

Introduction to ANSYS Fluent scripting – Part 2 – Accessing solver data – Detailed Examples

Description _____	2	Using a text command to delete surfaces ____	12
General notes _____	2	Changing type and settings of boundary	
Getting the path to the case file _____	2	conditions _____	13
Goal _____	2	Goal _____	13
Development _____	2	Development _____	13
Alternatives _____	6	Alternatives _____	17
Compressed definition of variables _____	7	Baffles _____	17
Backwards loop _____	7	Creating multiple post-processing surfaces	18
For-each loop _____	7	Goal _____	18
Removing default interior zones from the list		Development _____	18
of post-processing surfaces _____	8	Planning _____	18
Goal _____	8	Coding _____	30
Development _____	8	Summary _____	50
Alternatives _____	12	References _____	51

If you have questions regarding this document, please try to contact the author, first:
akram.radwan@ansys.com

Note that the content of this document is not covered by ANSYS support contracts. It is provided as self-help content for your convenience in addition to the contents of the ANSYS Fluent documentation. Always remember that ANSYS can decline to provide technical support if your project relies on Scheme scripts even if your problem description has nothing to do with Scheme.

Description

This document contains tutorial-style explanations of the examples provided in ANSYS solution 2042682 [1]. You might want to have 2042682 and 2042681 [2] ready to be able to look-up certain commands and procedures used in these examples.

The following chapters develop the example scripts line by line and also offer extensions and usage scenarios that go beyond what is available in the main document of solution 2042682.

General notes

When you develop your own Scheme scripts, it's best to take baby steps. Develop and test your script line by line. Debugging can be really painful for complex scripts.

Always use a text editor that supports context highlighting and that keeps track of parentheses. Also use a font with a fixed width to make use of the systematic indentation.

Define an alias to quickly load your Scheme script:

```
(alias 'lp (lambda () (load "mp.scm")))
```

This assumes that your file is called "mp.scm" (which is short for my-procedure).

Getting the path to the case file

Goal

Sometimes it can be useful to get either the path or the case name. It might be necessary to have the global path for file access or just to add the case name to images you write out. With RP variables it is also possible to transfer the path or the filename to a UDF.

Development

First, consider how you get case and path name.

```
(in-package cl-file-package rc-filename)
```

```
> (in-package cl-file-package rc-filename)
E:/Ansys/Scheme/panel/multiphase
```

From the output you don't know what type it is. It could be a string or a symbol, so it's best to check that with `string?` before developing the script.

```
> (string? (in-package cl-file-package rc-filename))
#t
```

Now you can be sure that the output is a string which allows you to use all string-related procedures to search for certain characters and break down the string into its individual components.

But before thinking further it's good to have robustness in mind. What happens if no case is loaded?

```
> (in-package cl-file-package rc-filename)

> (string? (in-package cl-file-package rc-filename))
#t
```

Fluent returns an empty string which could be captured and processed. Although it's probably best to just check if a case is loaded with `case-valid?` in case robustness is an issue for you. For now, you can disregard robustness.

Let's look at the return value in more detail.

In the example case used for the development it is `E:/Ansys/Scheme/panel/multiphase`. Notice three things about this output:

1. Obviously it's a Windows path because it starts with the drive letter `E`:
2. Windows uses a backslash `\` to navigate through a folder structure. But Fluent prints a forward slash `/` like it is used on Linux systems. This makes it easier for you to work independent of the operating system. If you need to identify the root folder you might need to take special care of the operating system, though.
3. "multiphase" is the file name. But there is no file extension available. It could be `*.cas` but it could also be `*.cas.gz` or `*.cas.h5`.

If you only need the filename, you can use `strip-directory` as explained in ANSYS solution 2038952 [3]. This does not solve the issue of the third observation, though. Identifying the suffix is more complicated and touched in ANSYS solution 2042682 after the description of `strip-directory`.

To get only the path, you need to remove the characters after the last forward slash from the string. You can analyze the string in forward or backward direction. It depends what you want to know which one is more efficient. The forward method is often preferred but there is no obvious reason for it in Scheme. You can find a version that searches backwards later.

First of all, define a procedure that you can execute over and over again and store the full path in a local variable for easy access.

```
001 (define (my-get-path)
002   (let ((my-full-path (in-package cl-file-package rc-filename)))
003     )
004   )
005 )
```

Code fragment 1: Outline for procedure "my-get-path" (2042682_Example1_01.scm)

Note that the Scheme best-practices suggest that all closing parentheses are on the same line. Especially for beginners this might be confusing. You can keep better track of the parentheses if you put the closing symbol below the opening one. This can be collapsed later to reduce the unnecessary white space in the file.

In the next step think about how you can access each individual character. Obviously you need a loop that goes either from start to end or from end to start. Both can be done with a `do` loop or with a `for-each` loop after converting the string to a list.

For a `do` loop you need initial and abort conditions. The loop should go over all characters of the string which means you need the number of characters in the string to abort the loop without errors. You can get the number of characters with `string-length`. This should be stored in another variable that the procedure `string-length` does not have to be executed over and over again.

During development of a script you should always test if the partial script is doing what you expect. Print variables to the console often. These display statements can be removed later.

```

001 (define (my-get-path)
002   (let ((my-full-path (in-package cl-file-package rc-filename))
003         (my-pathlength)
004         )
005     (set! my-pathlength (string-length my-full-path))
006     (display my-pathlength)
007   )
008 )

```

Code fragment 2: Get the length of the full path (2042682_Example1_02.scm)

```

> (my-get-path)
32

```

The line numbers with modifications are marked purple that you can locate changes easily.

Line 3: Additional definition of the new variable

Line 5: Assigning the length of the string that holds the full path to the new variable

Line 6: Show the result

Then loop over all characters with a do loop. Start with the first one which has the index 0 and increase this index for each loop by one. When the index is equal to the length of the string the loop stops without executing any code. Note that the last value of i is not printed. This is important because the indexing starts at zero.

```

001 (define (my-get-path)
002   (let ((my-full-path (in-package cl-file-package rc-filename))
003         (my-pathlength)
004         )
005     (set! my-pathlength (string-length my-full-path))
006     (display my-pathlength) (newline)
007     (do ((i 0 (+ i 1)))
008         ((eqv? i my-pathlength))
009         (begin (display i) (display " ")))
010   )
011 )
012 )

```

Code fragment 3: Loop over all characters and check the index (2042682_Example1_03.scm)

```

> (my-get-path)
32
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 #f

```

Line 7: Define the do loop with the counter variable i that starts with zero and is increased by 1 after each loop

Line 8: The first statement after the loop definition itself is the abort condition. Here the loop aborts when the index is identical with my-pathlength. Nothing special happens then, since there are no additional statements the script stops.

Line 9: Display the loop index variable followed by a white space.

You can access the individual characters of a string with string-ref. This is required to compare it to the forward slash. But before you do that it's better to display the result again to be sure you get the correct characters.

```

001 (define (my-get-path)
002   (let ((my-full-path (in-package cl-file-package rc-filename))
003         (my-pathlength)
004         )
005     (set! my-pathlength (string-length my-full-path))
006     (display my-pathlength) (newline)
007     (do ((i 0 (+ i 1)))
008         ((eqv? i my-pathlength))
009         (begin
010           (display (string-ref my-full-path i)) (display " ")
011         )
012     )
013 )
014 )

```

Code fragment 4: Replace the index by the character (2042682_Example1_04.scm)

```

> (my-get-path)
32
E : / A n s y s / S c h e m e / p a n e l / m u l t i p h a s e #f

```

Now you can locate the index of the last /. To compare two characters you can use char=?. The index should be stored in a new variable that you can use it later to get your string.

You can also remove the display lines that you no longer need. But keep in mind that you might break what you have written already. Often it's better to just comment them out with a semicolon. To save space they are no longer included in the example code.

```

001 (define (my-get-path)
002   (let ((my-full-path (in-package cl-file-package rc-filename))
003         (my-pathlength)
004         (my-last-folder 0)
005         )
006     (set! my-pathlength (string-length my-full-path))
007     (do ((i 0 (+ i 1)))
008         ((eqv? i my-pathlength))
009         (begin
010           (if (char=? #\/ (string-ref my-full-path i))
011               (set! my-last-folder i))
012           (display my-last-folder) (display " ")
013         )
014     )
015 )
016 )

```

Code fragment 5: Identify the character index that separates the folders (2042682_Example1_05.scm)

```

> (my-get-path)
0 0 2 2 2 2 2 2 8 8 8 8 8 8 8 15 15 15 15 15 15 21 21 21 21 21 21 21 21 21 21 #f

```

Line 10: Check if the current character is identical with the forward slash. Execute line 11 if it is, continue with the next loop if it's not

Line 11: Set the variable my-last-folder to the current index

Line 12: Display the result after the loop finished

Now that you know the index of the last / you can store the path in a new variable. You can use substring to get the string from the beginning to the stored index. Then you can print the result with display or use it as return value.

The begin inside of the do loop can be neglected now because the if statement could be written in a single line.

```

001 (define (my-get-path)
002   (let ((my-full-path (in-package cl-file-package rc-filename))
003         (my-pathlength)
004         (my-last-folder 0)
005         (my-path)
006         )
007     (set! my-pathlength (string-length my-full-path))
008     (do ((i 0 (+ i 1)))
009         ((eqv? i my-pathlength))
010         (if (char=? #\ / (string-ref my-full-path i))
011             (set! my-last-folder i)
012         )
013     )
014     (set! my-path (substring my-full-path 0 my-last-folder))
015     my-path
016 )
017 )

```

Code fragment 6: Strip the file name from the path (2042682_Example1_06.scm)

```

> (my-get-path)
E:/Ansys/Scheme/panel

```

Note that the last character is not included in the output because substring includes the start but excludes the end character.

Finally, clean up the script and remove all unnecessary pieces like left-over display commands. You can also compress it by putting all the closing parentheses in a single line which is the best-practice for Scheme programming. But it might make the code more difficult to read if you are new to Scheme.

```

001 (define (my-get-path)
002   (let ((my-full-path (in-package cl-file-package rc-filename))
003         (my-pathlength)
004         (my-last-folder 0)
005         (my-path))
006     (set! my-pathlength (string-length my-full-path))
007     (do ((i 0 (+ i 1)))
008         ((eqv? i my-pathlength))
009         (if (char=? #\ / (string-ref my-full-path i))
010             (set! my-last-folder i)))
011     (set! my-path (substring my-full-path 0 my-last-folder))
012     my-path))

```

Code fragment 7: Cleanup the code (2042682_Example1_07_final.scm)

Alternatives

The alternatives are not explained in detail. You can use them as reference to investigate different methods to get to the same result. Some of these alternatives might be faster but it really depends on the input string.

Compressed definition of variables

Instead of defining the local variable `my-pathlength` in line 3 and setting it in line 6 you can combine it in a single statement. But you need to replace `let` with `let*` in line 2.

```
001 (define (my-get-path)
002   (let* ((my-full-path (in-package cl-file-package rc-filename))
003         (my-pathlength (string-length my-full-path))
004         (my-last-folder 0)
005         (my-path))
006     (do ((i 0 (+ i 1)))
007         ((eqv? i my-pathlength))
008         (if (char=? #\ / (string-ref my-full-path i))
009             (set! my-last-folder i)))
010     (set! my-path (substring my-full-path 0 my-last-folder))
011     my-path))
```

Code fragment 8: Simplified variable definition (2042682_Example1_08_alternative1.scm)

Backwards loop

As described before, you could also search backwards. The `do` loop looks different for that version.

```
001 (define (my-get-path)
002   (let* ((my-full-path (in-package cl-file-package rc-filename))
003         (my-pathlength (string-length my-full-path))
004         (my-last-folder 0)
005         (my-path))
006     (do ((i (- my-pathlength 1) (- i 1)))
007         ((char=? #\ / (string-ref my-full-path i)) (set! my-last-folder i)))
008     (set! my-path (substring my-full-path 0 my-last-folder))
009     my-path))
```

Code fragment 9: Searching for path separation backwards instead of forward (2042682_Example1_09_alternative2.scm)

Especially for long strings this version is faster than the first one. Note that not only the loop itself but also the abort condition needs to be changed. But there is no code that is executed inside the loop except checking for the abort condition.

If you have trouble understanding this code, you can add display commands like you have seen it before to analyze how this works. Don't forget the `begin` statements where required.

For-each loop

Instead of a `do` loop you can also use a `for-each` loop. This changes the structure of the code a bit.

```
001 (define (my-get-path)
002   (let* ((my-full-path (in-package cl-file-package rc-filename))
003         (my-pathlength (string-length my-full-path))
004         (my-last-folder 0)
005         (my-path)
006         (i 0))
007     (for-each (lambda (c)
008                 (if (char=? #\ / c)
009                     (set! my-last-folder i)
010                     (set! i (+ i 1))))
011              (string->list my-full-path))
012     (set! my-path (substring my-full-path 0 my-last-folder))
013     my-path))
```

Code fragment 10: Using a for-each loop instead of a do loop (2042682_Example1_10_alternative2.scm)

You could also search backwards by reversing the list.

