Fortran 90 Overview

J.E. Akin,

This overview of Fortran 90 (F90) features is presented as a series of tables that illustrate the syntax and abilities of F90. Frequently comparisons are made to similar features in the C++ and F77 languages and to the Matlab environment.

These tables show that F90 has significant improvements over F77 and matches or exceeds newer software capabilities found in C++ and Matlab for dynamic memory management, user defined data structures, matrix operations, operator definition and overloading, intrinsics for vector and parallel processors and the basic requirements for object-oriented programming.

They are intended to serve as a condensed quick reference guide for programming in F90 and for understanding programs developed by others.

List of Tables

Language	Syntax	Location
MATLAB	`% comment (to end of line)`	anywhere
C	`/*comment*/`	anywhere
F90	`! comment (to end of line)`	anywhere
F77	`* comment (to end of line)`	column 1

Table 1: Comment syntax.

Storage	MATLAB[a]	C++	F90	F77
byte		`char`	`character::`	`character`
integer		`int`	`integer::`	`integer`
single precision		`float`	`real::`	`real`
double precision		`double`	`real*8::`	`double precision`
complex		[b]	`complex::`	`complex`
Boolean		`bool`	`logical::`	`logical`
argument			`parameter::`	`parameter`
pointer		`*`	`pointer::`	
structure		`struct`	`type::`	

[a] MATLAB4 requires no variable type declaration; the only two distinct types in MATLAB are strings and reals (which include complex). Booleans are just 0s and 1s treated as reals. MATLAB5 allows the user to select more types.

[b] There is no specific data type for a complex variable in C++; they must be created by the programmer.

Table 2: Intrinsic data types of variables.

Description	MATLAB[a]	C++	Fortran[b]
addition	`+`	`+`	`+`
subtraction[c]	`-`	`-`	`-`
multiplication	`*` and `.*`	`*`	`*`
division	`/` and `./`	`/`	`/`
exponentiation	`^` and `.^`	pow[d]	`**`
remainder		`%`	
increment		`++`	
decrement		`--`	
parentheses (expression grouping)	`()`	`()`	`()`

[a] When doing arithmetic operations on matrices in MATLAB, a period (‘`.`’) must be put before the operator if scalar arithmetic is desired. Otherwise, MATLAB assumes matrix operations; figure out the difference between ‘`*`’ and ‘`.*`’. Note that since matrix and scalar addition coincide, no ‘`.+`’ operator exists (same holds for subtraction).

[b] Fortran 90 allows the user to change operators and to define new operator symbols.

[c] In all languages the minus sign is used for negation (i.e., changing sign).

[d] In C++ the exponentiation is calculated by function *pow* (x, y).

Table 3: Arithmetic operators.

Description	MATLAB	C++	F90	F77
Equal to	==	==	==	.EQ.
Not equal to	~=	!=	/=	.NE.
Less than	<	<	<	.LT.
Less or equal	<=	<=	<=	.LE.
Greater than	>	>	>	.GT.
Greater or equal	>=	>=	>=	.GE.
Logical NOT	~	!	.NOT.	.NOT.
Logical AND	&	&&	.AND.	.AND.
Logical inclusive OR	!	||	.OR.	.OR.
Logical exclusive OR	xor		.XOR.	.XOR.
Logical equivalent	==	==	.EQV.	.EQV.
Logical not equivalent	~=	!=	.NEQV.	.NEQV.

Table 4: Relational operators (arithmetic and logical).

MATLAB Operators	C++ Operators	F90 Operators[a]	F77 Operators
()	() [] -> .	()	()
+ -	! ++ -- + - * & (type) sizeof	**	**
* /	* / %	* /	* /
+ -[b]	+ -[b]	+ -[b]	+ -[b]
< <= > >=	<< >>	//	//
== ~=	< <= > =>	== /= < <= > >=	.EQ. .NE. .LT. .LE. .GT. .GE.
~	== !=	.NOT.	.NOT.
&	&&	.AND.	.AND.
|	||	.OR.	.OR.
=	|	.EQV. .NEQV.	.EQV. .NEQV.
	?:		
	= += -= *= /= %= &= ^= |= <<= >>=		
	,		

[a] User-defined unary (binary) operators have the highest (lowest) precedence in F90.
[b] These are binary operators representing addition and subtraction. Unary operators + and - have higher precedence.

Table 5: Precedence pecking order.

B = Beginning, E = Ending, I = Increment

Syntax	F90	MATLAB
Default	B:E:I	B:I:E
$\geq$ B	B:	B:
$\leq$ E	:E	:E
Full range	:	:

Use	F90	MATLAB
Array subscript ranges	yes	yes
Character positions in a string	yes	yes
Loop control	no	yes
Array element generation	no	yes

Table 6: Colon Operator Syntax and its Applications.

