entered. The number should be the only thing on the line. However, if it is followed by a space, comma or parenthesis, then any other characters on the line are ignored. If the characters typed in by the user cannot be interpreted as a single number, then the returned value will be a string containing the sequence of characters.
int(*a*)	converts ***a*** to integer. If it cannot be converted (e.g., if ***a*** is a string), then zero is returned.
ln(*a*)	natural logarithm of ***a***
log(*a*)	base-ten logarithm of ***a***
lose_mem(*nw,ia*)	returns ***nw*** *FISH*-variable objects to *UDEC* for reuse. The parameter ***ia*** is the address of the start of the array of objects; there is no checking done to ensure that ***ia*** is a valid address. The return value is undefined (see Section 2.5.5.2).
max(*a,b*)	returns maximum of ***a***, ***b***.
mem(*memptr*)	returns contents of memory address ***memptr*** (see Section 2.5.5.2).
min(*a,b*)	returns minimum of ***a***, ***b***.
not(*a*)	bit-wise logical **not** of ***a***
null	end of linked-list (pointer variable)
or(*a,b*)	bit-wise logical inclusive **or** of ***a***, ***b***
out(*s*)	prints out the message contained in ***s*** to the screen (and to the log file, if it is open). The variable ***s*** must be of type *string*. The returned value of the function is zero if no error is detected, and 1 if there is an error in the argument (e.g., if ***s*** is not a string).
round(*a*)	converts ***a*** to an integer using arithmetic rounding convention.
sgn(*a*)	sign of ***a*** (returns -1 if $a < 0$; else, 1)

sin(*a*)	sine of ***a*** (***a*** is in radians)
sqrt(*a*)	square root of ***a***
string(*a*)	converts ***a*** to a string. If ***a*** is already of type *string*, then the function simply returns ***a*** as its value. If ***a*** is *int* or *float*, then a character string that corresponds to the number as it would be printed out will be returned. However, no blanks are included in the string.
tan(*a*)	tangent of ***a*** (***a*** is in radians)
type(*e*)	the variable type of ***e*** is returned as an integer with value 1, 2, 3, 4 or 5, according to the type of the argument: *int*, *float*, *string*, *pointer* or *array*, respectively.
urand	random number drawn from uniform distribution between 0.0 and 1.0

2.5.5 Special Functions – Tables and Memory Access

The functions described in the previous section are "conventional" in the sense that they simply return a value, given some parameter(s), or they are executed for some effect. In other words, they always appear on the right-hand side of any assignment statement. In contrast, the functions described in this section may appear on either side of an assignment (**=** sign). They act partly as functions and partly as arrays.

2.5.5.1 Tables

The **table**, **xtable**, **ytable** and **table_size** functions allow *FISH* functions to create and manipulate *UDEC* tables, which are indexed arrays of number pairs used in several of *UDEC*'s commands and operations. However, tables are different from arrays in other programming languages. Tables are dynamic data structures; items may be inserted and appended, and interpolation between values may be done automatically. Consequently, the manipulation of tables by *FISH* is time-consuming. Use them with caution! The action of each function depends on whether it is the *source* or *destination* for a given data item. Hence, each function is described twice.

A table is a list of pairs of floating-point numbers (denoted for convenience as x and y), although the numbers may stand for any variables, not necessarily coordinates. Each table entry (or (x,y) pair) also has a sequence number in the table. However, the sequence number of a given (x,y) pair may change if a new item is inserted in the table. Sequence numbers are integers that start from 1 and go up to the number of items in the table. Each table has a unique identification number, which may be any integer except zero.

There are two distinct ways tables may be used in a *FISH* function. The **table** function behaves in the same way as the regular *UDEC* **TABLE** command (i.e., insertion and interpolation are done

automatically). The other functions, **xtable** and **ytable**, allow items to be added or updated by reference to the sequence numbers; no interpolation or insertion is done.

y = table(n,x) The existing table ***n*** is consulted and a y-value found by interpolation, corresponding to the given value of x. The value of x should lie between two consecutive stored x-values, for the results to be meaningful. An error is signaled if table ***n*** does not exist.

