1 *FISH* BEGINNER'S GUIDE

1.1 Introduction and Overview

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 model 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 was developed in response to requests from users who wanted to do things with Itasca software that were either difficult or impossible to do with existing program structures. Rather than incorporate many new and specialized features into the standard code, an embedded language was provided so that users could write their own functions. Some useful *FISH* functions have already been written; a library of these is provided with the It is possible for someone without experience in programming to write simple *FISH* functions or to modify some of the simpler existing functions. Section 1.2 contains an introductory tutorial for non-programmers. However, *FISH* programs can also become very complicated (which is true of code in any programming language) – for more details, refer to Section 2.

As with all programming tasks, *FISH* functions should be constructed in an incremental fashion, checking operations at each level before moving on to more complicated code. *FISH* does less error-checking than most compilers, so all functions should be tested on simple data sets before they are used for real applications.

FISH programs are simply embedded in a normal *UDEC* data file (lines following the word **DEFINE** are processed as a *FISH* function); the function terminates when the word **END** 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.

All of the *FISH* language rules and intrinsic functions are discussed in Section 2. This includes rules for syntax, data types, arithmetic, variables and functions. All *FISH* language names are described in Section 2.

1.2 Beginner's Guide and Tutorial

This section is intended for people who have run *UDEC* (at least for simple problems) but have not used the *FISH* language; no programming experience is assumed. To get the maximum benefit from the examples given here, you should try them out with *UDEC* running interactively. The short programs may be typed in directly. After running an example, give the *UDEC* command **NEW**, to "wipe the slate clean," ready for the next example. Alternatively, the more lengthy programs may be created on file and **CALL**ed when required.

Type the lines in Example 1.1 after *UDEC*'s command prompt, pressing <ENTER> at the end of each line.

Example 1.1 Defining a FISH function

```
model new
fish def abc
  abc = 22 * 3 + 5
end
```

Note that the command prompt changes to **Def>** after the first line has been typed in; then it changes back to the usual prompt when the **END** command is entered. This change in prompt lets you know whether you are sending lines to *UDEC* or to *FISH*. Normally, all lines following the **DEFINE** statement are taken as part of the definition of a *FISH* function (until the **END** statement is entered). However, if you type in a line that contains an error (e.g., you type the **=** sign instead of the **+** sign), then you will get the *UDEC* prompt back again. In this case, you should give the **NEW** command and try again from the beginning. Since it is very easy to make mistakes, *FISH* programs are normally typed into a file using an editor. These are then **CALL**ed into *UDEC* just like a regular *UDEC* data file. We will describe this process later; for now, we'll continue to work interactively. Assuming that you typed in the preceding lines without error, and that you now see the *UDEC* prompt `udec:`, you can "execute" the function **abc*** defined earlier in Example 1.1 by typing the line

```
print abc
```

The message

```
abc =  71
```

should appear on the screen. By defining the symbol **abc** (using the **DEFINE ... END** construction, as in Example 1.1), we can now refer to it in many ways using *UDEC* commands.

For example, the **PRINT** command causes the value of a *FISH* symbol to be displayed; the value is computed by the series of arithmetic operations in the line

---

* We will use **courier boldface** to identify user-defined *FISH* functions and declared variables in the text.

```
abc = 22 * 3 + 5
```

This is an "assignment statement." If an equal sign is present, the expression on the right-hand side of the equal sign is evaluated and given to the variable on the left-hand side. Note that arithmetic operations follow the usual conventions: addition, subtraction, multiplication and division are done with the signs **+**, **-**, ***** and **/**, respectively. The sign **^** denotes "raised to the power of."

We now type in a slightly different program (using the command **NEW** to erase the old one):

Example 1.2 Using a variable

```
model new
fish def abc
  hh = 22
  abc = hh * 3 + 5
end
```

Here we introduce a "variable," **hh**, which is given the value of 22 and then used in the next line. If we give the command **PRINT abc**, then exactly the same output as in the previous case appears. However, we now have two *FISH* symbols; they both have values, but one (**abc**) is known as a "function" and the other (**hh**) as a "variable." The distinction is as follows.