Description	MATLAB	C++	F90	F77
exponential	`exp(x)`	`exp(x)`	`exp(x)`	`exp(x)`
natural log	`log(x)`	`log(x)`	`log(x)`	`log(x)`
base 10 log	`log10(x)`	`log10(x)`	`log10(x)`	`log10(x)`
square root	`sqrt(x)`	`sqrt(x)`	`sqrt(x)`	`sqrt(x)`
raise to power (x^r)	`x.^r`	`pow(x,r)`	`x**r`	`x**r`
absolute value	`abs(x)`	`fabs(x)`	`abs(x)`	`abs(x)`
smallest integer>x	`ceil(x)`	`ceil(x)`	`ceiling(x)`	
largest integer<x	`floor(x)`	`floor(x)`	`floor(x)`	
division remainder	`rem(x,y)`	`fmod(x,y)`	`mod(x,y)`[a]	`mod(x,y)`
modulo			`modulo(x,y)`[a]	
complex conjugate	`conj(z)`		`conjg(z)`	`conjg(z)`
imaginary part	`imag(z)`		`imag(z)`	`aimag(z)`
drop fraction	`fix(x)`		`aint(x)`	`aint(x)`
round number	`round(x)`		`nint(x)`	`nint(x)`
cosine	`cos(x)`	`cos(x)`	`cos(x)`	`cos(x)`
sine	`sin(x)`	`sin(x)`	`sin(x)`	`sin(x)`
tangent	`tan(x)`	`tan(x)`	`tan(x)`	`tan(x)`
arc cosine	`acos(x)`	`acos(x)`	`acos(x)`	`acos(x)`
arc sine	`asin(x)`	`asin(x)`	`asin(x)`	`asin(x)`
arc tangent	`atan(x)`	`atan(x)`	`atan(x)`	`atan(x)`
arc tangent[b]	`atan2(x,y)`	`atan2(x,y)`	`atan2(x,y)`	`atan2(x,y)`
hyperbolic cosine	`cosh(x)`	`cosh(x)`	`cosh(x)`	`cosh(x)`
hyperbolic sine	`sinh(x)`	`sinh(x)`	`sinh(x)`	`sinh(x)`
hyperbolic tangent	`tanh(x)`	`tanh(x)`	`tanh(x)`	`tanh(x)`
hyperbolic arc cosine	`acosh(x)`			
hyperbolic arc sine	`asinh(x)`			
hyperbolic arctan	`atanh(x)`			

[a] Differ for $x < 0$.

[b] `atan2(x,y)` is used to calculate the arc tangent of x/y in the range $[-\pi, +\pi]$. The one-argument function `atan(x)` computes the arc tangent of x in the range $[-\pi/2, +\pi/2]$.

Table 7: Mathematical functions.

Description	**C++**	**F90**	**F77**	**MATLAB**
Conditionally execute statements	`if` `{ }`	`if` `end if`	`if` `end if`	`if` `end`
Loop a specific number of times	`for k=1:n` `{ }`	`do k=1,n` `end do`	`do # k=1,n` `# continue`	`for k=1:n` `end`
Loop an indefinite number of times	`while` `{ }`	`do while` `end do`	— —	`while` `end`
Terminate and exit loop	`break`	`exit`	`go to`	`break`
Skip a cycle of loop	`continue`	`cycle`	`go to`	—
Display message and abort	`error()`	`stop`	`stop`	`error`
Return to invoking function	`return`	`return`	`return`	`return`
Conditional array action	—	`where`	—	`if`
Conditional alternate statements	`else` `else if`	`else` `elseif`	`else` `elseif`	`else` `elseif`
Conditional array alternatives	— —	`elsewhere` —	— —	`else` `elseif`
Conditional case selections	`switch { }`	`select case` `end select`	`if` `end if`	`if` `end`

Table 8: Flow Control Statements.

Loop	**MATLAB**	**C++**	**Fortran**

Indexed loop	`for index=matrix`
	`  statements`
	`  end`
		`for (init;test;inc)`
		`  {`
		`  statements`
		`  }`
			`do index=b,e,i`
			`  statements`
			`  end do`

Pre-test loop	`while test`
	`  statements`
	`  end`
		`while (test) {`
		`  statements`
		`  }`
			`do while (test)`
			`  statements`
			`  end do`

Post-test loop		`do {`
		`  statements`
		`  } while (test)`
			`do`
			`  statements`
			`  if (test) exit`
			`  end do`

Table 9: Basic loop constructs.

MATLAB	Fortran	C++

`if l_expression`
`  true group`
`end`
	`IF (l_expression) THEN`
	`  true group`
	`END IF`
		`if (l_expression)`
		`  {`
		`  true group;`
		`  }`

	`IF (l_expression) true statement`	`if (l_expression)`
		`  true state-`
		`ment;`

Table 10: IF Constructs. The quantity `l_expression` means a logical expression having a value that is either TRUE of FALSE. The term *`true statement`* or *`true group`* means that the statement or group of statements, respectively, are executed if the conditional in the `if` statement evaluates to TRUE.