Removing default interior zones from the list of post-processing surfaces

Goal

There is not much you can do with default interior zones from a post-processing perspective. Nevertheless, Fluent creates them automatically from the boundary zone. You can use a Scheme script to remove all default interior zones. Of course, they stay in the boundary zones list but they do not get in the way during post-processing. See also solution 2042751 [4].

Development

First, think about the procedures you need:

- List of all surfaces: (inquire-surface-names)
- List of all boundaries: (inquire-face-thread-names)
- Check if a boundary is a default interior: (%is-default-interior? <id>)
- Delete a surface: (delete-surfaces <list>) or a text command

Then start your new script with the basic skeleton. You probably need local variables, so you can also consider the let statement. You need a variable inside the list or Fluent reports an error when you load the file.

```
001 (define (remove-default-interior)
002   (let ((some-var))
003     )
004   )
```

Code fragment 11: Basic structure of the procedure "remove-default-interior" (2042682_Example2_01.scm)

The next step is the loop over all boundary zones to get the names.

```
001 (define (remove-default-interior)
002   (let ((some-var))
003     (for-each (lambda (zone-name)
004                 (display zone-name) (newline)
005               )
006             (inquire-face-thread-names))
007   )
008   )
```

Code fragment 12: Loop over all surfaces (2042682_Example2_02.scm)

Right now it just prints all surface names to the console.

Important:

When you just read a mesh boundary names and surface names are identical. To be sure you get the correct list you can add some post-processing surfaces. These additional surfaces should not be printed by the current state of the script.

After you are sure you get the correct list you can comment out line 4 and add a logic to detect if the current boundary zone is a default interior. Remember that %is-default-interior? takes the boundary id as argument. Therefore, it's necessary to convert the zone name to the id.


```

001 (define (remove-default-interior)
002   (let ((some-var))
003     (for-each (lambda (zone-name)
004       ; (display zone-name) (newline)
005       (if (%is-default-interior? (thread-name->id zone-name))
006         (begin
007           (display zone-name) (display " is default interior") (newline)
008         )
009       )
010     )
011     (inquire-face-thread-names))
012   )
013 )

```

Code fragment 13: Check if the surfaces are of the type default-interior (2042682_Example2_03.scm)

Now you get a list of the default interiors. Double check that this is correct. They should be named “interior-<zone-name>”, e.g. “interior-fluid” for the cell zone “fluid”.

You have two options how you want to delete these unwanted zones.

1. Use a text command to delete them one by one. This can take some time if you have a lot of zones.
2. Use a Scheme procedure to delete them all at once. This can be more efficient for many zones but will be slower if you have only a few zones. But in that case you should not notice the speed difference anyway.

The Scheme procedure (delete-surfaces <list>) takes a list of zone ids as argument. Therefore, it's necessary to build the list within the loop and delete the surfaces afterwards.

To create the list you need a new variable. You can replace the placeholder some-var in line 2. It's important that this is a list already, otherwise you can't add items to the list.

Like before, you can comment out line 7 for now.

```

001 (define (remove-default-interior)
002   (let ((default-interior-list '()))
003     (for-each (lambda (zone-name)
004       ; (display zone-name) (newline)
005       (if (%is-default-interior? (thread-name->id zone-name))
006         (begin
007           ; (display zone-name) (display " is default interior") (newline)
008           (set! default-interior-list (append default-interior-list (list
009             (surface-name->id zone-name))))
010         )
011       )
012     )
013     (inquire-face-thread-names))
014     (display default-interior-list)
015   )

```

Code fragment 14: Create a list of all default interior zones (2042682_Example2_04.scm)

The output in line 13 can be improved. It's good to print some sort of status message in the end. To get the names back from the ids you can do the reverse operation. Alternatively, you could have stored names and ids in separate lists which would be faster.

```

001 (define (remove-default-interior)
002   (let ((default-interior-list '()))
003     (for-each (lambda (zone-name)
004       ; (display zone-name) (newline)
005       (if (%is-default-interior? (thread-name->id zone-name))
006         (begin
007           ; (display zone-name) (display " is default interior") (newline)
008           (set! default-interior-list (append default-interior-list (list
009             (surface-name->id zone-name))))
010         )
011       )
012     (inquire-face-thread-names))
013     (display (format #f "The following surfaces are identified as default
interior zones\n~a" (map surface-id->name default-interior-list)))
014     (newline)
015   )

```

Code fragment 15: Print all surfaces that are in the list (2042682_Example2_05.scm)

Before you delete the surfaces think about robustness. You can be sure that for a new case boundary zones and surfaces have the same names. But you don't know if a user has modified boundary or surface names. Therefore, you should check if the boundary name of the default interior exists in the list of surfaces. This is also useful to avoid adding surfaces to the list that are deleted already.

```

001 (define (remove-default-interior)
002   (let ((default-interior-list '()))
003     (for-each (lambda (zone-name)
004       ; (display zone-name) (newline)
005       (if (%is-default-interior? (thread-name->id zone-name))
006         (begin
007           ; (display zone-name) (display " is default interior") (newline)
008           (if (not (eqv? #f (member zone-name (inquire-surface-names))))
009             (set! default-interior-list (append default-interior-list
010               (list (surface-name->id zone-name))))
011           )
012         )
013       )
014     (inquire-face-thread-names))
015     (display (format #f "The following surfaces are identified as default
interior zones\n~a" (map surface-id->name default-interior-list)))
016     (newline)
017   )

```

Code fragment 16: Check if the zone for the surface exists before adding it to the list (2042682_Example2_06.scm)

This check is a simple if-statement that is added in line 8. (inquire-surface-names) returns a list with the available surfaces. (member zone-name ...) checks if the content of zone-name exists in this list. Remember that most lists contain symbols instead of strings. If a zone can't be found member returns #f but it returns the zone name if it exists. Therefore, it's better to check for the return value #f which is checked with the eqv? statement. Now this returns #t but nothing should happen if the zone has no surface with the same name. Therefore, it is negated with the not-statement which ensures that the statement in line 9 is only executed when the surface does exist.

Then you can delete the recognized surfaces.

```
001 (define (remove-default-interior)
002   (let ((default-interior-list '()))
003     (for-each (lambda (zone-name)
004       ; (display zone-name) (newline)
005       (if (%is-default-interior? (thread-name->id zone-name))
006         (begin
007           ; (display zone-name) (display " is default interior") (newline)
008           (if (not (eqv? #f (member zone-name (inquire-surface-names))))
009             (set! default-interior-list (append default-interior-list
010               (list (surface-name->id zone-name))))
011           )
012         )
013       )
014     (inquire-face-thread-names))
015     (display (format #f "The following surfaces are identified as default
interior zones\n~a" (map surface-id->name default-interior-list)))
016     (delete-surfaces default-interior-list)
017   )
018 )
```

Code fragment 17: Delete all identified surfaces (2042682_Example2_07.scm)

Check that it removed the correct surfaces.

If you execute the procedure again nothing happens because the list is empty. You can add a warning message with an additional if statement for the convenience of the user.

```
001 (define (remove-default-interior)
002   (let ((default-interior-list '()))
003     (for-each (lambda (zone-name)
004       ; (display zone-name) (newline)
005       (if (%is-default-interior? (thread-name->id zone-name))
006         (begin
007           ; (display zone-name) (display " is default interior") (newline)
008           (if (not (eqv? #f (member zone-name (inquire-surface-names))))
009             (set! default-interior-list (append default-interior-list
010               (list (surface-name->id zone-name))))
011           )
012         )
013       )
014     (inquire-face-thread-names))
015     (if (eqv? default-interior-list '())
016       (display "No default interior zones left as surface\n")
017       (begin
018         (display (format #f "The following surfaces are identified as
default interior zones\n~a" (map surface-id->name default-interior-list)))
019         (delete-surfaces default-interior-list)
020       )
021     )
022   )
023 )
```

Code fragment 18: Improve robustness for empty lists (2042682_Example2_08.scm)

Now you can clean up the code to get rid of the comments and the unnecessary white spaces. Some of the begin statements can also be removed.

```

001 (define (remove-default-interior)
002   (let ((default-interior-list '()))
003     (for-each (lambda (zone-name)
004       (if (%is-default-interior? (thread-name->id zone-name))
005         (if (not (eqv? #f (member zone-name (inquire-surface-names))))
006           (set! default-interior-list (append default-interior-list (list
007             (surface-name->id zone-name))))))
008       (inquire-face-thread-names))
009     (if (eqv? default-interior-list '())
010       (display "No default interior zones left as surface\n")
011       (begin
012         (display (format #f "The following surfaces are identified as
013         default interior zones\n~a" (map surface-id->name default-interior-list)))
014         (newline)
015         (delete-surfaces default-interior-list))))))

```

Code fragment 19: Cleanup the code (2042682_Example2_09_final.scm)

Alternatives

Using a text command to delete surfaces

Instead of creating the list of surfaces you can delete them within the loop with the text command /surface/delete-surface. This simplifies the code considerably.

```

001 (define (remove-default-interior)
002   (for-each (lambda (zone-name)
003     (if (%is-default-interior? (thread-name->id zone-name))
004       (if (not (eqv? #f (member zone-name (inquire-surface-names))))
005         (ti-menu-load-string (format #f "/surface/delete-surface ~a" zone-
006           name))))))
007   (inquire-face-thread-names))

```

Code fragment 20: Alternative approach without lists (2042682_Example2_10_alternative.scm)

Changing type and settings of boundary conditions

Goal

Sometimes it is necessary to change many boundary conditions simultaneously. In general, this is not recommended because changing the type of a boundary condition takes a lot of time, especially when Fluent runs on many cores on a cluster. But there are scenarios when this is required.

In this example all zones that start with a user-defined prefix should be set to a wall with a user-defined heat flux. See also ANSYS solution 2039572 [5].

Development

Before you start with the script think about the required text commands.

To set a boundary zone to a certain type you can use

```
/define/boundary-conditions/zone-type <zone-name> <type>
```

The text command doesn't take a list of zones which means it has to be invoked for each zone separately. As long as the type exists there is also no issue with robustness.

To define the settings of a wall you need a different text command

```
/define/boundary-conditions/wall <arguments>
```

Unfortunately, this is a text command with a variable number of arguments.

Without any active models it takes three arguments as long as the answer to the second and third question is "no".

```
> /define/boundary-conditions/wall
(wall1 wall2 wall3)
zone id/name [wall1]
Wall Motion [motion-bc-stationary]: Change current value? [no]
Shear Boundary Condition [shear-bc-noslip]: Change current value? [no]
```

With active energy equation it takes 13 arguments.

```
> /define/boundary-conditions/wall
(wall1 wall2 wall3)
zone id/name [wall1]
Wall Thickness (m) [0]
Use Profile for Heat Generation Rate? [no]
Heat Generation Rate (w/m3) [0]
material-name [aluminum]: Change current value? [no]
Thermal BC Type [heat-flux]: Change current value? [no]
Use Profile for Heat Flux? [no]
Heat Flux (w/m2) [0]
Enable shell conduction? [no]
Wall Motion [motion-bc-stationary]: Change current value? [no]
Shear Boundary Condition [shear-bc-noslip]: Change current value? [no]
Use Profile for Convective Augmentation Factor? [no]
Convective Augmentation Factor [1]
```

With energy and turbulence, it has 17 arguments, radiation models add additional arguments that depend on the chosen model. And to make the confusion complete, baffles have also a different number of arguments compared to walls that face to the outside. Furthermore, it is possible that the number of arguments change with different Fluent versions because of model changes.

Making this text command really robust will be a nightmare.

Instead of thinking about each possible version of this text command, let's just consider laminar flow with active energy equation for now. All other models have to be disabled.