> *When a FISH symbol name is mentioned (e.g., in a* **PRINT** *statement), the associated function is executed if the symbol corresponds to a function. However, if the symbol is not a function name, then the current value of the symbol is used.*

The following experiment may help to clarify the distinction between variables and functions. (Before doing the experiment, note that *UDEC*'s **SET** command can be used to set the value of any user-defined *FISH* symbol, independent of the *FISH* program in which the symbol was introduced.) Now type in the following lines without giving the **NEW** command, since we want to keep our previously entered program in memory.

Example 1.3* SET*ting variables

```
fish set @abc=0
fish set @hh=0
fish list @hh
fish list @abc
fish list @hh
```

The **SET** command sets the values of both **abc** and **hh** to zero. Since **hh** is a variable, the first **PRINT** command simply displays the current value of **hh**, which is zero. The second **PRINT** command causes **abc** to be executed (since **abc** is the name of a function); the values of both **hh** and **abc** are thereby recalculated. Accordingly, the third **PRINT** statement shows that **hh** has indeed been

reset to its original value. As a test of your understanding, you should type in the slightly modified sequence shown in Example 1.4 and figure out why the displayed answers are different.

Example 1.4 Test your understanding of function and variable names

```
model new
fish def abc
  abc = hh * 3 + 5
end
fish set @hh=22
fish list @abc
fish set @abc=0 @hh=0
fish list @hh
fish list @abc
fish list @hh
```

At this stage, it may be useful to list the most important *UDEC* commands that directly refer to simple *FISH* variables or functions. (In Table 1.1, ***var*** stands for the name of the variable or function.)

Table 1.1 Commands that directly refer to FISH names

PRINT	*var*
SET	*var = value*
HISTORY	*var*

We have already seen examples of the first two (refer to Examples 1.3 and 1.4); the third case is useful when histories of things that are not provided in the standard *UDEC* list of history variables are required. Example 1.5 shows how this can be done.

Example 1.5 Capturing the history of a FISH variable

```
model new
block create polygon 0,0  0,10  10,10  10,0
block zone gen edge 10
block property mat=1 dens 1000 bulk 1e9 shear 0.7e9
block gridpoint apply vel-y 0.0 range pos-y -0.01, 0.01
block mech grav 0 -10

fish def stress_y
  zoneIdx = bl.zone(block.head)
  stress_y = bl.zo.str.yy(zoneIdx)
```

```
end

fish history @stress_y
block cycle 200
```

In this example, a history of the vertical stress in one zone is recorded. The symbols **b_zone()**, **block_head** and **z_syy()** are predefined names that permit access to *UDEC*'s data structures. We obtained the index of the first zone in the one block in our model. With that index we can access a number of parameters associated with that zone. In this case, we have accessed the vertical stress and monitored its change in a history.

In addition to the predefined variable names mentioned above, there are many other predefined objects available to a *FISH* program. These fall into several classes. One such class consists of *scalar* variables, which are single numbers. For example:

clock	clock time in hundredths of a second
pi	π
step	current step number
unbal	maximum unbalanced force
urand	random number drawn from uniform distribution between 0.0 and 1.0.

This is just a small selection; the full list is given in Section 2.5.2.

Another useful class of built-in objects is the set of *intrinsic functions*, which enable things like sines and cosines to be calculated from within a *FISH* program. A complete list is provided in Section 2.5.4. A few are given here:

abs(*a*)	absolute value of ***a***
cos(*a*)	cosine of ***a*** (***a*** is in radians)
log(*a*)	base-ten logarithm of ***a***
max(*a*,*b*)	returns maximum of ***a***, ***b***
sqrt(*a*)	square root of ***a***

An example of the use of intrinsic functions will be presented later, but now we must discuss one more way a *UDEC* data file can make use of user-defined *FISH* names:

> *Wherever a number is expected in a UDEC input line, you may substitute the name of a FISH variable or function.*

This simple statement is the key to a very powerful feature of *FISH* that allows such things as ranges, applied stresses, properties, etc. to be computed in a *FISH* function and used by *UDEC* input in symbolic form. Hence, parameter changes can be made very easily, without the need to change many numbers in an input file.