MATLAB	Fortran	C++

`if l_expression1`
`  true group A`
`  if l_expression2`
`  true group B`
`  end`
`  true group C`
`end`
`statement group D`
	`IF (l_expression1) THEN`
	`  true group A`
	`  IF (l_expression2) THEN`
	`  true group B`
	`  END IF`
	`  true group C`
	`END IF`
	`statement group D`
		`if (l_expression1)`
		`  {`
		`  true group A`
		`  if (l_expression2)`
		`  {`
		`  true group B`
		`  }`
		`  true group C`
		`  }`
		`statement group D`

Table 11: Nested IF Constructs.

MATLAB	Fortran	C++

`if l_expression`
`  true group A`
`else`
`  false group B`
`end`
	`IF (l_expression) THEN`
	`  true group A`
	`ELSE`
	`  false group B`
	`END IF`
		`if (l_expression)`
		`  {`
		`  true group A`
		`  }`
		`else`
		`  {`
		`  false group B`
		`  }`

Table 12: Logical IF-ELSE Constructs.

MATLAB	Fortran	C++

`if l_expression1`
`  true group A`
`elseif l_expression2`
`  true group B`
`elseif l_expression3`
`  true group C`
`else`
`  default group D`
`end`
	`IF (l_expression1) THEN`
	`  true group A`
	`ELSE IF (l_expression2) THEN`
	`  true group B`
	`ELSE IF (l_expression3) THEN`
	`  true group C`
	`ELSE`
	`  default group D`
	`END IF`
		`if (l_expression1)`
		`  {`
		`  true group A`
		`  }`
		`else if (l_expression2)`
		`  {`
		`  true group B`
		`  }`
		`else if (l_expression3)`
		`  {`
		`  true group C`
		`  }`
		`else`
		`  {`
		`  default group D`
		`  }`

Table 13: Logical IF-ELSE-IF Constructs.

F90	C++

SELECT CASE (expression)
  CASE (value 1)
  group 1
  CASE (value 2)
  group 2
  ⋮
  CASE (value n)
  group n
  CASE DEFAULT
  default group
END SELECT
	switch (expression)
	  {
	  case value 1 :
	  group 1
	  break;
	  case value 2 :
	  group 2
	  break;
	  ⋮
	  case value n :
	  group n
	  break;
	  default:
	  default group
	  break;
	  }

Table 14: Case Selection Constructs.

F90 Named `IF`	F90Named `SELECT`

name: IF (logical_1) THEN
  true group A
ELSE IF (logical_2) THEN
  true group B
ELSE
  default group C
ENDIF name
	name: SELECT CASE (expression)
	  CASE (value 1)
	  group 1
	  CASE (value 2)
	  group 2
	  CASE DEFAULT
	  default group
	  END SELECT name

Table 15: F90 Optional Logic Block Names.

Fortran	C++

DO 1 ...
  DO 2 ...
  ...
  IF (disaster) THEN
  GO TO 3
  END IF
  ...
2 END DO
1 END DO
3 *next statement*
	for (...) {
	  for (...) {
	  ...
	  if (disaster)
	  go to error
	  ...
	  }
	}
	error:

Table 16: `GO TO` Break-out of Nested Loops. This situation can be an exception to the general recommendation to avoid `GO TO` statements.

F77	F90	C++

DO 1 I = 1,N
...
IF (skip condi-
tion) THEN
GO TO 1
ELSE
false group
END IF
1 continue
	DO I = 1,N
	...
	IF (skip condi-
	tion) THEN
	CYCLE ! to next I
	ELSE
	false group
	END IF
	END DO
		for (i=1; i<n; i++)
		{
		if (skip condition)
		continue; // to next
		else if
		false group
		end
		}

Table 17: Skip a Single Loop Cycle.

F77	F90	C++

DO 1 I = 1,N
IF (exit condi-
tion) THEN
GO TO 2
ELSE
false group
END IF
1 CONTINUE
2 *next statement*
	DO I = 1,N
	IF (exit condi-
	tion) THEN
	EXIT ! this do
	ELSE
	false group
	END IF
	END DO
	next statement
		for (i=1; i<n; i++)
		{
		if (exit condition)
		break;// out of loop
		else if
		false group
		end
		}
		next statement

Table 18: Abort a Single Loop.

```
main: DO ! forever
 test: DO k=1,k_max
  third: DO m=m_max,m_min,-1
   IF (test condition) THEN
    CYCLE test ! loop on k
   END IF
  END DO third ! loop on m
  fourth: DO n=n_min,n_max,2
   IF (main condition) THEN
    EXIT main ! forever loop
  END DO fourth ! on n
 END DO test ! over k
END DO main
next statement
```

Table 19: F90 DOs Named for Control.

MATLAB	C++

`initialize test`
`while l_expression`
`  true group`
`  change test`
`end`
	`initialize test`
	`while (l_expression)`
	`  {`
	`  true group`
	`  change test`
	`  }`

F77	F90

`  initialize test`
`# continue`
`  IF (l_expression) THEN`
`   true group`
`   change test`
`   go to #`
`  END IF`
	`initialize test`
	`do while (l_expression)`
	`  true group`
	`  change test`
	`end do`

Table 20: Looping While a Condition is True.