As usual, start with the outer skeleton that defines the name and the two parameters the procedure requires. You can also add if statements to add at least a little bit of robustness. You should check if the parameters have the expected type and you can check if the energy equation is active.

```

001 (define (change-zone-wall-hflux zone-prefix hflux)
002   (if (rf-energy?)
003     (if (and (string? zone-prefix) (number? hflux))
004       (let ((some-var))
005         (display "Basic checks successful\n"))
006       )
007     (display "Error: Invalid arguments! First argument has to be a
string, the second has to be a number.\n"))
008   )
009   (display "Error: Energy equation has to be active!\n")
010 )
011 )

```

Code fragment 21: Basic structure of the procedure "change-zone-wall-hflux" with some status messages (2042682_Example3_01.scm)

You need to find all boundary zones that start with the user-defined prefix. The required procedure is get-thread-list.

```

001 (define (change-zone-wall-hflux zone-prefix hflux)
002   (if (rf-energy?)
003     (if (and (string? zone-prefix) (number? hflux))
004       (let ((zone-list))
005         ; (display "Basic checks successful\n")
006         (set! zone-list (get-thread-list (string-append zone-prefix "*")))
007         (display zone-list)
008       )
009       (display "Error: Invalid arguments! First argument has to be a
string, the second has to be a number.\n"))
010     )
011     (display "Error: Energy equation has to be active!\n")
012   )
013 )

```

Code fragment 22: Get a list of all matching zones (2042682_Example3_02.scm)

Note that the list is empty if no boundary zone with the specified prefix exists. This can be captured with another if statement.

```

001 (define (change-zone-wall-hflux zone-prefix hflux)
002   (if (rf-energy?)
003     (if (and (string? zone-prefix) (number? hflux))
004       (let ((zone-list))
005         ; (display "Basic checks successful\n")
006         (set! zone-list (get-thread-list (string-append zone-prefix "*")))
007         (if (eqv? zone-list '())
008             (display (format #f "No zone with the specified prefix ~a found"
zone-prefix))
009             (display zone-list)
010         )
011       )
012     (display "Error: Invalid arguments! First argument has to be a
string, the second has to be a number.\n")
013   )
014   (display "Error: Energy equation has to be active!\n")
015 )
016 )

```

Code fragment 23: Check if the list is empty (2042682_Example3_03.scm)

Since the text commands don't take lists, line 9 has to be replaced with a loop over the list. For-each is best suited because it loops over each element of the list and it provides the current item to work with.

```

001 (define (change-zone-wall-hflux zone-prefix hflux)
002   (if (rf-energy?)
003     (if (and (string? zone-prefix) (number? hflux))
004       (let ((zone-list))
005         ; (display "Basic checks successful\n")
006         (set! zone-list (get-thread-list (string-append zone-prefix "*")))
007         (if (eqv? zone-list '())
008             (display (format #f "No zone with the specified prefix ~a found"
zone-prefix))
009             (for-each (lambda (zone)
010                         (display zone) (newline)
011                       )
012                     zone-list
013             )
014         )
015     )
016     (display "Error: Invalid arguments! First argument has to be a
string, the second has to be a number.\n")
017   )
018   (display "Error: Energy equation has to be active!\n")
019 )
020 )

```

Code fragment 24: Loop over all identified zones (2042682_Example3_04.scm)

Always remember that “zone-list” in the above example is a variable that contains a list. This is the last argument for the for-each loop and may not be encapsulated in parentheses. Each individual item of that list is passed to the variable “zone” defined in the lambda environment.

Since the for-each loop is a single statement with everything inside, it is not necessary to use begin statements to allow multiple lines of code inside the if statement.

Now you can add the text commands instead of the display of the zone in line 9.

```

001 (define (change-zone-wall-hflux zone-prefix hflux)
002   (if (rf-energy?)
003     (if (and (string? zone-prefix) (number? hflux))
004       (let ((zone-list))
005         ; (display "Basic checks successful\n")
006         (set! zone-list (get-thread-list (string-append zone-prefix "*")))
007         (if (eqv? zone-list '())
008             (display (format #f "No zone with the specified prefix ~a found"
zone-prefix))
009             (for-each (lambda (zone)
010               ; (display zone) (newline)
011               (ti-menu-load-string (string-append "/define/boundary-
conditions/zone-type " (symbol->string zone) " wall\n"))
012               (ti-menu-load-string (format #f "/define/boundary-
conditions/wall ~a 0 no 0 no yes heat-flux no ~a no no no 1\n\n"
(symbol->string zone) hflux))
013             )
014             zone-list
015           )
016         )
017       (display "Error: Invalid arguments! First argument has to be a
018 string, the second has to be a number.\n")
019     )
020     (display "Error: Energy equation has to be active!\n")
021   )
022 )

```

Code fragment 25: Change type and settings of all identified zones (2042682_Example3_05.scm)

Remember that the text command in line 12 is only valid for laminar flow without any additional models except energy active. Furthermore, it only applies to outer boundaries. Internal two-sided walls (baffles) have a different number of arguments.

The text command in line 12 takes the zone name as string, not as symbol. Therefore, you need to convert it to a string, first.

Different models can be added with if statements before line 12.

After cleanup you get the following script.


```

001 (define (change-zone-wall-hflux zone-prefix hflux)
002   (if (rf-energy?)
003     (if (and (string? zone-prefix) (number? hflux))
004       (let ((zone-list))
005         (set! zone-list (get-thread-list (string-append zone-prefix "*"))))
006       (if (eqv? zone-list '())
007         (display (format #f "No zone with the specified prefix ~a found"
zone-prefix))
008         (for-each (lambda (zone)
009                     (ti-menu-load-string (string-append "/define/boundary-
conditions/zone-type " (symbol->string zone) " wall\n")))
010                     (ti-menu-load-string (format #f "/define/boundary-
conditions/wall ~a 0 no 0 no yes heat-flux no ~a no no no no 1\n\n"
(symbol->string zone) hflux)))
011                     zone-list)))
012     (display "Error: Invalid arguments! First argument has to be a
string, the second has to be a number.\n"))
013     (display "Error: Energy equation has to be active!\n"))))

```

Code fragment 26: Cleanup the code (2042682_Example3_06_final.scm)

Alternatives

Baffles

For coupled walls that consist of the wall and wall-shadow pair the heat flux should not be defined. In fact, the text command doesn't work with the specified number of parameters.

To avoid conflicts you can check if the zone is a wall with (wall-thread? <id>).

It is possible to get the shadow id with the procedure (%get-coupled-wall-shadow <id>). This returns an empty list if no shadow zones exist. You can use this to skip the second text command.

```

001 (define (change-zone-wall-hflux zone-prefix hflux)
002   (if (rf-energy?)
003     (if (and (string? zone-prefix) (number? hflux))
004       (let ((zone-list))
005         (set! zone-list (get-thread-list (string-append zone-prefix "*"))))
006       (if (eqv? zone-list '())
007         (display (format #f "No zone with the specified prefix ~a found"
zone-prefix))
008         (for-each (lambda (zone)
009                     (if (not (wall-thread? (zone-name->id zone)))
010                       (ti-menu-load-string (string-append "/define/boundary-
conditions/zone-type " (symbol->string zone) " wall\n")))
011                     (if (eqv? '()) (%get-coupled-wall-shadow (zone-name->id
zone))))
012                     (ti-menu-load-string (format #f "/define/boundary-
conditions/wall ~a 0 no 0 no yes heat-flux no ~a no no no no 1\n" (symbol-
>string zone) hflux))))))
013                     zone-list)))
014     (display "Error: Invalid arguments! First argument has to be a
string, the second has to be a number.\n"))
015     (display "Error: Energy equation has to be active!\n"))))

```

Code fragment 27: Consider baffles and exclude them from the list (2042682_Example3_07_alternative.scm)

Creating multiple post-processing surfaces

Goal

If you need many post-processing surfaces, you can adjust the following script to your needs. See also ANSYS solution 2042772 [6] for an alternate version with panel. This is also discussed in solution 2042683 [7]. Solutions 912 [8] and 1030 [9] can also be of interest to get additional ideas how to adjust the script for different applications.


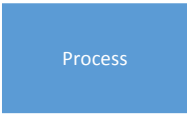
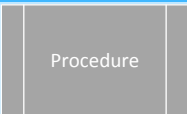



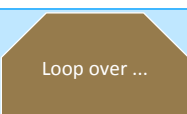
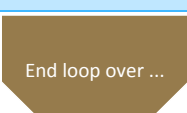
A simple post-processing surface is an iso-surface. It needs only two inputs, the Cartesian axis and the distance from the origin, to define such a surface. The goal is to create a user-defined number of these surfaces and distribute them within a user-defined region.

Development

In the previous examples you have seen a more or less chaotic approach to reach the goal. Instead of starting the script right away and extend it until it does what it should, you can also plan the script with different charts. The steps are very similar to before, though.

Planning

There are many different standards available to create such charts that help to design the code before actually coding. For this example, flow charts are used with the following symbols:

Symbol	Description
	Start or end of a procedure, sub-procedure or partial procedure
	A process that can consist of a single line or multiple lines of code. This can also refer to complex procedures available in Fluent that are not coded by you
	A procedure, sub-procedure or code that is specified in its own flow chart.
	Input or output of data, mostly printing of status or error messages.
	A decision that can be a simple if-statement or a more complex case structure.
	A connector for multiple paths or a result of a decision (e.g. Yes, No or a case).
	Start a loop
	End a loop. Usually there is no arrow back to the beginning of the loop.

First, think about the required user-inputs. In this case you need at least five different inputs:

1. Number of iso-surfaces (essential)
2. Direction (essential)
3. Start coordinate (optional, use domain boundaries as default)
4. End coordinate (optional, use domain boundaries as default)
5. Name pattern (optional, default name pattern can be used)

The user has to provide the first two inputs when calling the procedure. The last three are optional. But if they are specified, the user has to add them after the essential arguments.

The name pattern requires some more thoughts. It would be easy just to use some sort of default or user-defined prefix and add an index in the end to have unique names. But it could also be useful to have the coordinates in the name of the planes. The user should decide how the name should look like within these boundaries:

- Prefix-index (default)
- Prefix-index-coordinates
- Prefix-coordinates

This means that three inputs are required for the name pattern which gives you five optional arguments in total.

From a coding perspective, it is important to check if the inputs are valid. This should be done immediately after the procedure is called. If you are coding not only for yourself, you should also give useful and describing error or warning messages. These usability enhancements require most of the coding.

If you want to make your script robust and easy to use, include this during the planning stage. The code can look very different if these steps are disregarded.

After you are sure all inputs are valid and exist in a usable format, you can create the iso-surfaces. This requires multiple steps that you can specify in more detail later.

Let's look at checking the inputs, first.

The user provides them when calling the procedure. You only have to make sure that they are in the correct order and valid.

For now, you just have to decide if you want to abort the script on the first error or if you want to evaluate all inputs. The second option is probably a bit more user-friendly because you can show the user multiple error messages if several inputs are invalid.

Before going into detail, just specify the order in which you want to check the inputs. Parallel processing is not possible with Scheme in Fluent.

The order is more or less arbitrary. It can make it easier to understand the code if you do the checks in the same order in which you expect the arguments during the call of the procedure.

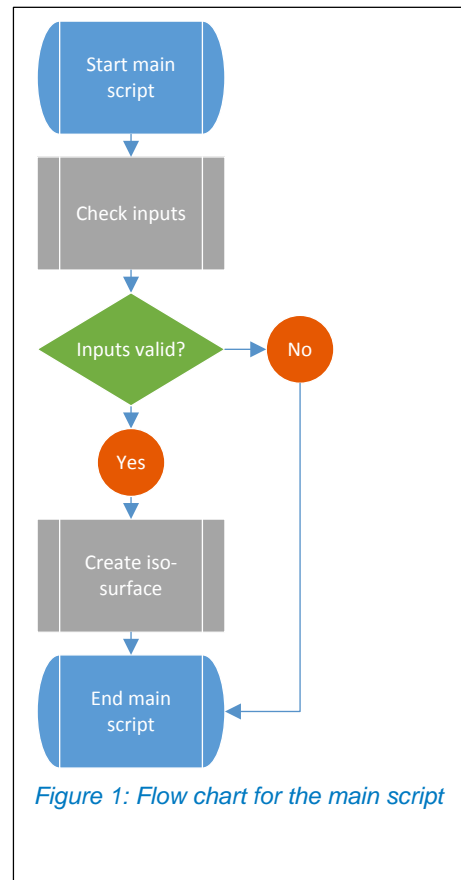


Figure 1: Flow chart for the main script

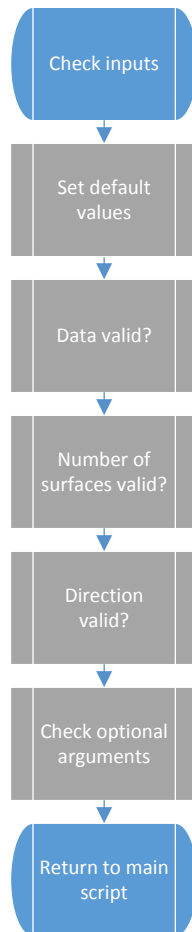


Figure 2: Flow chart for the procedure "check inputs"

Note that the first check in the chart is required because the text command to create iso-surfaces requires valid data.

Now that the order is specified, you can think about the individual procedures or code fragments.

"Set default values" should be defined later. For now, this is only a place holder because you usually need a number of default values for more complex scripts. You already know that you need default values for the name pattern and the start and end coordinates. But there might be more. Therefore, it's best to keep track of all the variables you encounter during the planning stage and patch this together later. It might even be necessary to squeeze in default values at different locations later.

"Data valid?" is a simple check. Fluent returns with (data-valid?) a Boolean value that you can use directly in an if-statement. Now it's time to decide how you want to abort the script later. You can use an error or an execution flag. Essentially both are Boolean values. The error flag has a default value of false and turns true if there is an error. An execution flag is the opposite. Usually, the error flag is easier to implement. But it is more dangerous to set the default state to "no errors". Nevertheless, you can use an error flag in this case.