table(n,x) = y An (x,y) pair is *inserted* into the first appropriate place in table ***n*** (i.e., the new item is inserted between two existing items with x-values that bracket the given x-value). The new item is placed at the beginning of the table or appended to the end if the given x is lower than the lowest x or greater than the greatest x, respectively. The number of items in the table is increased by one, following execution of this statement. If table ***n*** does not exist, it is created, and the given item is taken as the first entry. The given statement is equivalent to the *UDEC* command **TABLE *n* insert *x y***. If the given x is identical to the stored x of an (x,y) pair, then the y-value is updated rather than inserted.

x = xtable(n,s) The x-value of the pair of numbers that has sequence number ***s*** in table ***n*** is returned. An error is signaled if table ***n*** does not exist or if sequence number ***s*** does not exist.

xtable(n,s) = x The given value of x is substituted for the stored value of x in the (x,y) pair having sequence number ***s***, in table ***n***. If sequence number ***s*** does not exist, then sufficient entries are appended to table ***n*** to encompass the given sequence number; the given x is then installed. If the given table does not exist, it is created. An error is signaled if ***n*** is given as zero, or if ***s*** is given as zero or negative.

y = ytable(n,s) The action of this statement is identical to the corresponding **xtable** statement except that the y-value of the (x,y) pair is retrieved instead of the x-value.

ytable(n,s) = y The action of this statement is identical to the corresponding **xtable** statement except that the y-value of the (x,y) pair is installed instead of the x-value.

i = table_size(n) The number of entries in table ***n*** is returned in value i.

Since the **xtable** and **ytable** functions can create tables of arbitrary length, they should be used with caution. It is suggested that the **table** function be used in constitutive models where interpolated values are needed from tables. The **xtable** and **ytable** functions are more useful when generating tables.

The following notes may be helpful when using the *FISH* table functions.

1. In large tables, for efficiency, sequence numbers should be scanned in the direction of ascending numbers. *UDEC* keeps track of the last-accessed sequence number for each table; this is used to start the search for the next requested number. If the requested number is smaller than the previous one, the whole table may need to be searched.

2. The **xtable** and **ytable** functions, rather than **table**, should be used to update values in an existing table. Although **table** will update an (x,y) pair if the given x is identical to the stored x, there may be slight numerical errors, which can result in insertion rather than updating.

3. In a *FISH* function that replaces old table values with new values, it is necessary to create the table first, since the action of retrieving old values will produce an error. A complete table may be created (and its entries all set to zero) by a single statement, as illustrated in

   ```
   xtable(4,100) = 0.0
   ```

   If table 4 does not exist, then it is created. 100 entries are also created, each containing (0.0,0.0). Subsequent statements, such as

   ```
   xtable(4,32) = xtable(4,32) + 1.0
   ytable(4,32) = ytable(4,32) + 4.5
   ```

   will update table values but will not alter the length of the table. If the latter statements are executed before table 4 exists, then an error will be detected.

4. Stored values (both x and y) in tables are always floating-point numbers. Given integers are converted to floating-point type before storing. Be careful about precision!

Refer to Example 2.11 for an example in the use of **table** functions.

2.5.5.2 *Special Functions to Access Memory Directly*

The functions **get_mem( )**, **lose_mem( )** and **mem( )** manipulate user-defined structures made up of blocks of *FISH*-type variables (i.e., variables that may be of type integer, floating-point, string or pointer). The "addresses" referred to here are machine addresses and cannot be converted to or from *UDEC* indices, which are used by other intrinsic functions. The functions described here are useful for rapid data manipulation, separate from *UDEC* data.

get_mem(*n*) This function gets ***n*** *FISH*-variable objects from the host's memory space and returns the address of the start of the contiguous array of objects.

lose_mem(*n*,*ad*) This function returns ***n*** *FISH*-variable objects to the host. The parameter ***ad*** is the address of the start of the array of objects; there is no checking done to verify that ***ad*** is a valid address. The returned value is undefined.

mem(*ad*) This function may be used either as a source or destination (i.e., on the right- or left-hand side of an expression):

- ***var* = mem(*ad*)** The value of the *FISH*-variable at address ***ad*** is transmitted to the regular *FISH* variable ***var*** (together with its type).
- **mem(*ad*)=*var*** The value and type of the regular *FISH* variable ***var*** is placed in the *FISH*-variable at address ***ad***.

It is the user's responsibility to make sure that the addresses given contain valid *FISH* variables. During execution, an error message is given if ***ad*** is not of type pointer.