Function Type	MATLAB[a]	C++	Fortran

program	*`statements`*
	`[y1...yn]=f(a1,...,am)`
	`[end of file]`
		`main(argc,char **argv)`
		`  {`
		`  statements`
		`  y = f(a1,I,am);`
		`  }`
			`program main`
			`type y`
			`type a1,...,type am`
			`statements`
			`y = f(a1,...,am)`
			`call s(a1,...,am)`
			`end program`

subroutine		`void  f`
		`(type a1,...,type am)`
		`  {`
		`  statements`
		`  }`
			`subroutine s(a1,...,am)`
			`type a1,...,type am`
			`  statements`
			`  end`

function	`function [r1...rn]`
	`  =f(a1,...,am)`
	`statements`
		type f (type a1,...,type am)
		{ *statements* }
			`function f(a1,...,am)`
			`type f`
			`type a1,...,type am`
			`  statements`
			`  end`

[a] Every function or program in MATLAB must be in separate files.

Table 21: Function definitions. In each case, the function being defined is named `f` and is called with `m` arguments `a1,...,am`.

One-Input, One-Result Procedures	
MATLAB	`function out = name (in)`
F90	`function name (in) ! name = out`
	`function name (in) result (out)`
C++	`name (in, out)`

Multiple-Input, Multiple-Result Procedures	
MATLAB	`function [inout, out2] = name (in1, in2, inout)`
F90	`subroutine name (in1, in2, inout, out2)`
C++	`name(in1, in2, inout, out2)`

Table 22: Arguments and return values of subprograms.

Global Variable Declaration	
MATLAB	`global` list of variables
F77	`common` /set_name/ list of variables
F90	`module` set_name `save` `type` (type_tag) :: list of variables `end module` set_name
C++	`extern` list of variables

Access to Global Variables	
MATLAB	`global` list of variables
F77	`common` /set_name/ list of variables
F90	`use` set_name, `only` subset of variables `use` set_name2 list of variables
C++	`extern` list of variables

Table 23: Defining and referring to global variables.

Action	C++	F90
Bitwise AND	&	`iand`
Bitwise exclusive OR	∧	`ieor`
Bitwise exclusive OR	|	`ior`
Circular bit shift		`ishftc`
Clear bit		`ibclr`
Combination of bits		`mvbits`
Extract bit		`ibits`
Logical complement	∼	`not`
Number of bits in integer	`sizeof`	`bit_size`
Set bit		`ibset`
Shift bit left	≪	`ishft`
Shift bit right	≫	`ishft`
Test on or off		`btest`
Transfer bits to integer		`transfer`

Table 24: Bit Function Intrinsics.

0	NUL	1	SOH	2	STX	3	ETX	4	EOT	5	ENQ	6	ACK	7	BEL
8	BS	9	HT	10	NL	11	VT	12	NP	13	CR	14	SO	15	SI
16	DLE	17	DC1	18	DC2	19	DC3	20	DC4	21	NAK	22	SYN	23	ETB
24	CAN	25	EM	26	SUB	27	ESC	28	FS	29	GS	30	RS	31	US
32	SP	33	!	34	"	35	#	36	$	37	%	38	&	39	’
40	(	41	)	42	*	43	+	44	,	45	-	46	.	47	/
48	0	49	1	50	2	51	3	52	4	53	5	54	6	55	7
56	8	57	9	58	:	59	;	60	<	61	=	62	>	63	?
64	@	65	A	66	B	67	C	68	D	69	E	70	F	71	G
72	H	73	I	74	J	75	K	76	L	77	M	78	N	79	O
80	P	81	Q	82	R	83	S	84	T	85	U	86	V	87	W
88	X	89	Y	90	Z	91	[	92	\	93	]	94	^	95	_
96	‘	97	a	98	b	99	c	100	d	101	e	102	f	103	g
104	h	105	i	106	j	107	k	108	l	109	m	110	n	111	o
112	p	113	q	114	r	115	s	116	t	117	u	118	v	119	w
120	x	121	y	122	z	123	{	124	|	125	}	126	~	127	DEL

Table 25: The ACSII Character Set.

`ACHAR (I)`	Character number I in ASCII collating set
`ADJUSTL (STRING)`	Adjust left
`ADJUSTR (STRING)`	Adjust right
`CHAR (I) *`	Character I in processor collating set
`IACHAR (C)`	Position of C in ASCII collating set
`ICHAR (C)`	Position of C in processor collating set
`INDEX (STRING, SUBSTRING)`[a]	Starting position of a substring
`LEN (STRING)`	Length of a character entity
`LEN_TRIM (STRING)`	Length without trailing blanks
`LGE (STRING_A, STRING_B)`	Lexically greater than or equal
`LGT (STRING_A, STRING_B)`	Lexically greater than
`LLE (STRING_A, STRING_B)`	Lexically less than or equal
`LLT (STRING_A, STRING_B)`	Lexically less than
`REPEAT (STRING, NCOPIES)`	Repeated concatenation
`SCAN (STRING, SET)`[a]	Scan a string for a character in a set
`TRIM (STRING)`	Remove trailing blank characters
`VERIFY (STRING, SET)`[a]	Verify the set of characters in a string
`STRING_A//STRING_B`	Concatenate two strings

[a] Optional arguments not shown.