Therefore, if the check returns #f, you set the error flag, show a describing error message and continue with the next check. If it returns #t, you simply continue without doing anything.

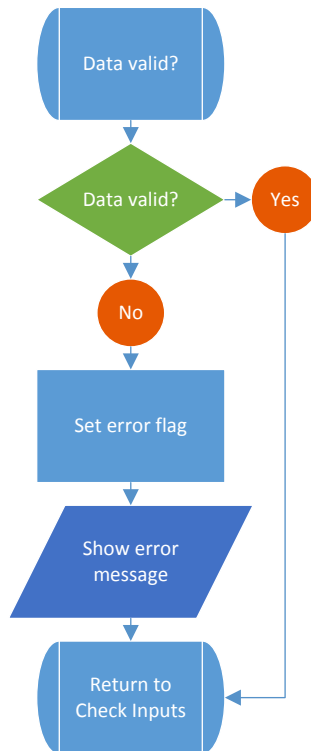


Figure 3: Flow chart for the procedure "data valid"

"Number of surfaces valid?" is very similar but it contains two checks. You should check if the input is an integer and if it is, if it's larger than zero. It's also good practice to set a maximum value to avoid Fluent to create too many surfaces which can have a negative impact on stability and reaction time of the GUI.

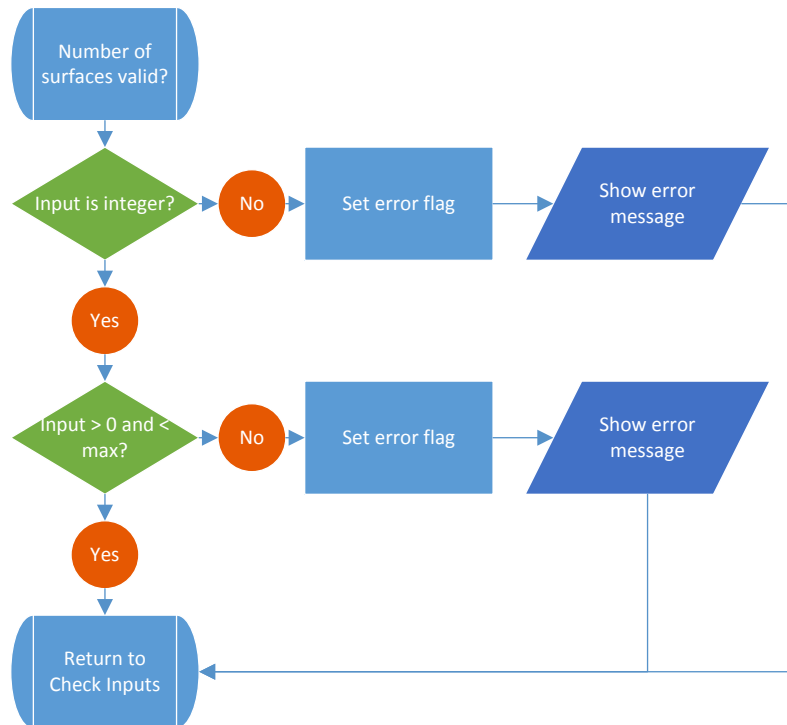


Figure 4: Flow chart for the procedure "number of surfaces valid"

From the chart it looks like there is duplicate code, which would be bad. Duplicate code should be avoided at all cost. But the error messages should be different for both errors to show the user what's wrong. Therefore, there is no duplicate code described in that chart.

Checking if the direction is valid is more complicated. You need to think about which inputs you want to allow. For this example, a numerical input (0, 1, 2) and the input with characters (x, X, y, Y, z, Z) should be allowed. Furthermore, this script should be applicable for 2D and 3D. Specifying a Z-direction for a 2D simulation doesn't make sense, therefore this has to be captured.

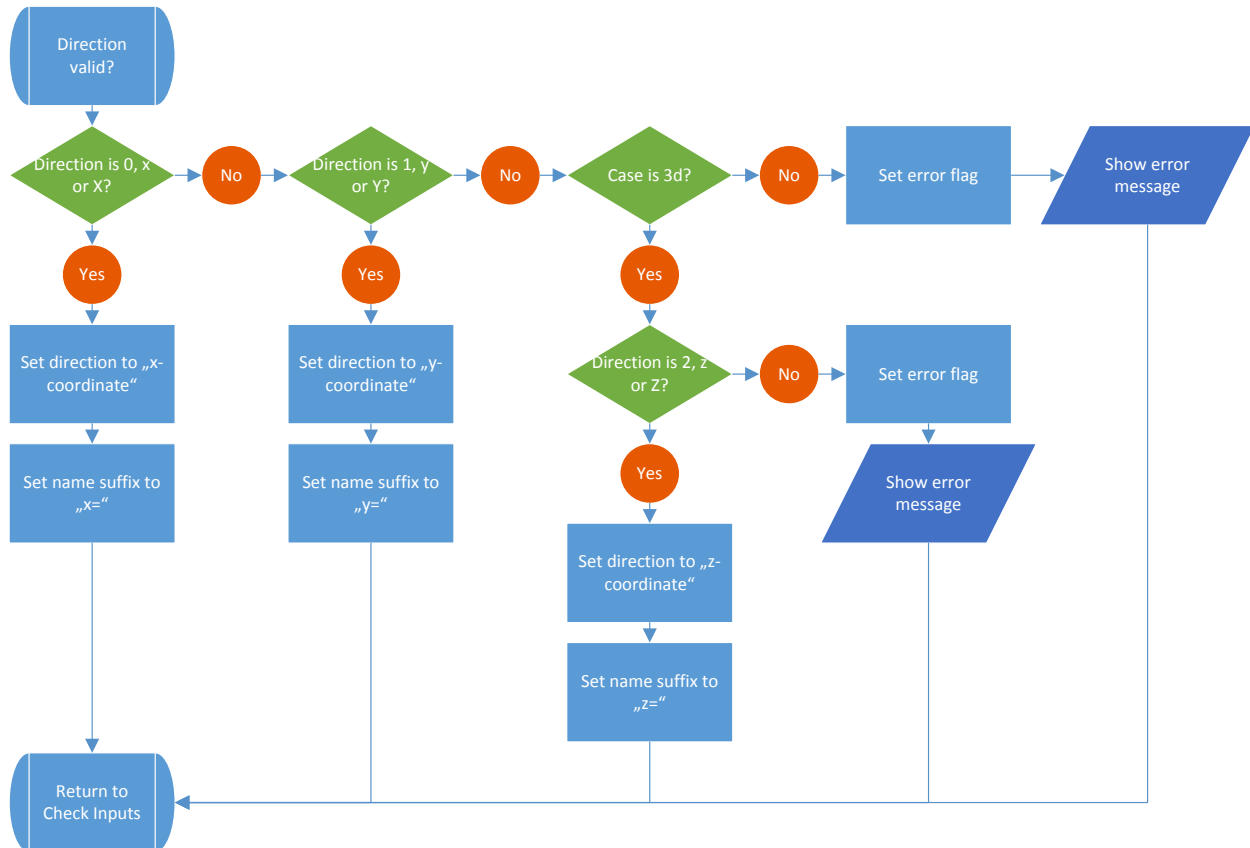


Figure 5: Flow chart for the procedure "direction valid"

The operations in "set direction to..." and "set name suffix to..." show the string that you need to implement later. The text command requires this exact string to create the iso-surfaces along one of the three Cartesian axes.

The last arguments are optional which requires additional effort to deal with it. You can rely on a specific order or you can think about a logic to deal with a variable order. For this example, you can use a mixed approach:

- Min and max coordinates are numbers that always come in a pair.
- The order of min and max should be irrelevant.
- The name pattern is a string that can stand alone or together with two Booleans.

Before checking the arguments, you can check the number of arguments:

- No optional arguments: Use default values for name pattern and min/max values.
- One optional argument: Can only be the name.
- Two optional arguments: Can only be min/max.
- Three optional arguments: Can be min/max + name pattern or only the name pattern. The type of the first argument specifies which one it is.
- Four optional arguments: Invalid.
- Five optional arguments: Either min/max comes first or the name pattern (string + 2 Booleans) comes first.
- More than five optional arguments: Invalid

The flow chart for these few points looks very busy. You don't have to go through it right now but the chart will help during the development of the code.

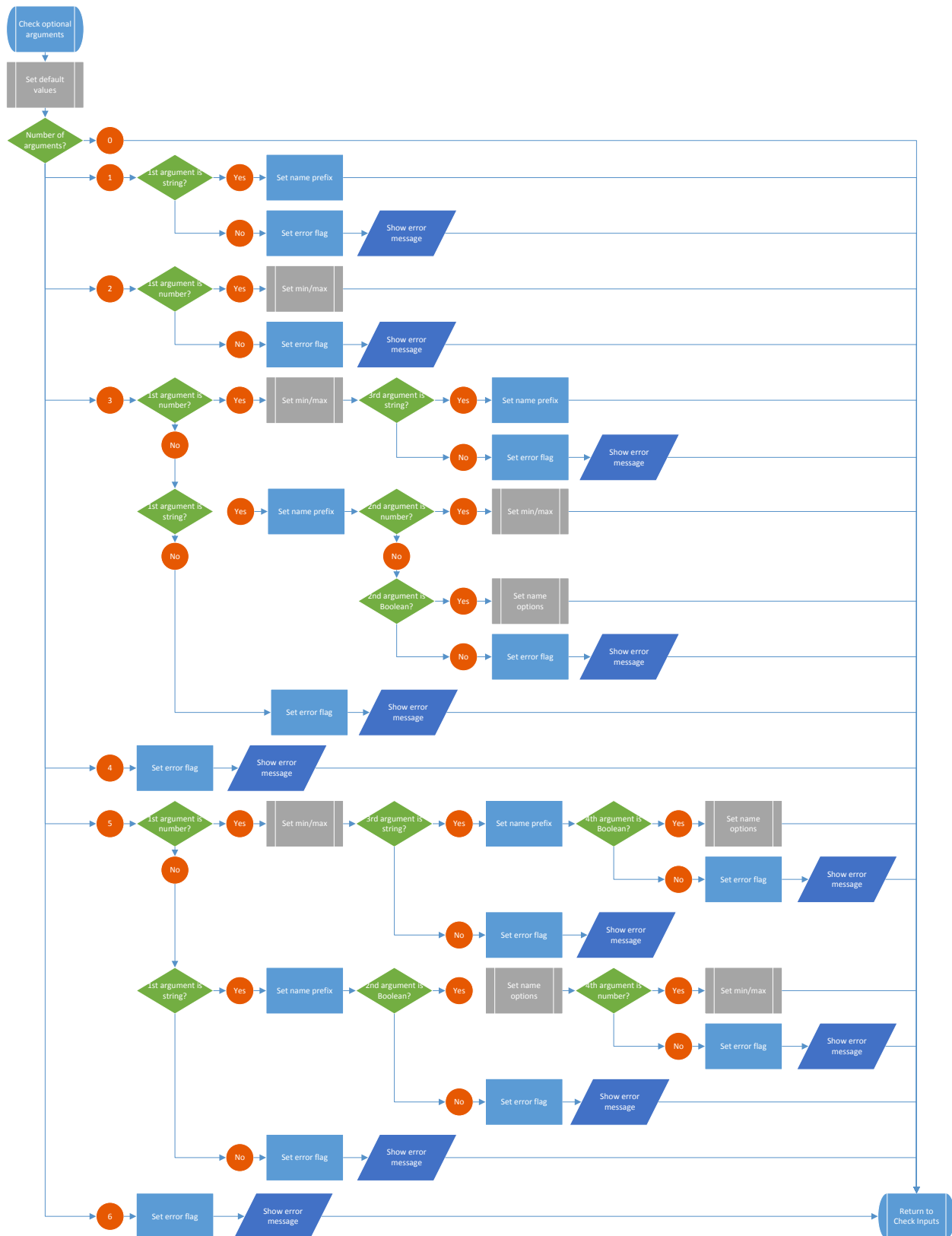


Figure 6: Flow chart for the procedure "check optional arguments"

There are a couple of sub procedures in the flow chart. To avoid forgetting something you can create charts for these right away. But it would also be possible to specify the details later.

Default values are required for start and end coordinates, name prefix and the Booleans for the two name suffixes. The name prefix could be set to a default value at the beginning but start and end coordinates require the selection of the direction, first. Note that there should also be a default value for the direction in case the input is invalid.

Each step in the following chart probably requires multiple lines of code.

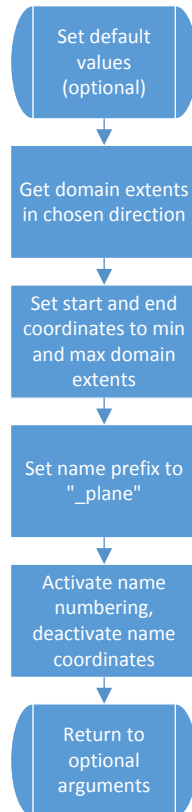


Figure 7: Flow chart for the procedure "set default values"

The sub-procedure for start and end coordinates requires more steps. It has to check:

- Is the second input valid? If not, abort.
- Are both inputs equal? If yes, use default values.
- Which of the two inputs is larger? Store values in new variables.
- Is one of both inputs outside of the domain? If outside, print warning and replace with default.