As an example, let us assume that we know the Young's modulus and Poisson's ratio of a material. Since *UDEC* needs the bulk and shear moduli, these may be derived with a *FISH* function, using Eqs. (1.1) and (1.2):

$$G = \frac{E}{2(1+\nu)} \tag{1.1}$$

$$K = \frac{E}{3(1-2\nu)} \tag{1.2}$$

Coding Eqs. (1.1) and (1.2) into a *FISH* function (called **derive**) can then be done as shown in Example 1.6:

Example 1.6 FISH functions to calculate bulk and shear moduli

```
model new
fish def derive
  s_mod = y_mod / (2.0 * (1.0 + p_ratio))
  b_mod = y_mod / (3.0 * (1.0 - 2.0 * p_ratio))
end
fish set @y_mod = 5e8 @p_ratio = 0.25
@derive
fish list @b_mod
fish list @s_mod
```

Note that here we execute the function **derive** by giving its name by itself on a line; we are not interested in its *value*, only what it *does*. If you run this example, you will see that values are computed for the bulk and shear moduli, **b_mod** and **s_mod**, respectively. These can then be used, in symbolic form, in *UDEC* input as shown in Example 1.7:

Example 1.7 Using symbolic variables in UDEC input

```
block create polygon 0,0  0,10  10,10  10,0
block zone gen edge 10
block zone cmodel assign elastic bulk=@b_mod shear=@s_mod
list zone property bulk
list zone property shear
```

The validity of this operation may be checked by printing out **bulk** and **shear** in the usual way. In these examples, our property input is given via the **SET** command (i.e., to variables **y_mod** and **p_ratio**, which stand for Young's modulus and Poisson's ratio, respectively).

Note that there is great flexibility in choosing names for *FISH* variables and functions. For instance, the underscore character (_) may be included in a name. Names must begin with a non-number and must not contain any of the arithmetic operators (+, -, /, * or ^). A chosen name should not be the same as one of the built-in (or reserved) names; Section 2.2.2 contains a complete list of names to be avoided, as well as some rules that should be followed.

In the preceding examples, we checked the computed values of *FISH* variables by giving their names explicitly as arguments to a **PRINT** command. Alternatively, we can list all current variables and functions. A printout of all current values, sorted alphabetically by name, is produced by giving the command

```
print fish
```

We now examine ways decisions can be made, and repeated operations done, in *FISH* programs. Two *FISH* statements allow specified sections of a program to be repeated many times:

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

ENDLOOP

The words **LOOP** and **ENDLOOP** are *FISH* statements, the symbol ***var*** stands for the loop variable, and ***expr1*** and ***expr2*** stand for expressions (or single variables). Example 1.8 shows the use of a loop (or repeated sequence) to produce the sum and product of the first ten integers.

Example 1.8 Controlled loop in FISH

```
model new
fish def xxx
  sum  = 0
  prod = 1
  loop n (1,10)
    sum  = sum + n
    prod = prod * n
  end_loop
end
```

```
@xxx
fish list @sum
fish list @prod
ret
```

In this case, the loop variable **n** is given successive values from 1 to 10, and the statements inside the loop (between the **LOOP** and **ENDLOOP** statements) are executed for each value. As mentioned, variable names or an arithmetic expression could be substituted for the numbers 1 or 10.

A practical use of the **LOOP** construct is to install a nonlinear initial distribution of elastic moduli in a *UDEC* grid. Suppose that the Young's modulus at a site is given by Eq. (1.3),

$$E = E_{\circ} + c\sqrt{z} \tag{1.3}$$

where z is the depth below surface, and c and $E_{\circ}$ are constants. We write a *FISH* function to install appropriate values of bulk and shear modulus in the grid, as in Example 1.9:

Example 1.9 Applying a nonlinear initial distribution of moduli

```
model new
;
block tolerance corner-round-length 0.1
block create polygon 0 -30  0 0  30 0 30 -30
block cut joint-set angle  36 0   trace 50 0  gap 0 0   spacing  8 0
block cut joint-set angle -58 0   trace 50 0  gap 0 0   spacing 12 0
block zone gen edge 1
block zone cmodel assign mohr-c
;
fish def install
  ; smoothness is the elevation tolerance on changing moduli
  bi = block.head
  loop while bi # 0
    zi = bl.zone(bi)
    loop while zi # 0
      z_depth = float(int(-bl.zone.pos.y(zi)/smoothness))
      y_mod = y_zero + cc * math.sqrt(z_depth)
      _shear = y_mod / (2.0*(1.0+p_ratio))
      _bulk  = y_mod / (3.0*(1.0-2.0*p_ratio))
      bl.zone.prop(zi,'bulk')  = _bulk
      bl.zone.prop(zi,'shear') = _shear
      zi = bl.zone.next(zi)
    end_loop
    bi = bl.next(bi)
  endloop
end
```

```
;
fish set @p_ratio=0.25 @y_zero=1e7 @cc=1e8 @smoothness=6.0
block property mat 1 dens 2000 bulk 1e8 shear 1e7
@install
ret
```

Again, you can verify correct operation of the function by printing or plotting shear and bulk moduli. A plot of the model is shown in Figure 1.1:

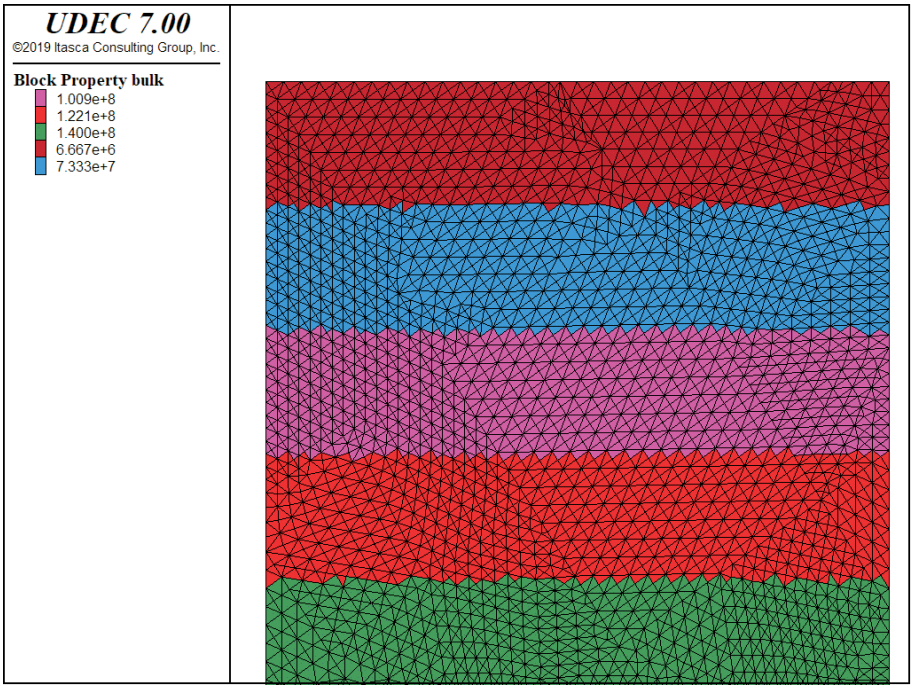

Figure 1.1 Model constructed in Example 1.9

In the function **`install`**, we have two loops: the outer loop scans through the blocks in the model, while the inner loop scans through the list of zones within each block. The parameter **`smoothness`** controls the tolerance on the change of the moduli, allowing the creation of strata of common moduli. The use of the FIN (for *FISH* INclude) file "ZMAT.FIN" gives us access to data structures within *UDEC*. This will be discussed in more depth in Section 2.5. For now, accept that this is a safe and efficient way of achieving our goal. We obtain the depth of the zone with the function **z_y(*zi*)**; the value is smoothed to provide a more distinct banding. The elastic moduli are then computed according to our rule, and assigned to the appropriate memory locations via the **fmem()** function call. More on this function is available in Section 2.5.