Table 26: F90 Character Functions.

Action	*ASCII Character*	F90 *Input*[a]	*C++ Input*
Alert (Bell)	7	Ctrl-G	\a
Backspace	8	Ctrl-H	\b
Carriage Return	13	Ctrl-M	\r
End of Transmission	4	Ctrl-D	Ctrl-D
Form Feed	12	Ctrl-L	\f
Horizontal Tab	9	Ctrl-I	\t
New Line	10	Ctrl-J	\n
Vertical Tab	11	Ctrl-K	\v

[a] “Ctrl-” denotes control action. That is, simultaneous pressing of the `CONTROL` key *and* the letter following.

Table 27: How to type non-printing characters.

C, C++	`Variable.component.sub_component`
F90	`Variable%component%sub_component`

Table 28: Referencing Structure Components.

C, C++	`struct data_tag {`
	`  intrinsic_type_1 component_names;`
	`  intrinsic_type_2 component_names;`
	`} ;`

F90	`type data_tag`
	`  intrinsic_type_1 :: component_names;`
	`  intrinsic_type_2 :: component_names;`
	`end type  data_tag`

Table 29: Defining New Types of Data Structure.

C, C++	`struct data_tag {`
	`  intrinsic_type_1 component_names;`
	`  struct tag_2  component_names;`
	`} ;`

F90	`type data_tag`
	`  intrinsic_type :: component_names;`
	`  type (tag_2) :: component_names;`
	`end type data_tag`

Table 30: Nested Data Structure Definitions.

C, C++	`struct data_tag variable_list; /* Definition */`
	`struct data_tag variable = {component_values}; /* Initialization */`
	`variable.component.sub_component = value; /* Assignment */`

F90	`type (data_tag) :: variable_list ! Definition`
	`variable = data_tag (component_values) ! Initialization`
	`variable%component%sub_component = value ! Assignment`

Table 31: Declaring, initializing, and assigning components of user-defined datatypes.

```
INTEGER, PARAMETER ::  j_max = 6
TYPE meaning_demo
    INTEGER, PARAMETER ::  k_max = 9, word = 15
    CHARACTER (LEN = word) ::  name(k_max)
END TYPE meaning_demo
TYPE (meaning_demo) derived(j_max)
```

Construct	Interpretation
`derived`	All components of all `derived`'s elements
`derived(j)`	All components of j^{th} element of `derived`
`derived(j)%name`	All `k_max` components of `name` within j^{th} element of `derived`
`derived%name(k)`	Component `k` of the `name` array for all elements of `derived`
`derived(j)%name(k)`	Component `k` of the `name` array of j^{th} element of `derived`

Table 32: F90 Derived Type Component Interpretation.

	C++	F90
Declaration	`type_tag *pointer_name;`	`type (type_tag), pointer :: pointer_name`
Target	`&target_name`	`type (type_tag), target ::  target_name`

Examples	`char *cp, c;`
	`int *ip, i;`
	`float *fp, f;`
	`cp = & c;`
	`ip = & i;`
	`fp = & f;`
		`character, pointer :: cp`
		`integer, pointer :: ip`
		`real, pointer :: fp`
		`cp => c`
		`ip => i`
		`fp => f`

Table 33: Definition of pointers and accessing their targets.

C, C++	`pointer_name = NULL`
F90	`nullify (list_of_pointer_names)`
F95	`pointer_name = NULL()`

Table 34: Nullifing a Pointer to Break Association with Target.

Purpose	F90	MATLAB
Form subscripts	( )	( )
Separates subscripts & elements	,	,
Generates elements & subscripts	:	:
Separate commands	;	;
Forms arrays	(/ /)	[ ]
Continue to new line	&	...
Indicate comment	!	%
Suppress printing	default	;

Table 35: Special Array Characters.