It's not necessary to check the first input again because this is done before this procedure is called.

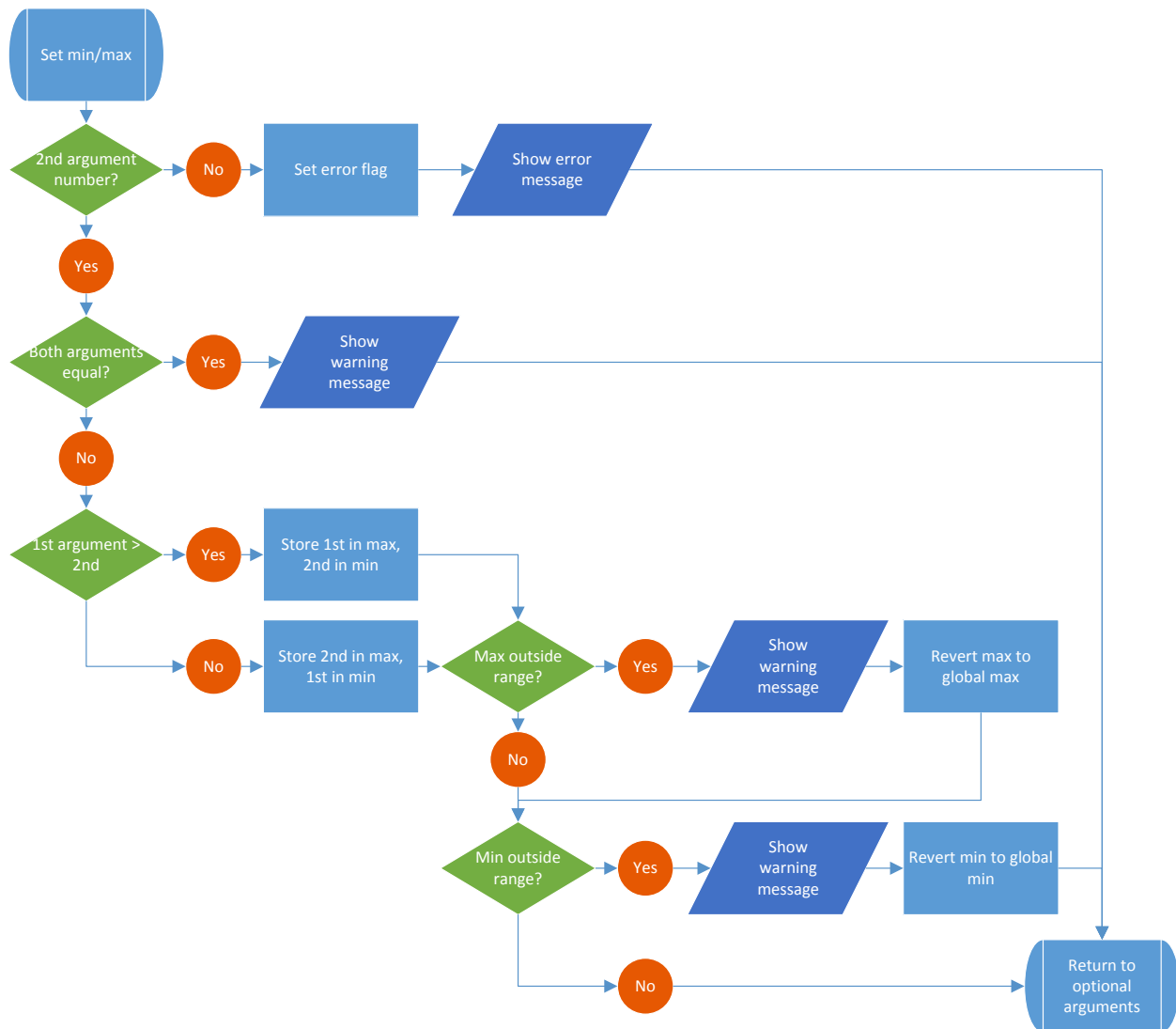


Figure 8: Flow chart for the procedure "set min max"

The last sub-procedure sets the options of the name of the planes. Like before, it has to check if the second argument is also a Boolean. If it's not, the script should abort.

If it is you should check if both are false. The names of the surfaces have to be different, therefore it's required that at least one of the two is true.

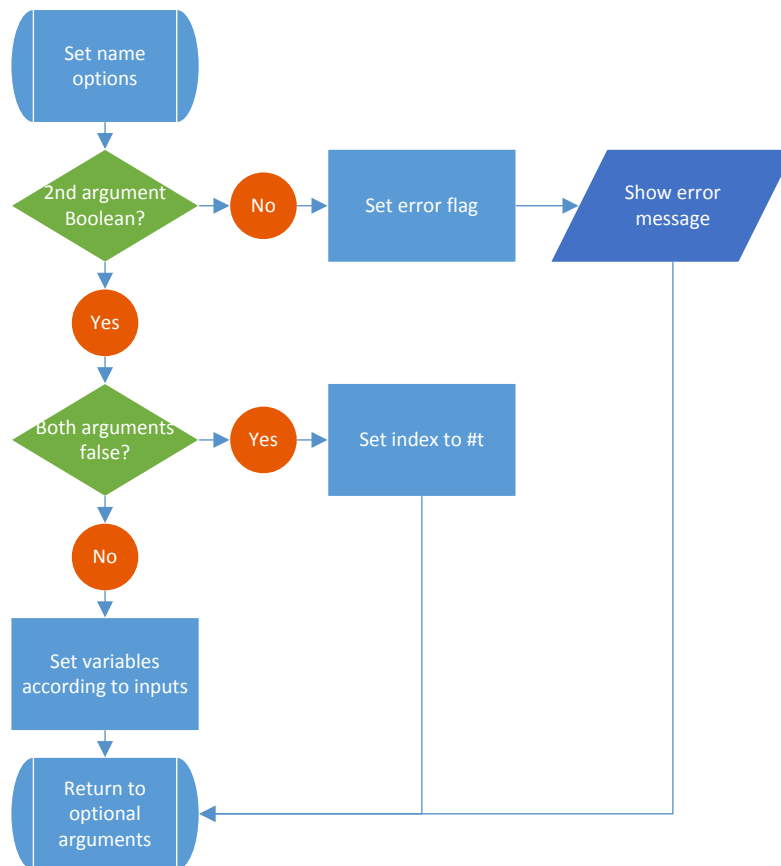


Figure 9: Flow chart for the procedure "set name options"

The final chart for the initial default values is still missing but it has to wait a little longer until the main functionality of the script is done.

Let's think about how these inputs can be turned into iso-surfaces.

To distribute the surfaces, you can use a loop. Each new surface needs an additional offset either from the previous position or from the starting position. This distance can be calculated outside of the loop.

Inside the loop you need to:

- Update the name of the surface
- Check if the name exists already. If it does, either delete the existing surface and replace it or abort the script.
- Create the iso-surface at the correct position

Fluent does not allow that a Scheme procedure takes additional user input while a procedure is executing. Therefore, you need to decide if it is more convenient for you to abort the loop or to replace the existing surface. For this example, existing surfaces will be replaced.

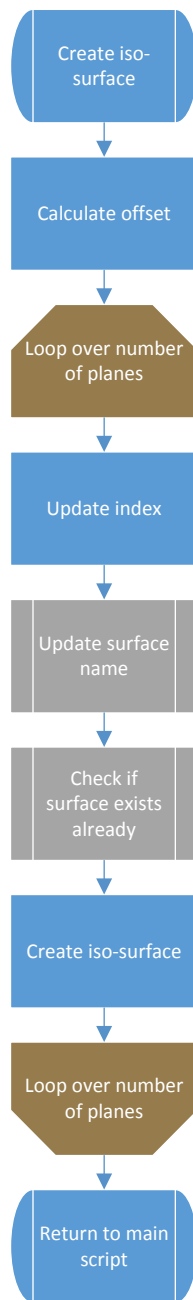


Figure 10: Flow chart for the procedure "create iso surface"

The name of the surfaces consists of three parts: prefix + index + coordinates. The prefix is always there. Index and coordinates are optional but at least one of them should be there. Since this was checked before, no additional check is required here.

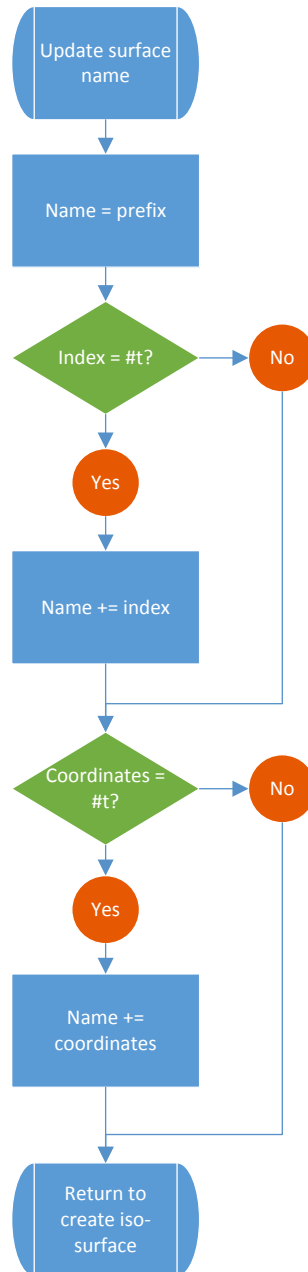


Figure 11: Flow chart for the procedure "update surface name"

Adding the index will take a few lines of code. The sorting algorithm in Fluent places the surface "x-10" before "x-2". To avoid that it's best to add leading zeros. This does not have to be put into a flow chart, though.

The last part of the code is to check if a surface exists already. This is trivial from a flow chart perspective.

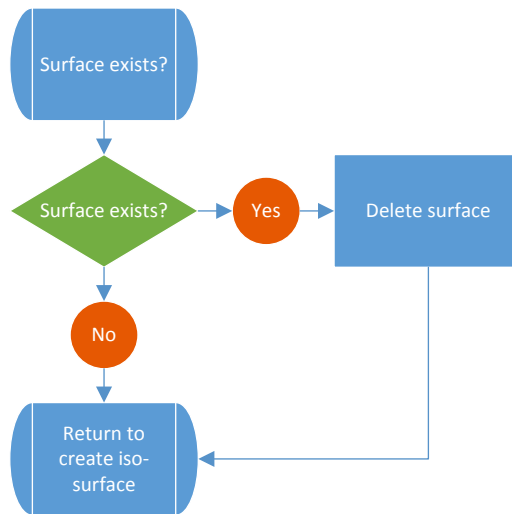


Figure 12: Flow chart for the procedure "surface exists"

Now that everything is known, you can add the final piece that's still missing. No additional default values popped up during the last flow charts, which means you just need to define the following variables at the beginning:

- Default direction = 0
- Error flag = #f

It's not necessary to create a flow chart for that, especially since you can add these in the very first step during coding.

Coding

Now it's time to get into coding. Start with the name, all arguments and a let environment that contains the default variables that you know of already. Use describing names for the variables. The prefix "ma" stands for "my argument", "mv" for "my variable", "mp" for "my procedure". The structure is given by Figure 1.

```

001 (define (post-iso-n ma-number ma-direction . args)
002   (let ((mv-direction 0) (mv-err-flag #f))
003     ; Data valid?
004     ; Number of surfaces valid?
005     ; Direction valid?
006     ; Optional arguments valid?
007     ; Create surfaces
008     (if (eqv? mv-err-flag #f)
009       (begin
010         (display "No errors encountered. Start with surface creation.\n")
011       )
012       (display "Abort script due to errors. See messages above for
013         details.\n")
014     )
015   )
  
```

Code fragment 28: Basic structure of the procedure "post-iso-n" (2042682_Example4_01.scm)

Start testing right away. When calling the procedure with (post-iso-n 0 0) it should show the message in line 10. However, providing less than these two arguments doesn't result in an error. This should not be a problem once all the checks are implemented.

```
> (post-iso-n 0 0)
No errors encountered. Start with surface creation.

> (post-iso-n)
No errors encountered. Start with surface creation.
```

Note: This example code is much longer than the previous examples. Therefore, not the complete listing is shown each time. If you follow along, pay attention to the line numbers. If a new line is added between line 5 and 6 it gets the number 6 and is marked purple. The old line 6 moves down, accordingly. Most listings will have enough code around the changed parts that you should be able to understand where it goes.

The next code snippet is the check if the data is valid which is outlined in Figure 2 and Figure 3. It's not necessary to move it into its own procedure because it is used only once.

003	; Data valid?
004	(if (not (data-valid?))
005	(begin
006	(set! mv-err-flag #t)
007	(display "Error! Data not valid. Please initialize the case.\n")
008)
009)
010	; Number of surfaces valid?