Access to elements of an array of *FISH* variables is via the addition operator; no other arithmetic operation is allowed on a pointer. When an integer ***n*** is added to a pointer, the pointer then points to a variable that is higher by ***n*** positions in the array. For example, suppose we create a 10-object array:

```
head = get_mem(10)
```

The first object (item 0) is accessed by

```
var = mem(head)
```

The fourth object (item 3), for example, is accessed by

```
var = mem(head+3)
```

and so on. The minus operator is not allowed; we can add a negative integer if we want to go backward in an array. The last item in a linked list is denoted by the intrinsic pointer variable **null**; the integer zero should not be used.

All direct manipulation of *UDEC*'s memory should be done with great caution; only experienced programmers should use the memory functions. As an example of the use of direct memory manipulation, the following program does an insertion sort on twenty random floating-point numbers stored in a linked list. A structure of two *FISH* variable objects is created for each generated random number, the number being stored in one of these variables. The other variable is given the address of the next such structure, with the final structure containing a null value, thus forming a linked list.

Each new random number is compared against values in the list, and inserted at the appropriate point. This insertion consists of reassigning the address values contained in the previous item to the address of the new item.

Example 2.14 Insertion sort of 20 random numbers

```
def inserter
  head = null                    ;list head
  loop n (1,20)
    number = urand               ;new random float
    ad = head
    prev = head
    section
      loop while ad # null       ;Scan existing numbers
        if number > mem(ad+1)    ;Exit if we are past
          exit section           ;required location
        end_if
        prev = ad                ;Remember previous object
        ad = mem(ad)
      end_loop
    end_section
    new = get_mem(2)             ;Create a double-object
    if prev = head               ;and link up
      mem(new) = head
      head = new
    else
      mem(new) = mem(prev)
      mem(prev) = new
    end_if
    mem(new+1) = number
  end_loop
;--- now scan list, and print out ---
  count = 1
  ad = head
  loop while ad # null
```

```
    if count < 10                ;a trick to line up
      nn = ' '+string(count)     ;numbers in columns
    else
      nn = string(count)
    end_if
    ii = out(nn+' '+string(mem(ad+1)))
    count = count + 1
    ad = mem(ad)
  end_loop
end
inserter
```

2.5.6 Access to UDEC's Data Structures

Warning! This section is intended for experienced programmers who are familiar with the use of linked lists. The techniques described here are powerful because they provide access to most of the internal data in *UDEC*, but they are dangerous if used without full understanding.

Most of *UDEC*'s data are stored in a single, one-dimensional array. A *FISH* program has access to this array via **imem** and **fmem**, which act like array names for integer and floating-point numbers, respectively. Given index ***iad*** (which *must* be an integer), floating-point (***f***) or integer (***i***) values can be found from

f = **fmem(*iad*)**

i = **imem(*iad*)**

Values can also be inserted in the array:

fmem(*iad*) = ***f***

imem(*iad*) = ***i***

These functions are potentially very dangerous, as any data can be changed in *UDEC*'s main array. Only experienced programmers should use them. No checking is done to verify that ***iad*** is an integer, so the user must be very careful.

Data structures in *UDEC* consist of one or more linked lists, with offsets to individual data items. Index ***iad*** then comprises two components:

```
iad = iaddress + ioffset
```

where ***iaddress*** is the index of the specific object (e.g., block, zone, cable, node, etc.) and ***ioffset*** is the offset number that identifies the specific data item (e.g., block velocity or cable node force). The data items and offsets for all *UDEC* objects are listed in Section 4.

Data in tables can also be manipulated with these functions. The address of table ***n*** can be found by using

table_head(*n*) index to list of ***(x, y)*** pairs in table ***n***.

This function returns ***iad*** – the values of ***ioffset*** are 1 for the x-entry, and 2 for the y-entry. Example 2.15 shows the code from the *FISH* library file "INT.FIS," which integrates the values in an existing table. Please refer to for complete instructions on how to use this function. It is listed here simply to illustrate the manipulation of the table values.