Description	*Equation*	*Fortran90 Operator*	*Matlab Operator*	*Original Sizes*	*Result Size*
Scalar plus scalar	$c = a \pm b$	$c = a \pm b$	$c = a \pm b$;	$1, 1$	$1, 1$
Element plus scalar	$c_{jk} = a_{jk} \pm b$	$c = a \pm b$	$c = a \pm b$;	m, n and $1, 1$	m, n
Element plus element	$c_{jk} = a_{jk} \pm b_{jk}$	$c = a \pm b$	$c = a \pm b$;	m, n and m, n	m, n
Scalar times scalar	$c = a \times b$	$c = a * b$	$c = a * b$;	$1, 1$	$1, 1$
Element times scalar	$c_{jk} = a_{jk} \times b$	$c = a * b$	$c = a * b$;	m, n and $1, 1$	m, n
Element times element	$c_{jk} = a_{jk} \times b_{jk}$	$c = a * b$	$c = a. * b$;	m, n and m, n	m, n
Scalar divide scalar	$c = a/b$	$c = a/b$	$c = a/b$;	$1, 1$	$1, 1$
Scalar divide element	$c_{jk} = a_{jk}/b$	$c = a/b$	$c = a/b$;	m, n and $1, 1$	m, n
Element divide element	$c_{jk} = a_{jk}/b_{jk}$	$c = a/b$	$c = a./b$;	m, n and m, n	m, n
Scalar power scalar	$c = a^b$	$c = a**b$	$c = a \wedge b$;	$1, 1$	$1, 1$
Element power scalar	$c_{jk} = a_{jk}^b$	$c = a**b$	$c = a \wedge b$;	m, n and $1, 1$	m, n
Element power element	$c_{jk} = a_{jk}^{b_{jk}}$	$c = a**b$	$c = a. \wedge b$;	m, n and m, n	m, n
Matrix transpose	$C_{kj} = A_{jk}$	$C = \text{transpose}\,(A)$	$C = A'$;	m, n	n, m
Matrix times matrix	$C_{ij} = \sum_k A_{ik} B_{kj}$	$C = \text{matmul}(A, B)$	$C = A * B$;	m, r and r, n	m, n
Vector dot vector	$c = \sum_k A_k B_k$	$c = \text{sum}(A * B)$	$c = \text{sum}(A. * B)$;	$m, 1$ and $m, 1$	$1, 1$
		$c = \text{dot\_product}(A, B)$	$c = A * B'$;	$m, 1$ and $m, 1$	$1, 1$

Table 36: Array Operations in Programming Constructs. Lower case letters denote scalars or scalar elements of arrays. Matlab arrays are allowed a maximum of two subscripts while Fortran allows seven. Upper case letters denote matrices or scalar elements of matrices.

Table 37: Equivalent Fortran90 and MATLAB Intrinsic Functions.
The following KEY symbols are utilized to denote the TYPE of the intrinsic function, or subroutine, and its arguments: A-complex, integer, or real; I-integer; L-logical; M-mask (logical); R-real; X-real; Y-real; V-vector (rank 1 array); and Z-complex. Optional arguments are not shown. Fortran90 and MATLAB also have very similar array operations and colon operators.

Type	**Fortran90**	**MATLAB**	**Brief Description**
A	ABS(A)	abs(a)	Absolute value of A.
R	ACOS(X)	acos(x)	Arc cosine function of real X.
R	AIMAG(Z)	imag(z)	Imaginary part of complex number.
R	AINT(X)	real(fix(x))	Truncate X to a real whole number.
L	ALL(M)	all(m)	True if all mask elements, M, are true.
R	ANINT(X)	real(round(x))	Real whole number nearest to X.
L	ANY(M)	any(m)	True if any mask element, M, is true.
R	ASIN(X)	asin(x)	Arcsine function of real X.
R	ATAN(X)	atan(x)	Arctangent function of real X.
R	ATAN2(Y,X)	atan2(y,x)	Arctangent for complex number(X, Y).
I	CEILING(X)	ceil(x)	Least integer >= real X.
Z	CMPLX(X,Y)	(x+yi)	Convert real(s) to complex type.
Z	CONJG(Z)	conj(z)	Conjugate of complex number Z.
R	COS(R_Z)	cos(r_z)	Cosine of real or complex argument.
R	COSH(X)	cosh(x)	Hyperbolic cosine function of real X.
I	COUNT(M)	sum(m==1)	Number of true mask, M, elements.
R,L	DOT_PRODUCT(X,Y)	x'⋆y	Dot product of vectors X and Y.
R	EPSILON(X)	eps	Number, like X, $\ll 1$.
R,Z	EXP(R_Z)	exp(r_z)	Exponential of real or complex number.
I	FLOOR(X)	floor	Greatest integer $\leq$ X.
R	HUGE(X)	realmax	Largest number like X.
I	INT(A)	fix(a)	Convert A to integer type.
R	LOG(R_Z)	log(r_z)	Logarithm of real or complex number.
R	LOG10(X)	log10(x)	Base 10 logarithm function of real X.
R	MATMUL(X,Y)	x⋆y	Conformable matrix multiplication, X*Y.
I,V	I=MAXLOC(X)	[y,i]=max(x)	Location(s) of maximum array element.
R	Y=MAXVAL(X)	y=max(x)	Value of maximum array element.
I,V	I=MINLOC(X)	[y,i]=min(x)	Location(s) of minimum array element.
R	Y=MINVAL(X)	y=min(x)	Value of minimum array element.
I	NINT(X)	round(x)	Integer nearest to real X.
A	PRODUCT(A)	prod(a)	Product of array elements.
call	RANDOM_NUMBER(X)	x=rand	Pseudo-random numbers in $(0, 1)$.
call	RANDOM_SEED	rand('seed')	Initialize random number generator.
R	REAL (A)	real(a)	Convert A to real type.
R	RESHAPE(X, (/ I, I2 /))	reshape(x, i, i2)	Reshape array X into I$\times$I2 array.
I,V	SHAPE(X)	size(x)	Array (or scalar) shape vector.
R	SIGN(X,Y)		Absolute value of X times sign of Y.
R	SIGN(0.5,X)-SIGN(0.5,-X)	sign(x)	Signum, normalized sign, –1, 0, or 1.
R,Z	SIN(R_Z)	sin(r_z)	Sine of real or complex number.
R	SINH(X)	sinh(x)	Hyperbolic sine function of real X.
I	SIZE(X)	length(x)	Total number of elements in array X.
R,Z	SQRT(R_Z)	sqrt(r_z)	Square root, of real or complex number.
R	SUM(X)	sum(x)	Sum of array elements.