Code fragment 29: Check if the data is valid (2042682_Example4_02.scm)

The negation of the return value is required because within an if-statement Fluent expects some code for the case #t but the case #f is optional and can be left blank.

Again, test it without a loaded case, without initialization and with initialization.

The next snippet is the check for the first argument as outlined in Figure 4. Again, this is only required once, therefore no procedure is required.

```

002 (let ((mv-direction 0) (mv-err-flag #f) (max-surfaces 10001))
...
010 ; Number of surfaces valid?
011 (if (integer? ma-number)
012     (if (not (and (> ma-number 0) (< ma-number max-surfaces)))
013         (begin
014             (set! mv-err-flag #t)
015             (display (format #f "Error! First argument outside range. Enter a
value between 0 and ~a (excluded).\n" max-surfaces))
016         )
017     )
018     (begin
019         (set! mv-err-flag #t)
020         (display (format #f "Error! First argument is not an integer. Enter
a value between 0 and ~a (excluded).\n" max-surfaces))
021     )
022 )
023 ; Direction valid?

```

Code fragment 30: Implementation of a simple sanity check of the first mandatory argument (2042682_Example4_03.scm)

Compared to the flow chart there are two modifications. A new variable is introduced for the maximum number of surfaces. It is more convenient to have this number available at the top in case it's necessary to adjust it.

The second modification is another negation in line 12. The rest is just the flow chart for "Number of surfaces valid?" (Figure 3) in Scheme code.

Again, test the code with different arguments. Now Fluent complains if no arguments are given. But the error message is not very descriptive.

```

> (post-iso-n 1.1)
Error! First argument is not an integer. Enter a value between 0 and 10001 (excluded).
Abort script due to errors. See messages above for details.

> (post-iso-n 1)
No errors encountered. Start with surface creation.

> (post-iso-n 0)
Error! First argument outside range. Enter a value between 0 and 10001 (excluded).
Abort script due to errors. See messages above for details.

> (post-iso-n)

Error: eval: unbound variable
Error Object: ma-number

> (post-iso-n "a")
Error! First argument is not an integer. Enter a value between 0 and 10001 (excluded).
Abort script due to errors. See messages above for details.

```

Although it is not in the flow charts, you should add a fix to this problem regarding insufficient number of arguments. In principle you can live with that error as long as your script is not called from another script. If you do such a modification during coding, make sure to update your charts, too.

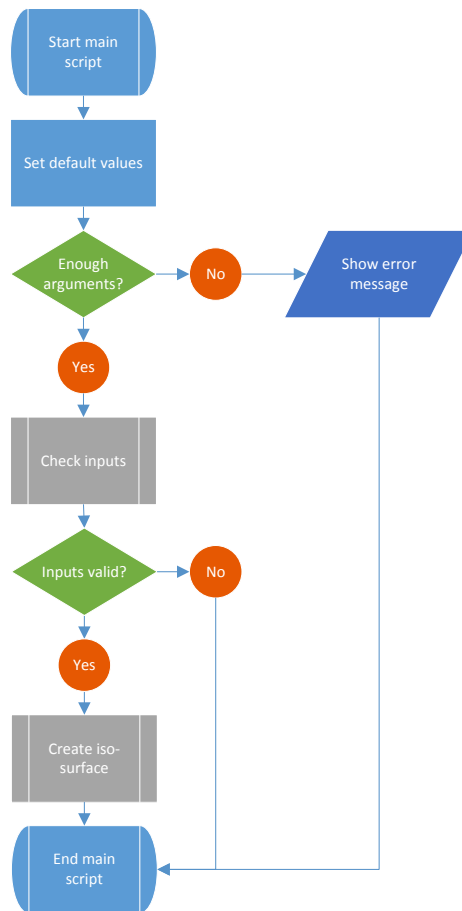


Figure 13: Update of the flow chart for the main script (Figure 1)

Since the mandatory arguments are not included in the list of arguments (args) it is not possible to check how many are specified. But Fluent avoids defining a variable when no value is specified. Therefore, you can check if the mandatory variables specified as arguments are known with the procedure `symbol-bound?`. If you use a consistent indentation almost all lines have to be indented by one level.

```

001 (define (post-iso-n ma-number ma-direction . args)
002   (let ((mv-direction 0) (mv-err-flag #f) (max-surfaces 10001))
003     ; Number of arguments valid?
004     (if (not (and (symbol-bound? 'ma-number (the-environment)) (symbol-
bound? 'ma-direction (the-environment))))
005       (display "Error! Not enough arguments specified. At least two
arguments are required. See (post-iso-n-docu) for details.\n")
006       (begin
007         ; Data valid?
008         (if (not (data-valid?))
009           (begin
010             (set! mv-err-flag #t)
011             (display "Error! Data not valid. Please initialize the
case.\n")
012           )
013         )
014         ; Number of surfaces valid?
015         (if (integer? ma-number)
016           (if (not (and (> ma-number 0) (< ma-number max-surfaces)))
017             (begin
018               (set! mv-err-flag #t)
019               (display (format #f "Error! First argument outside range.
Enter a value between 0 and ~a (excluded).\n" max-surfaces))
020             )
021           )
022           (begin
023             (set! mv-err-flag #t)
024             (display (format #f "Error! First argument is not an integer.
Enter a value between 0 and ~a (excluded).\n" max-surfaces))
025           )
026         )
027         ; Direction valid?
028         ; Optional arguments valid?
029         ; Create surfaces
030         (if (eqv? mv-err-flag #f)
031           (begin
032             (display "No errors encountered. Start with surface
creation.\n")
033           )
034           (display "Abort script due to errors. See messages above for
details.\n")
035         )
036       )
037     )
038   )
039 )

```

Code fragment 31: Improving robustness by checking the number of mandatory arguments
(2042682_Example4_04.scm)

```

> (post-iso-n "a")
Error! Not enough arguments specified. At least two arguments are required. See (post-iso-n-docu) for details.

> (post-iso-n)
Error! Not enough arguments specified. At least two arguments are required. See (post-iso-n-docu) for details.

> (post-iso-n 0)
Error! Not enough arguments specified. At least two arguments are required. See (post-iso-n-docu) for details.

> (post-iso-n 0 0)
Error! First argument outside range. Enter a value between 0 and 10001 (excluded).
Abort script due to errors. See messages above for details.

> (post-iso-n 1 0)
No errors encountered. Start with surface creation.

```

The next input is the direction. The code is outlined in Figure 5. This requires the introduction of two new variables in line 2. To improve readability, they are moved into line 3.

```

001 (define (post-iso-n ma-number ma-direction . args)
002   (let ((mv-direction 0) (mv-err-flag #f) (max-surfaces 10001)
003         (mv-direction-iso "x-coordinate") (mv-direction-name "x="))
...
028   ; Direction valid?
029   (if (or (and (number? ma-direction) (eqv? ma-direction 0))
030         (and (string? ma-direction) (string-ci=? ma-direction "x")))
031       )
032   (begin
033     (set! mv-direction-iso "x-coordinate")
034     (set! mv-direction-name "x=")
035     (set! mv-direction 0)
036   )
037   (if (or (and (number? ma-direction) (eqv? ma-direction 1))
038         (and (string? ma-direction) (string-ci=? ma-direction "y")))
039       )
040   (begin
041     (set! mv-direction-iso "y-coordinate")
042     (set! mv-direction-name "y=")
043     (set! mv-direction 1)
044   )
045   (if (rp-3d?)
046       (if (or (and (number? ma-direction) (eqv? ma-direction 2))
047             (and (string? ma-direction) (string-ci=? ma-direction
048               "z")))
049           )
049       (begin
050         (set! mv-direction-iso "z-coordinate")
051         (set! mv-direction-name "z=")
052         (set! mv-direction 2)
053       )
054       (begin
055         (set! mv-err-flag #t)
056         (display "Error! Direction not recognized for 3D case.
057           Use 0 (x), 1 (y) or 2 (z) as second argument.\n")
058       )

```

```

059         (begin
060             (set! mv-err-flag #t)
061             (display "Error! Direction not recognized for 2D case. Use
0 (x) or 1 (y) as second argument.\n")
062         )
063     )
064 )
065 )
066 (display (format #f "Direction: ~a --- plane: ~a --- name: ~a\n"
mv-direction mv-direction-iso mv-direction-name))
067 ; Optional arguments valid?

```

Code fragment 32: Check the direction and set usable values for the different possible inputs (2042682_Example4_05.scm)

As usual, after the code is implemented, test to call the procedure with different arguments to see if it shows the expected behavior.

```

> (post-iso-n 1 "z")
Direction: 2 --- plane: z-coordinate --- name: z=
No errors encountered. Start with surface creation.

> (post-iso-n 1 "Y")
Direction: 1 --- plane: y-coordinate --- name: y=
No errors encountered. Start with surface creation.

> (post-iso-n 1 2)
Direction: 2 --- plane: z-coordinate --- name: z=
No errors encountered. Start with surface creation.

> (post-iso-n 1 3)
Error! Direction not recognized for 3D case. Use 0 (x), 1 (y) or 2 (z) as second argument.
Direction: 0 --- plane: x-coordinate --- name: x=
Abort script due to errors. See messages above for details.

> (post-iso-n 1 0)
Direction: 0 --- plane: x-coordinate --- name: x=
No errors encountered. Start with surface creation.

```

Note that line 66 is only for debugging and testing purposes and will be commented out in the next step.

The three if-statements in lines 29/30, 37/38 and 46/47 might look a bit confusing if you're not used to nested logical tests. It definitely looks more complicated than in the flow chart (Figure 5).

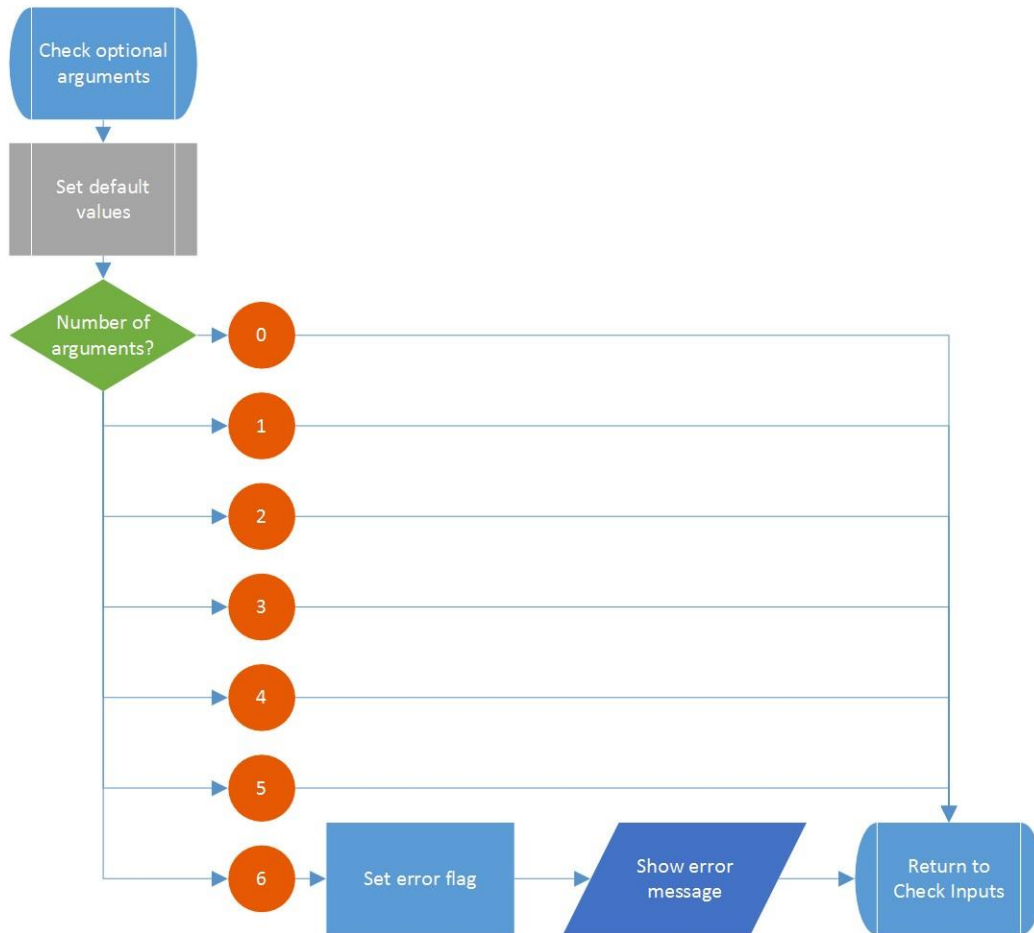
Such nested logical statements can be easier to read when going from back to front. The last test in line 30 (string-ci=?) checks if the direction is either "x" or "X". string-ci=? works only for strings. Therefore, it is important to exclude all other possible variable types before it is used which is done with the string? statement before. Both are combined with an and statement. Fortunately, if the first argument of an and statement is false, the second is no longer checked which makes this part of the code robust.