Having seen several examples of *FISH* programs, let's briefly examine the question of program syntax and style. A complete *FISH* statement must occupy one line; there are no continuation lines. If a formula is too long to fit on one line, then a temporary variable must be used to split the formula. Example 1.10 shows how this can be done:

Example 1.10 Splitting lines

```
model new
fish def long_sum  ;example of a sum of many things
  temp1 = v1 + v2 + v3 + v4 + v5 + v6 + v7 + v8 + v9 + v10
  long_sum = temp1 + v11 + v12 + v13 + v14 + v15
end
```

In this case, the sum of 15 variables is split into two parts. Also note also the use of the semicolon in line 2 of Example 1.10 to indicate a comment. Any characters that follow a semicolon are ignored by the *FISH* compiler, but they *are* echoed to the log file. It is good programming practice to annotate programs with informative comments. Some of the programs have been shown with *indentation* (i.e., space inserted at the beginning of some lines to denote a related group of statements). Any number of space characters may be inserted (optionally) between variable names and arithmetic operations to make the program more readable. Again, it is good programming practice to include indentation to indicate things like loops, conditional clauses and so on. Spaces in *FISH* are significant in the sense that space characters may not be inserted into a variable or function name.

One other topic that should be addressed now is that of variable *type*. You may have noticed, when printing out variables from the various program examples, that numbers are either printed without decimal points or in "E-format" (i.e., as a number with an exponent denoted by "E"). At any instant in time, a *FISH* variable or function name is classified as one of three types: *integer*, *floating-point* or *string*. These types may change dynamically, depending on context, but the casual user should not normally have to worry about the type of a variable, since it is set automatically. Consider Example 1.11:

Example 1.11 Variable types

```
model new
fish def haveone
  aa = 2
  bb = 3.4
  cc = 'Have a nice day'
  dd = aa * bb
  ee = cc + ', old chap'
end
@haveone
fish list
```

The resulting screen display is

```
                Value    Name
                -----    ----
                    2    aa
          3.4000e+000    bb
           - string -    cc
          6.8000e+000    dd
           - string -    ee
                    0    haveone
```

The variables **aa**, **bb** and **cc** are converted to *integer*, *float* and *string*, respectively, corresponding to the numbers (or strings) that were assigned to them. Integers are exact numbers (without decimal points), but are of limited range; floating-point numbers have limited precision (about six decimal places), but are of much greater range; string variables are arbitrary sequences of characters. There are various rules for conversion between the three types. For example, **dd** becomes a floating-point number because it is set to the product of a floating-point number and an integer; the variable **ee** becomes a string because it is the sum (concatenation) of two strings. The topic can get quite complicated, but it is fully explained in Sections 2.2.4 and 2.4.

There is another language element in *FISH* that is commonly used: the **IF** statement. Three statements allow decisions to be made within a *FISH* program:

IF *expr1 test expr2* **THEN**

ELSE

ENDIF

These statements allow conditional execution of *FISH* program segments; **ELSE** and **THEN** are optional. 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 expressions or single variables. 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, the program jumps to the first line after **ENDIF**. The action of these statements is illustrated in Example 1.12:

Example 1.12 Action of the* IF ELSE ENDIF *construct

```
model new
fish def abc
  if xx > 0 then
    abc = 33
  else
    abc = 11
  end_if
end
fish set @xx = 1
fish list @abc
fish set @xx = -1
fish list @abc
```

The displayed value of **abc** in Example 1.12 depends on the set value of **xx**. You should experiment with different test symbols (e.g., replace > with <).

Until now, our *FISH* programs have been invoked from *UDEC*, either by using the **PRINT** command or by giving the name of the function on a separate line of *UDEC* input. It is also possible to do the reverse (i.e., to give *UDEC* commands from within a *FISH* function). Most valid *UDEC* commands can be embedded between two *FISH* statements:

COMMAND
ENDCOMMAND

There are two main reasons for sending out *UDEC* commands from a *FISH* program. First, it is possible to use a *FISH* function to perform operations that are not possible using the predefined variables that we already discussed. Second, we can control a complete *UDEC* run with *FISH*.

As an illustration of the first use of the **COMMAND** statement, we can write a *FISH* program to place a number of cable elements around a specific segment of a tunnel.