Example 2.15 Using **table_head**

```
def integrate
  command
    table int_out delete
  endcommand
;
  tpnt = table_head(int_in)
  xold = fmem(tpnt+1)
  yold = fmem(tpnt+2)
  val = 0.0
  table(int_out,xold) = val
;
  loop while tpnt # 0
    xnew = fmem(tpnt+1)
    ynew = fmem(tpnt+2)
    val = val + 0.5*(yold + ynew)*(xnew-xold)
    table(int_out,xnew) = val
    xold = xnew
    yold = ynew
    tpnt = imem(tpnt)
  end_loop
;
end
```

2.5.7 Determining Failure States of Zones

UDEC stores a state variable with 16 bits that can be used to represent a maximum of 15 distinct states. Some of the states that are used by constitutive models in *UDEC* are given in Table 2.3:

Table 2.3 Failure states

State	State Value
Failure in shear now	1
Failure in tension now	2
Failure in shear in the past	4
Failure in tension in the past	8
Failure in joint shear now	16
Failure in joint tension now	32
Failure in joint shear in the past	64
Failure in joint tension in the past	128
Failure in volume now	256
Failure in volume in the past	512

Example 2.16 Plasticity states

```
DEF _States
shearnow          = 1         ; 1
tensionnow        = 2         ; 2
shearpast         = 4         ; 3
tensionpast       = 8         ; 4
jointshearnow     = 16        ; 5
jointtensionnow   = 32        ; 6
jointshearpast    = 64        ; 7
jointtensionpast  = 128       ; 8
volumenow         = 256       ; 9
volumepast        = 512       ; 10
;
; Quick reference
;          Model                        States used
;         ------                        -----------
;       Bilinear, Strain-Hardening/
;       Softening Ubiquitous-Joint       1 - 8
;       Cam-Clay                         1 , 3
;       Chsoil                           1 - 4
;       Cpower                           1 - 4
;       Cvisc                            1 - 4
;       Cwipp                            1 - 4
;       Double-Yield                     1 - 4, 9 - 10
;       Drucker-Prager                   1 - 4
;       Hoek-Brown                       1 , 3
;       Mhoek-Brown                      1 - 4
;       Mohr-Coulomb                     1 - 4
;       Strain-Hardening/Softening       1 - 4
;       Ubiquitous-Joint                 1 - 8
;       Pwipp                            1 - 4
;
END
;
_States
```

The numbers listed in Table 2.3 are given symbolic *FISH* names in file "STATES.FIS," the contents of which are listed in Example 2.16. The symbolic names should be used instead of actual numbers, so that assignments can be changed in the future without the need to change existing *FISH* functions.

The data file in Example 2.17 illustrates the process of determining failure state of zones in a model using *FISH* function **z_state**.

Example 2.17 Printing zone plasticity states

```
new
set log  myfile.log
set log on
round .01
block 0,0 0,1 1,1 1,0
gen edge 1
;
change cons 3
prop mat 1 bulk 2e6 shea 2e3 coh 2e4 ten 2e3 dens 2000
prop jmat = 1 jkn 1e3 jks 1e3
gravity 0,-10
;
boun -.1 1.1 0.9 1.1 stress 0,9,-3e2
boun -.1 1.1 -.1 .1 yvel 0
;
step 100
;
call states.fis  ; failure states defined as FISH variables
DEF _querystate
  ;
  iab = block_head
  loop while iab # 0
    iaz = b_zone(iab)
    loop while iaz # 0
      i_mess = 'zone '+string(iaz)
      curr_state = z_state(iaz)
      if and(curr_state, shearnow) # 0 then
        i_mess = i_mess+' shear'
      else
        if and(curr_state, shearpast) # 0 then
          i_mess = i_mess+' shear_past'
        endif
      endif
      if and(curr_state, tensionnow) # 0 then
        i_mess = i_mess+' tension'
      else
        if and(curr_state, tensionpast) # 0 then
          i_mess = i_mess+' tension_past'
        endif
      endif
      if and(curr_state, jointshearnow) # 0 then
        i_mess = i_mess+' joint_shear'
      else
```

```
        if and(curr_state, jointshearpast) # 0 then
          i_mess = i_mess+' joint_shear_past'
        endif
      endif
      if and(curr_state, jointtensionnow) # 0 then
        i_mess = i_mess+' joint_tension'
      else
        if and(curr_state, jointtensionpast) # 0 then
          i_mess = i_mess+' joint_tension_past'
        endif
      endif
      if and(curr_state, volumenow) # 0 then
        i_mess = i_mess+' volume'
      else
        if and(curr_state, volumepast) # 0 then
          i_mess = i_mess+' volume_past'
        endif
      endif
      ii = out(i_mess)
      iaz = z_next(iaz)
    endloop
    iab = b_next(iab)
  endloop
END
_querystate
set log off
return
```

2.6 *FISH* I/O Routines

The set of *FISH* functions described in this section allow data to be written to, and read from, a file. There are two modes: an "ASCII" mode that allows a *FISH* program to exchange data with other programs, and a "*FISH*" mode that allows data to be passed between *FISH* functions. In *FISH* mode, the data are written in binary, without loss of precision; numbers written out in ASCII form may lose precision when read back into a *FISH* program. In *FISH* mode, the value of the *FISH* variable, not the name of the variable, is written to the file. Only one file may be open at any one time.

close The currently open file is closed; 0 is returned for a successful operation.

open(*filename, wr, mode*)

This function opens a file ***filename***, for writing or reading. The variable ***filename*** can be a quoted string or a *FISH* string variable.