Similarly, in line 29, the last test (eqv?) checks if the direction is exactly 0 and the one before (number?) checks if it is a number. Again, the order is important and both conditions need to be true that this part evaluates to true.

But only one of the two and statements needs to return true for the whole if statement to become true because of the or statement that connects them both. If the check in line 29 evaluates to true already, line 30 will not be checked at all because of the or-statement.

The same is done for Y and Z directions but only if the previous check failed already.

The next large block is for the optional arguments. This time it's necessary to implement additional procedures because "Set min/max" and "Set name options" are called from different locations. But first, start with the implementation of the default values and the skeleton for the cond structure as outlined in Figure 6. For now this chart can be simplified to only implement the structure.



Code fragment 33: Simplified structure for checking the optional arguments (see also Figure 6)

```

002  (let ((mv-direction 0) (mv-err-flag #f) (max-surfaces 10001)
003      (mv-direction-iso "x-coordinate") (mv-direction-name "x=")
004      (mv-name-prefix "_plane") (mv-name-index #t) (mv-name-position #f)
(mv-name)
005      (mv-min) (mv-max) (mv-glb-min) (mv-glb-max)
006      (mv-num-args 0))
...
069  ; (display (format #f "Direction: ~a --- plane: ~a --- name: ~a\n"
mv-direction mv-direction-iso mv-direction-name))
070      ; Optional arguments valid?
071      ; Set defaults
072      (set! mv-glb-min (list-ref (client-inquire-domain-extents) 0))
073      (set! mv-min mv-glb-min)
074      (set! mv-glb-max (list-ref (client-inquire-domain-extents) 1))
075      (set! mv-max mv-glb-max)
076      (set! mv-num-args (length args))
077      (cond
078          ((eqv? mv-num-args 1) (display "1 optional argument.\n"))
079          ((eqv? mv-num-args 2) (display "2 optional arguments.\n"))
080          ((eqv? mv-num-args 3) (display "3 optional arguments.\n"))
081          ((eqv? mv-num-args 4) (display "4 optional arguments.\n"))
082          ((eqv? mv-num-args 5) (display "5 optional arguments.\n"))
083          ((> mv-num-args 5) (display "More than 5 optional arguments. >
Error!\n")))
084      )
085      ; Create surfaces

```

Code fragment 34: Basic structure to check the number and content of the optional arguments (2042682_Example4_06.scm)

```

> (post-iso-n 3 0 1)
1 optional argument.
No errors encountered. Start with surface creation.

> (post-iso-n 3 0 1 2)
2 optional arguments.
No errors encountered. Start with surface creation.

> (post-iso-n 3 0 1 2 3)
3 optional arguments.
No errors encountered. Start with surface creation.

> (post-iso-n 3 0 1 2 3 4)
4 optional arguments.
No errors encountered. Start with surface creation.

> (post-iso-n 3 0 1 2 3 4 5)
5 optional arguments.
No errors encountered. Start with surface creation.

> (post-iso-n 3 0 1 2 3 4 5 6)
More than 5 optional arguments. > Error!
No errors encountered. Start with surface creation.

```

Note that there is no condition for the case "0 arguments" because this means using default values.

Again, there was a change in the structure of the program that should be considered in the charts. Some variables got their default values during declaration in lines 2 to 6 which simplifies the definition of the default values for optional arguments (Figure 7 replaced by Figure 14).

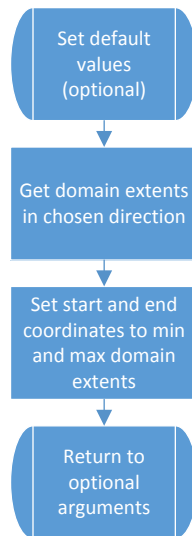


Figure 14: Procedure "set default values" which replaces the outline of Figure 7

Now you can start implementing each case of the cond statement as outlined in Figure 6.

In case of a single optional argument, check if it's a string and set the name prefix accordingly.

```

077      (cond
078      ((eqv? mv-num-args 1) (begin
079      ;      (display "1 optional argument.\n")
080      (if (string? (list-ref args 0))
081      (set! mv-name-prefix (list-ref args 0))
082      (begin
083      (set! mv-err-flag #t)
084      (display "Error! Expected a string for the name pattern as
third argument. See (post-iso-n-docu) for details.\n")))
085      )))
086      ((eqv? mv-num-args 2) (display "2 optional arguments.\n"))

```

Code fragment 35: Settings for a single optional argument (2042682_Example4_07.scm)

```

> (post-iso-n 3 0 1)
Error! Expected a string for the name pattern as third argument. See (post-iso-n-docu) for details.
Abort script due to errors. See messages above for details.

> (post-iso-n 3 0 "1")
No errors encountered. Start with surface creation.

```

Although this code does exactly what it should, it can be a problem still. Fluent has some naming restrictions for surfaces. The first character has to be letter. Some more characters are allowed, like the underscore for example. Implementing this check requires more code and since it should be used at several locations it's best to put it into a new procedure.

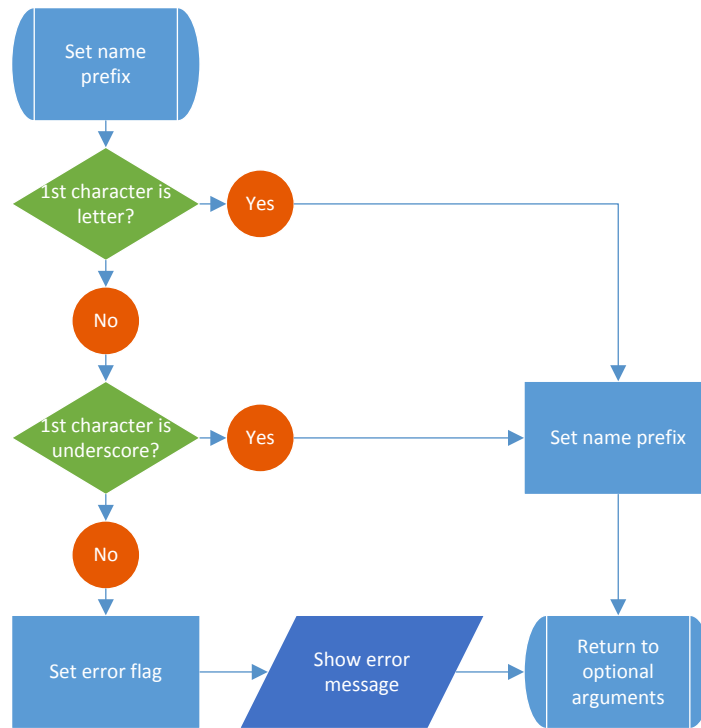


Figure 15: Flow chart to avoid invalid characters for the plane names

```

002  (let* ((mv-direction 0) (mv-err-flag #f) (max-surfaces 10001)
003         (mv-direction-iso "x-coordinate") (mv-direction-name "x=")
004         (mv-name-prefix "_plane") (mv-name-index #t) (mv-name-position #f)
(mv-name)
005         (mv-min) (mv-max) (mv-glb-min) (mv-glb-max)
006         (mv-num-args 0)
007         (mp-name-prefix (lambda (mlv-name-prefix)
008             (let ((mlv-char))
009                 (set! mlv-char (string-ref mlv-name-prefix 0))
010                 (if (or (char-alphabetic? mlv-char) (eqv? mlv-char #\_))
011                     (set! mv-name-prefix mlv-name-prefix)
012                     (begin
013                         (set! mv-err-flag #t)
014                         (display "Error! Name pattern invalid. Start with a letter
or underscore.\n")
015                     ))
016             )))
...
087  (cond
088      ((eqv? mv-num-args 1) (begin
089          (display "1 optional argument.\n")
090          (if (string? (list-ref args 0))
091              (mp-name-prefix (list-ref args 0))
092              (begin
093                  (set! mv-err-flag #t)
094                  (display "Error! Expected a string for the name pattern as
third argument. See (post-iso-n-docu) for details.\n"))
095              )))
096      ((eqv? mv-num-args 2) (display "2 optional arguments.\n")))

```

Code fragment 36: Make sure the surface name is valid (2042682_Example4_08.scm)

Note that the `let` statement has to be exchanged with `let*` in line 2 that the procedure can access all the variables.

The next condition is similar but this time it's about setting the start and end coordinates. Another procedure is required for this which is outlined in Figure 8.

```

015         ))
016     )))
017     (mp-minmax (lambda (mlv-start mlv-end mlv-index)
018         (if (not (number? mlv-end))
019             (begin
020                 (set! mv-err-flag #t)
021                 (display (format #f "Error! Expected end coordinates as
number for argument ~a.\n" mlv-index))
022             )
023             (if (eqv? mlv-start mlv-end)
024                 (display (format #f "Warning! Start and end coordinates are
identical. Using global values ~a .. ~a instead.\n" mv-min mv-max))
025                 (begin
026                     (if (> mlv-start mlv-end)
027                         (begin
028                             (set! mv-min mlv-end)
029                             (set! mv-max mlv-start)
030                         )
031                         (begin
032                             (set! mv-max mlv-end)
033                             (set! mv-min mlv-start)
034                         ))
035                     (if (> mv-max mv-glb-max)
036                         (begin
037                             (set! mv-max mv-glb-max)
038                             (display (format #f "Warning. End coordinates outside
of domain. Using ~a instead.\n" mv-max))
039                         ))
040                     (if (< mv-min mv-glb-min)
041                         (begin
042                             (set! mv-min mv-glb-min)
043                             (display (format #f "Warning. Start coordinates
outside of domain. Using ~a instead.\n" mv-min))
044                         ))
045                     ))
046                 ))
047             ))
048         ))
049     )
050 ; Number of arguments valid?

```

Code fragment 37: Implementation of the sub-procedure "set min max" (2042682_Example4_09.scm)

This is the exact implementation of the chart "Set min/max". By itself it doesn't do much. It has to be called like it is outlined in Figure 6 for the case of two optional arguments.

```

128         )))
129         ((eqv? mv-num-args 2) (begin
130 ;         (display "2 optional arguments.\n")
131         (if (number? (list-ref args 0))
132             (mp-minmax (list-ref args 0) (list-ref args 1) (+ 2 mv-num-
args))
133             (begin
134                 (set! mv-err-flag #t)
135                 (display "Error! Expected a number for the start
coordinates as third argument. See (post-iso-n-docu) for details.\n")))
136             )))
137         ((eqv? mv-num-args 3) (display "3 optional arguments.\n"))
138         ((eqv? mv-num-args 4) (display "4 optional arguments.\n"))
139         ((eqv? mv-num-args 5) (display "5 optional arguments.\n"))
140         ((> mv-num-args 5) (display "More than 5 optional arguments. >
Error!\n")))
141     )
142     (display (format #f "Direction: ~a\n" mv-direction))
143     (display (format #f "Start coordinates: ~a\n" mv-min))
144     (display (format #f "End coordinates: ~a\n" mv-max))
145     (display (format #f "Name prefix: ~a\n" mv-name-prefix))
146     (display (format #f "Index suffix: ~a\n" mv-name-index))
147     (display (format #f "Position suffix: ~a\n" mv-name-position))
148 ; Create surfaces

```

Code fragment 38: Settings for two optional arguments (2042682_Example4_10.scm)

In addition to what is described in the chart “Check optional arguments” (Figure 6) lines 142 to 147 are added for debugging and testing purposes. They can be removed later.

Now it's time to test the code again. Use all possible inputs for two optional arguments and check if the output is as expected:

- (post-iso-n 1 0 0 0)
- (post-iso-n 1 0 0 "0")
- (post-iso-n 1 0 "0" "0")
- (post-iso-n 1 0 "0" 0)
- (post-iso-n 1 0 0.01 0)
- (post-iso-n 1 0 0 0.01)
- (post-iso-n 1 0 0 0100)
- (post-iso-n 1 0 100 0)
- (post-iso-n 1 0 100 100)

```

> (post-iso-n 1 0 0.04 0.01)
Warning. End coordinates outside of domain. Using 0.02 instead.
Direction: 0
Start coordinates: 0.01
End coordinates: 0.02
Name prefix: _plane
Index suffix: #t
Position suffix: #f
No errors encountered. Start with surface creation.

> (post-iso-n 1 0 0.04 -0.03)
Warning. End coordinates outside of domain. Using 0.02 instead.
Warning. Start coordinates outside of domain. Using -0.02 instead.
Direction: 0
Start coordinates: -0.02
End coordinates: 0.02
Name prefix: _plane
Index suffix: #t
Position suffix: #f
No errors encountered. Start with surface creation.