(continued)

Type	Fortran90	MATLAB	Brief Description
R	TAN(X)	tan(x)	Tangent function of real X.
R	TANH(X)	tanh(x)	Hyperbolic tangent function of real X.
R	TINY(X)	realmin	Smallest positive number like X.
R	TRANSPOSE(X)	x'	Matrix transpose of any type matrix.
R	X=1	x=ones(length(x))	Set all elements to 1.
R	X=0	x=zero(length(x))	Set all elements to 0.

For more detailed descriptions and example uses of these intrinsic functions see Adams, J.C., *et al.*, *Fortran 90 Handbook*, McGraw-Hill, New York, 1992, ISBN 0–07–000406–4.

C++	–	`int`	–	–	`floor`	`ceil`
F90	`aint`	`int`	`anint`	`nint`	`floor`	`ceiling`
MATLAB	`real`(fix)	`fix`	`real`(round)	`round`	`floor`	`ceil`
Argument	**Value of Result**					
–2.000	–2.0	–2	–2.0	–2	–2	–2
–1.999	–1.0	–1	–2.0	–2	–2	–1
–1.500	–1.0	–1	–2.0	–2	–2	–1
–1.499	–1.0	–1	–1.0	–1	–2	–1
–1.000	–1.0	–1	–1.0	–1	–1	–1
–0.999	0.0	0	–1.0	–1	–1	0
–0.500	0.0	0	–1.0	–1	–1	0
–0.499	0.0	0	0.0	0	1	0
0.000	0.0	0	0.0	0	0	0
0.499	0.0	0	0.0	0	0	1
0.500	0.0	0	1.0	1	0	1
0.999	0.0	0	1.0	1	0	1
1.000	1.0	1	1.0	1	1	1
1.499	1.0	1	1.0	1	1	2
1.500	1.0	1	2.0	2	1	2
1.999	1.0	1	2.0	2	1	2
2.000	2.0	2	2.0	2	2	2

Table 38: Truncating Numbers.

```
WHERE (logical_array_expression)
   true_array_assignments
ELSEWHERE
   false_array_assignments
END WHERE
```

```
WHERE (logical_array_expression) true_array_assignment
```

Table 39: F90 `WHERE` Constructs.

Function	*Description*	*Opt*	*Example*
`all`	Find if all values are true, for a fixed dimension.	d	`all(B = A, DIM = 1)` (true, false, false)
`any`	Find if any value is true, for a fixed dimension.	d	`any (B > 2, DIM = 1)` (false, true, true)
`count`	Count number of true elements for a fixed dimension.	d	`count(A = B, DIM = 2)` (1, 2)
`maxloc`	Locate first element with maximum value given by mask.	m	`maxloc(A, A < 9)` (2, 3)
`maxval`	Max element, for fixed dimension, given by mask.	b	`maxval (B, DIM=1, B > 0)` (2, 4, 6)
`merge`	Pick true array, A, or false array, B, according to mask, L.	–	`merge(A, B, L)` $\begin{bmatrix} 0 & 3 & 5 \\ 2 & 4 & 8 \end{bmatrix}$
`minloc`	Locate first element with minimum value given by mask.	m	`minloc(A, A > 3)` (2, 2)
`minval`	Min element, for fixed dimension, given by mask.	b	`minval(B, DIM = 2)` (1, 2)
`pack`	Pack array, A, into a vector under control of mask.	v	`pack(A, B < 4)` (0, 7, 3)
`product`	Product of all elements, for fixed dimension, controlled by mask.	b	`product(B)` ; (720) `product(B, DIM = 1, T)` (2, 12, 30)
`sum`	Sum all elements, for fixed dimension, controlled by mask.	b	`sum(B)` ;(21) `sum(B, DIM = 2, T)` (9, 12)
`unpack`	Replace the true locations in array B controlled by mask L with elements from the vector U.	–	`unpack(U, L, B)` $\begin{bmatrix} 7 & 3 & 8 \\ 2 & 4 & 9 \end{bmatrix}$

$$A = \begin{bmatrix} 0 & 3 & 5 \\ 7 & 4 & 8 \end{bmatrix}, \quad B = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}, \quad L = \begin{bmatrix} T & F & T \\ F & F & T \end{bmatrix}, \quad U = (7, 8, 9)$$

Table 40: F90 Array Operators with Logic Mask Control. T and F denote true and false, respectively. Optional arguments: `b -- DIM & MASK`, `d -- DIM`, `m -- MASK`, `v -- VECTOR` and `DIM = 1` implies for any rows, `DIM = 2` for any columns, and `DIM = 3` for any plane.