Starting and ending angles (counterclockwise from the positive x-axis) are specified. The variable **radius1** is the radius of the tunnel, and **radius2** gives the outer ends of the cables. Example 1.13 shows the code:

Example 1.13 Automated placing of cable elements

```
model new

fish def setup
; Create vars for later use
  xCentre = 0.0         ; x-coord of tunnel centre
  yCentre = 0.0         ; y-coord of tunnel centre
  theta1  = 10.0        ; starting angle for cables
```

```
  theta2  = 160.0       ; ending angle for cables
                        ; (don't wrap back on theta1)
  radius1 = 8.0         ; radius of tunnel
  radius2 = 16.0        ; ending radius for remote end of cables
  nCables = 15          ; number of cables
end

fish def place_cables
; This example places cable elements along a given arc of tunnel.

  ; calculate angle increment between successive cables
  theta1 = math.degrad * theta1
  theta2 = math.degrad * theta2
  _angInc = (theta2 - theta1) / float(nCables - 1)
  _ang = theta1

  ; get endpoint coordinates
  loop ii (1, nCables)
    _x1 = radius1 * math.cos(_ang) + xCentre
    _y1 = radius1 * math.sin(_ang) + yCentre
    _x2 = radius2 * math.cos(_ang) + xCentre
    _y2 = radius2 * math.sin(_ang) + yCentre

    ; place the cable
    command
      cable @_x1 @_y1  @_x2 @_y2  5 2 3
    endcommand
    _ang = _ang + _angInc

  endloop

end

@setup

block create polygon -25.0, -25.0  -25.0, 25.0  25.0, 25.0  25.0, -25.0
block cut split 0.0, -25.0  0.0, 25.0
block cut split -25.0, 0.0  25.0, 0.0
block cut tunnel 0.0, 0.0, @radius1, 16
block zone gen edge 10

block delete range annulus center 0.0, 0.0  rad 0  8
@place_cables
```

Each time through the loop, rectangular coordinates are calculated from the polar data in the model. These coordinates are passed to the **CABLE** command inside the **COMMAND – ENDCOMMAND** structure, where the cable element is actually created. The results can be seen in Figure 1.2:

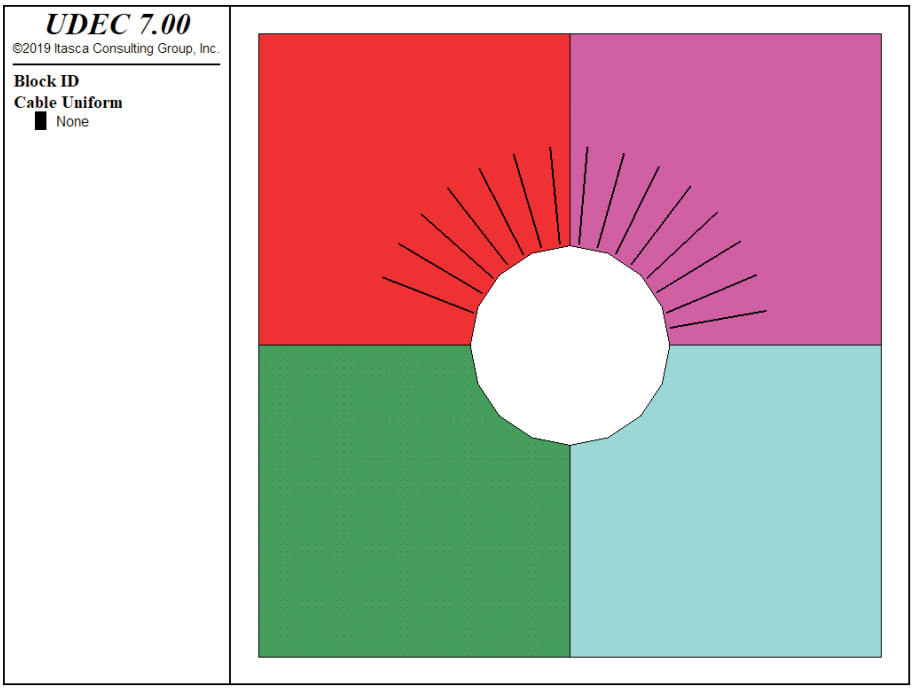

Figure 1.2 Model constructed in Example 1.13