Parameter ***wr*** must be an integer with one of two values:

0 file opened for reading; file must exist

1 file opened for writing; existing file will be overwritten

Parameter ***mode*** must be an integer with one of two values:

0 read/write of *FISH* variables; only the data corresponding to the *FISH* variable (integer, float or string), not the name of the variable, are transferred.

1 read/write of ASCII data; on a read operation, the data are expected to be organized in lines, with CR/LF between lines. A maximum of 80 characters per line is allowed.

The returned value denotes the following conditions.

0 file opened successfully

1 ***filename*** is not a string

2 ***filename*** is a string, but is empty

3 ***wr*** or ***mode*** (not integers)

4 bad ***mode*** (not 0 or 1)

5 bad ***wr*** (not 0 or 1)

6 cannot open file for reading (e.g., file does not exist)

7 file already open

8 not a *FISH* mode file (for read access in *FISH* mode)

read(*ar, n*)

reads ***n*** records into the array ***ar***. Each record is either a line of ASCII data or a single *FISH* variable. The array ***ar*** must be an array of at least ***n*** elements. The returned value is:

0	requested number of lines were input without error
−1	error on read (except end-of-file)
n	positive value indicates that end-of-file was encountered after reading *n* lines

In *FISH* mode, the number and type of records read must exactly match the number and type of records written. It is up to the user to control this. If an arbitrary number of variables are to be written, the first record could be made to contain this number, so that the correct number could subsequently be read.

write(*ar, n*)

writes ***n*** records from the first ***n*** elements of the array ***ar***. Each record is either a line of ASCII data or a single *FISH* variable. For ASCII mode, each element written must be of type *string*. The array ***ar*** must be an array of at least ***n*** elements. The returned value is:

0	requested number of lines were output without error
−1	error on write
n	positive value (in ASCII mode) indicates that the *n*th element was not a string (hence only $n - 1$ lines were written). An error message is also displayed on the screen.

The following intrinsic functions do not perform file operations, but can be used to extract items from ASCII data that are derived from a file.

parse(*s, i*)

This function scans the string ***s*** and decodes the ***i***th item, which it returns. Integers, floats and strings are recognized. Delimiters are the same as for general commands (i.e., spaces, commas, parentheses, tabs and equal signs). If the ***i***th item is missing, zero is returned. An error message is displayed and zero is returned if the variable ***s*** is not a string.

pre_parse(*s, i*)

This function scans the string ***s*** and returns an integer value according to the type of the ***i***th item, as follows.

0 missing item
1 integer
2 float
3 string missing (unable to interpret as *int* or *float*)

Example 2.18 illustrates the use of the *FISH* I/O functions:

Example 2.18 Using the FISH I/O functions

```
def setup
  a_size = 20
  IO_READ  = 0
  IO_WRITE = 1
  IO_FISH  = 0
  IO_ASCII = 1
  filename = 'junk.dat'
end
setup
;
def io
  array aa(a_size) bb(a_size)
;
  ; ASCII I/O TEST ------------------
  status = open(filename, IO_WRITE, IO_ASCII)
  aa(1)  = 'Line 1 ... Fred'
  aa(2)  = 'Line 2 ... Joe'
  aa(3)  = 'Line 3 ... Roger'
  status = write(aa,3)
  status = close
  status = open(filename, IO_READ, IO_ASCII)
  status = read(bb, a_size)
  if status # 3 then
    oo = out(' Bad number of lines')
  endif
  status = close
;
  ; now check results...
  loop n (1,3)
    if parse(bb(n), 2) # n then
      oo = out(' Bad 2nd item in loop ' + string(n))
      exit
```

```
    endif
  endloop
;
  if pre_parse(bb(3), 4) # 3 then
    oo = out(' Not a string')
    exit
  endif
;
  ; FISH I/O TEST -----------------
  status = open(filename, IO_WRITE, IO_FISH)
  funny_int   = 1234567
  funny_float = 1.2345e6
  aa(1)  = '---> All tests passed OK'
  aa(2)  = funny_int
  aa(3)  = funny_float
;
  status = write(aa,3)
  status = close
  status = open(filename, IO_READ, IO_FISH)
  status = read(bb, 3)
  status = close
;
  ; now check results...
  if type(bb(1)) # 3 then
    oo = out(' Bad FISH string read/write')
      exit
    endif
  if bb(2) # funny_int then
    oo = out(' Bad FISH integer read/write')
    exit
  endif
  if bb(3) # funny_float then
    oo = out(' Bad FISH float read/write')
    exit
  endif
  oo = out(bb(1)) ; (should be a good message)
  command
    sys del junk.dat
  endcommand
end
;
io
```