	MATLAB	C++	F90
Pre-allocate linear array	`A(100)=0`	`int A[100];`[a]	`integer A(100)`

Initialize to a constant value of 12	`for j=1:100 % slow`
	`  A(j)=12`
	`  end`
	`% better way`
	`A=12*ones(1,100)`
		`for (j=0; j<100; j++)`
		`  A[j]=12;`
			`A=12`

Pre-allocate two-dimensional array	`A=ones(10,10)`	`int A[10][10];`	`integer A(10,10)`

[a] C++ has a starting subscript of 0, but the argument in the allocation statement is the array's size.

Table 41: Array initialization constructs.

Action	MATLAB	C++	F90
Define size	`A=zeros(2,3)`[a]	`int A[2][3];`	`integer,dimension(2,3)::A`

Enter rows	`A=[1,7,-2;`
	`  3, 4, 6];`
		`int A[2][3]={`
		`  {1,7,2},`
		`  {3,4,6}`
		`};`
			`A(1,:)=(/1,7,-2/)`
			`A(2,:)=(/3,4,6/)`

[a] Optional in MATLAB, but improves efficiency.

Table 42: Array initialization constructs.

	MATLAB	**C++**	**F90**

Addition $\mathbf{C} = \mathbf{A} + \mathbf{B}$	`C=A+B`	`for (i=0; i<10; i++){`
		`  for (j=0; j<10; j++){`
		`  C[i][j]=A[i][j]+B[i][j];`
		`  }`
		`  }`
			`C=A+B`

Multiplication $\mathbf{C} = \mathbf{AB}$	`C=A*B`	`for (i=0; i<10; i++){`
		`  for (j=0; j<10; j++){`
		`  C[i][j] = 0;`
		`  for (k=0; k<10; k++){`
		`  C[i][j] += A[i][k]*B[k][j];`
		`  }`
		`  }`
		`  }`
			`C=matmul(A,B)`

Scalar multiplication $\mathbf{C} = a\mathbf{B}$	`C=a*B`	`for (i=0; i<10; i++){`
		`  for (j=0; j < 10; j++){`
		`  C[i][j] = a*B[i][j];`
		`  }`
		`  }`
			`C=a*B`

Matrix inverse $\mathbf{B} = \mathbf{A}^{-1}$	`B=inv(A)`	[a]	`B=inv(A)`[a]

[a] Neither C++ nor F90 have matrix inverse functions as part of their language definitions nor as part of standard collections of mathematical functions (like those listed in Table 7). Instead, a special function, usually drawn from a library of numerical functions, or a user defined operation, must be used.

Table 43: Elementary matrix computational routines.

C++	`int*  point, vector, matrix`
	`...`
	`point = new type_tag`
	
	`vector = new type_tag [space_1]`
	`if (vector == 0) {error_process}`
	`matrix = new type_tag [space_1 * space_2]`
	`...`
	`delete matrix`
	`...`
	`delete vector`
	`delete point`

F90	`type_tag, pointer, allocatable :: point`
	`type_tag, allocatable :: vector (:), matrix (:,:)`
	`...`
	`allocate (point)`
	`allocate (vector (space_1), STAT = my_int)`
	`if (my_int /= 0) error_process`
	`allocate (matrix (space_1, space_2))`
	`...`
	`deallocate (matrix)`
	`if (associated (point, target_name)) pointer_action...`
	`if (allocated (matrix)) matrix_action...`
	`...`
	`deallocate (vector)`
	`deallocate (point)`

Table 44: Dynamic allocation of arrays and pointers.

```
SUBROUTINE AUTO_ARRAYS (M,N, OTHER)
USE GLOBAL_CONSTANTS ! FOR INTEGER K
   IMPLICIT NONE
   INTEGER, INTENT (IN) :: M,N
   type_tag, INTENT (OUT) :: OTHER (M,N) ! dummy array

!  Automatic array allocations
   type_tag :: FROM_USE (K)
   type_tag :: FROM_ARG (M)
   type_tag :: FROM_MIX (K,N)
   ...
!  Automatic deallocation at end of scope
END SUBROUTINE AUTO_ARRAYS
```

Table 45: Automatic memory management of local scope arrays.

```
module derived_class_name
      use base_class_name
! new attribute declarations, if any
                ...
contains


    ! new member definitions
                ...
end module derived_class_name
```

Table 46: F90 Single Inheritance Form.

```
module derived_class_name
      use base_class_name, only: list_of_entities
! new attribute declarations, if any
            ...
contains


    ! new member definitions
            ...
end module derived_class_name
```

Table 47: F90 Selective Single Inheritance Form.

```
module derived_class_name
      use base_class_name, local_name => base_entity_name
! new attribute declarations, if any
            ...
contains


    ! new member definitions
            ...
end module derived_class_name
```

Table 48: F90 Single Inheritance Form, with Local Renaming.

```
module derived_class_name
      use base1_class_name
      use base2_class_name
      use base3_class_name, only: list_of_entities
      use base4_class_name, local_name => base_entity_name
! new attribute declarations, if any
            ...
contains


      ! new member definitions
            ...
end module derived_class_name
```

Table 49: F90 Multiple Selective Inheritance with Renaming.