2.7 Socket I/O Routines

FISH contains the option to allow data to be exchanged between two or more Itasca codes running as separate processes, using *socket connections* (as used for TCP/IP transmission over the Internet). At present, these versions (or later) of Itasca codes are required for socket I/O: *UDEC* Version 3.0, *FLAC* Version 4.0, *PFC2D* Version 2.1, *PFC3D* Version 2.1 and *FLAC3D* Version 2.1. It is possible to pass data between two or more instances of the same code (e.g., two instances of *UDEC*), but the main use is anticipated to be coupling of dissimilar codes such as *UDEC* and *PFC2D*.

The data contained in *FISH* arrays may be passed in either direction between two codes. The data are transmitted in binary with no loss of precision. Up to six data channels may be open at any one time; these may exist between two codes, or may connect several codes simultaneously. The following *FISH* intrinsics are provided. The word *process* denotes the instance of the code that is currently running. All functions return a value of 10 if the ID number is invalid.

sopen(*mode, ID*)

The integer ***mode*** takes the value 0 or 1. A value of 1 causes the data channel of number ***ID*** to be initiated, with the process acting as a *server*. Another process can link to the server, with the same ***ID***, by invoking **sopen**, with ***mode*** = 0, which denotes the process as a *client*. The ***ID*** number must be in the range 0 to 5, inclusive, giving a total of six possible channels of communication. The server **sopen** function must be issued before the client **sopen** function, for a given ***ID***. While waiting for a connection, the server process is unresponsive. The **sopen** function returns 0 when a good connection has been made, and nonzero if an error has been detected.

sclose(*ID*)

Channel ***ID*** is closed.

swrite(*arr, num, ID*)

num *FISH* variables are sent on channel ***ID*** from array ***arr***. The data in ***arr*** may consist of a mixture of integers, reals and strings. Zero is returned for a good data transmission; nonzero is returned if an error is detected. In addition, error messages may be issued for various problems, such as incorrect array size, etc.

sread(*arr, num, ID*)

num *FISH* variables are received from channel ***ID*** and placed in array ***arr***, which is overwritten, and which must be at least ***num*** elements in size. The returned value is zero if data are received without error; it is nonzero if an error has occurred. Note that the function **sread** does not return until the requested number of items have been received. Therefore, a process will appear to "lock up" if insufficient data have been sent by the sending process.

In order to achieve socket communication between two processes, codes must be started separately from separate directories. To illustrate the procedure, we can send messages between two instances of *UDEC*, as follows.

Example 2.19 Server data file

```
new
def serve
  array arr(3)
  arr(1) = 1234
  arr(2) = 57.89
  arr(3) = 'hello from the server'
  _ret = sopen(1,1)
  if _ret = 0 then
    oo = swrite(arr,3,1)
    oo = sread(arr,1,1)
    oo = sclose(1)
    oo = out(arr(1))
  endif
end
serve
```

The client data file is as follows.

Example 2.20 Client data file

```
new
def client
  array arr(3)
  oo = sopen(0,1)
  oo = sread(arr,3,1)
  oo = out(' Received values ... ')
  oo = out('   '+string(arr(1)))
  oo = out('   '+string(arr(2)))
```

```
  oo = out('   '+string(arr(3)))
  arr(1) = 'greetings from the client'
  oo = swrite(arr,1,1)
  oo = sclose(1)
end
;   Received values should be...
;     1234
;     5.7890E+01
;     hello from the server
client
```

Data have been passed both ways between the two code instances.