```

You need a last sub-procedure to set the name flags that is called when the second or fourth optional argument is a Boolean (Figure 9). It checks whether the next argument is also a Boolean and it makes sure that at least one of the two is true.

```

048      ))
049      (mp-name-options (lambda (mlv-index-flag mlv-position-flag mlv-
index)
050          (if (not (boolean? mlv-position-flag))
051              (begin
052                  (set! mv-err-flag #t)
053                  (display (format #f "Error! Expected Boolean value for
argument ~a.\n" mlv-index))
054              )
055              (if (and (eqv? mlv-index-flag #f) (eqv? mlv-position-flag #f))
056                  (begin
057                      (set! mv-name-index #t)
058                      (display "Warning! Both name options are false. Setting
index to true.\n"))
059                  (begin
060                      (set! mv-name-index mlv-index-flag)
061                      (set! mv-name-position mlv-position-flag)
062                  ))
063          ))
064      ))
065  )
066      ; Number of arguments valid?

```

Code fragment 39: Check the optional arguments for the settings of naming options (2042682_Example4_11.scm)

The implementation of three optional arguments is as sketched in the chart.

```

152         )))
153         ((eqv? mv-num-args 3) (begin
154 ;         (display "3 optional arguments.\n")
155         (if (number? (list-ref args 0))
156         (begin
157         (mp-minmax (list-ref args 0) (list-ref args 1) 4)
158         (if (string? (list-ref args 2))
159         (mp-name-prefix (list-ref args 2))
160         (begin
161         (set! mv-err-flag #t)
162         (display "Error! Expected a string for the name pattern
as fifth argument. See (post-iso-n-docu for details.\n"))
163         ))
164         (begin
165         (if (string? (list-ref args 0))
166         (begin
167         (mp-name-prefix (list-ref args 0))
168         (if (number? (list-ref args 1))
169         (mp-minmax (list-ref args 1) (list-ref args 2) 5)
170         (if (boolean? (list-ref args 1))
171         (mp-name-options (list-ref args 1) (list-ref args
2) 5)
172         (begin
173         (set! mv-err-flag #t)
174         (display "Error! Expected number or Boolean as
fourth argument. See (post-iso-n-docu) for details.\n")
175         ))))
176         (begin
177         (set! mv-err-flag #t)
178         (display "Error! Expected number or string as third
argument. See (post-iso-n-docu) for details.\n")
179         ))
180         ))
181         ))
182         ((eqv? mv-num-args 4) (display "4 optional arguments.\n"))

```

Code fragment 40: Setting options for three optional arguments (2042682_Example4_12.scm)

As usual, test the script with the different arguments to see if all the outputs are correct:

- (post-iso-n 1 0 "name" 0.01 -0.01)
- (post-iso-n 1 0 "name" #f #f)
- (post-iso-n 1 0 "name" #t #f)
- (post-iso-n 1 0 "name" #f #t)
- (post-iso-n 1 0 "name" #t #t)
- (post-iso-n 1 0 "name" #t -0.01)
- (post-iso-n 1 0 0.01 -0.01 "name")
- (post-iso-n 1 0 0.01 -0.01 0)

```

> (post-iso-n 1 0 "_name" #f #t)
Direction: 0
Start coordinates: -0.02
End coordinates: 0.02
Name prefix: _name
Index suffix: #f
Position suffix: #t
No errors encountered. Start with surface creation.

```

There is no scenario with four optional arguments. Just output an error message for this case.

```

181         ))
182         ((eqv? mv-num-args 4) (begin
183             (set! mv-err-flag #t)
184             (display "Error! Incomplete number of arguments. See (post-iso-
n-docu) for details.\n")))
185         ((eqv? mv-num-args 5) (display "5 optional arguments.\n"))

```

Code fragment 41: Print an error for four optional arguments (2042682_Example4_13.scm)

The last block in this conditional statement is the largest. But essentially all the code is already there. You just need to combine it with a number of if-statements.

```

184         (display "Error! Incomplete number of arguments. See (post-iso-
n-docu) for details.\n")))
185         ((eqv? mv-num-args 5) (begin
186             ; (display "5 optional arguments.\n"))
187             (if (number? (list-ref args 0))
188                 (begin
189                     (mp-minmax (list-ref args 0) (list-ref args 1) 3)
190                     (if (string? (list-ref args 2))
191                         (begin
192                             (mp-name-prefix (list-ref args 2)
193                             (if (boolean? (list-ref args 3))
194                                 (mp-name-options (list-ref 3) (list-ref 4) 6)
195                                 (begin
196                                     (set! mv-err-flag #t)
197                                     (display "Error! Expected Boolean as sixth
argument. See (post-iso-n-docu) for details.\n")
198                                 ))))
199                         (begin
200                             (set! mv-err-flag #t)
201                             (display "Error! Expected string as fifth argument. See
(post-iso-n-docu) for details.\n")
202                             )))

```

```

203      (if (string? (list-ref args 0))
204      (begin
205      (mp-name-prefix (list-ref args 0))
206      (if (boolean? (list-ref args 1))
207      (begin
208      (mp-name-options (list-ref args 1) (list-ref args 2)
4)
209      (if (number? (list-ref args 3))
210      (mp-minmax (list-ref args 3) (list-ref args 4) 6)
211      (begin
212      (set! mv-err-flag #t)
213      (display "Error! Expected number as sixth
argument. See (post-iso-n-docu) for details.\n")
214      )))
215      (begin
216      (set! mv-err-flag #t)
217      (display "Error! Expected Boolean as fourth argument.
See (post-iso-n-docu) for details.\n")
218      )))
219      (begin
220      (set! mv-err-flag #t)
221      (display "Error! Expected string or number as third
argument. See (post-iso-n-docu) for details.\n")
222      )))))
223      ((> mv-num-args 5) (display "More than 5 optional arguments. >
Error!\n")))

```

Code fragment 42: Finish the implementation for five optional arguments (2042682_Example4_14.scm)

Of course, all of this has to be tested again:

- (post-iso-n 1 0 "name" #t #f -0.01 0.01)
- (post-iso-n 1 0 "name" 0 #f -0.01 0.01)
- (post-iso-n 1 0 "name" #t #t 0.01)
- (post-iso-n 1 0 -0.01 0.01 "name" #t #f)
- (post-iso-n 1 0 -0.01 0.01 0 #t #f)
- (post-iso-n 1 0 #f #t #f -0.01 0.01)

```

> (post-iso-n 1 0 "name" #t #f -0.01 0.01)
Direction: 0
Start coordinates: -0.01
End coordinates: 0.01
Name prefix: name
Index suffix: #t
Position suffix: #f
No errors encountered. Start with surface creation.

```

So far you have about 240 lines of code and all of it is just for usability and robustness of the script. The core functionality comes now.

Two more sub-procedures (Figure 11 and Figure 12) are required. For starters, you can just implement dummies to get the main code working.

```

006      (mv-num-args 0)
007      (mv-name) (mv-distance) (mv-position)
008      (mp-name-prefix (lambda (mlv-name-prefix)
...
065      ))
066      (mp-update-name (lambda (mlv-name-prefix mlv-index-flag mlv-
position-flag mlv-index mlv-position)
067      (display "Updating name\n")
068      ))
069      (mp-check-surface (lambda (mlv-name)
070      (display "Checking for existing surface\n")
071      ))
072      )
073      ; Number of arguments valid?
...
238      ; Create surfaces
239      (if (eqv? mv-err-flag #f)
240      (begin
241      ; (display "No errors encountered. Start with surface
creation.\n")
242      (set! mv-distance (/ (- mv-max mv-min) (+ ma-number 1)))
243      (do ((i 1 (+ i 1)))
244      ((> i ma-number) (display (format #f "\nCreated ~a iso-
surfaces.\n" ma-number)))
245      (begin
246      (set! mv-position (+ mv-min (* i mv-distance)))
247      (set! mv-name (mp-update-name mv-name-prefix mv-name-index
mv-name-position i mv-position))
248      (mp-check-surface mv-name)
249      (display (format #f "Create surface ~a at position ~a\n" i
mv-position))
250      )
251      )
252      )
253      (display "Abort script due to errors. See messages above for
details.\n")

```

Code fragment 43: Prepare the creation of iso-surfaces (2042682_Example4_15.scm)

Line 249 is just a placeholder for the text command. Because of the missing sub-procedures for Figure 10. The text command would not work right now, therefore this is the last sub-procedure to implement.

The calculation of the distance in line 242 assumes that the user wants to create the defined number of iso-surfaces between the specified min and max coordinates. No surface will be created exactly at the specified coordinates. Usually, boundaries or interiors exist at the coordinates the user specifies. Therefore, they are excluded here.

Several outputs (lines 244, 249, 67, 70) make it possible to test the current state of the script already.

```

> (post-iso-n 2 0 "name" #t #f -0.01 0.01)
Direction: 0
Start coordinates: -0.01
End coordinates: 0.01
Name prefix: name
Index suffix: #t
Position suffix: #f
Updating name
Checking for existing surface
Create surface 1 at position -0.0033333333
Updating name
Checking for existing surface
Create surface 2 at position 0.0033333333
Created 2 iso-surfaces.

```

Next comes the name of the planes as described in Figure 11. You can implement this according to the prepared chart. Adding the index requires a few more lines to add some leading zeros if the index is too small. This improves how the surfaces are available in the native Fluent panels.

```

066      (mp-update-name (lambda (mlv-name-prefix mlv-index-flag mlv-
067      position-flag mlv-index mlv-position)
068      ;      (display "Updating name\n")
069      (let ((mlv-name) (mlv-full-index))
070      (set! mlv-name mlv-name-prefix)
071      (if (eqv? mlv-index-flag #t)
072      (begin
073      (cond
074      ((< mlv-index 10) (set! mlv-full-index (format #f
075      "0000~a" mlv-index)))
076      ((< mlv-index 100) (set! mlv-full-index (format #f
077      "000~a" mlv-index)))
078      ((< mlv-index 1000) (set! mlv-full-index (format #f
079      "00~a" mlv-index)))
080      ((< mlv-index 10000) (set! mlv-full-index (format #f
081      "0~a" mlv-index)))
082      (else (set! mlv-full-index mlv-index)))
083      (set! mlv-name (format #f "~a-~a" mlv-name mlv-full-
084      index)))
085      )
086      )
087      (if (eqv? mlv-position-flag #t)
088      (set! mlv-name (format #f "~a-~a~a" mlv-name mv-direction-
089      name mlv-position))
090      )
091      (display (format #f "Iso-Surface ~a name: ~a\n" mlv-index mlv-
092      name))
093      mlv-name
094      )
095      ))
096      (mp-check-surface (lambda (mlv-name)

```

Code fragment 44: Complete the name for the iso-surfaces (2042682_Example4_16.scm)

Note that the variable that holds the name is the last statement in the procedure (line 85). This is required that the procedure returns the contents of that variable which allows you to reuse it afterwards.

Again, the display statements allow you to test the script without creating the surfaces yet.


```

> (post-iso-n 2 0 "name" #t #t -0.01 0.01)
Direction: 0
Start coordinates: -0.01
End coordinates: 0.01
Name prefix: name
Index suffix: #t
Position suffix: #t
Iso-Surface 1 name: name-00001-x=-0.0033333333
Checking for existing surface
Create surface 1 at position -0.0033333333
Iso-Surface 2 name: name-00002-x=0.0033333333
Checking for existing surface
Create surface 2 at position 0.0033333333
Created 2 iso-surfaces.

```

The last sub-procedure deletes the surfaces if they exist already. This can be done quickly but requires access to all surfaces that exist already and multiple data type conversions.

```

087         ))
088         (mp-check-surface (lambda (mlv-name)
089 ;           (display "Checking for existing surface\n")
090           (if (member (string->symbol mlv-name) (inquire-surface-names))
091             (begin
092               (delete-surfaces (list (surface-name->id (string->symbol
mlv-name))))))
093           (display (format #f "Warning! Replacing surface ~a.\n" mlv-
name))))))
094         ))
095       )
096       ; Number of arguments valid?

```

Code fragment 45: Check if a surface with the specified name exists already and delete it (2042682_Example4_17.scm)

To test this code, you can create a new surface manually either with a text command or by using the panels available in the Fluent Ribbon under Postprocessing > Surface > Create

```

> (post-iso-n 2 0 "name" #t #f -0.01 0.01)
Direction: 0
Start coordinates: -0.01
End coordinates: 0.01
Name prefix: name
Index suffix: #t
Position suffix: #f
Iso-Surface 1 name: name-00001
Warning! Replacing surface name-00001.
Create surface 1 at position -0.0033333333
Iso-Surface 2 name: name-00002
Create surface 2 at position 0.0033333333
Created 2 iso-surfaces.

```

Now that everything seems to work, you can replace line 272 with the correct text command to actually create the iso-surfaces.

```

271         (mp-check-surface mv-name)
272         (ti-menu-load-string (format #f "/surface/iso-surface ~a ~a
() () ~a ()\n" mv-direction-iso mv-name mv-position))
273     )

```

Code fragment 46: Create the iso surfaces (2042682_Example4_18.scm)

When you test this version of the script, make sure to display the iso-surfaces to confirm the position.

```

> (post-iso-n 2 0 "name" #t #f -0.01 0.01)
Direction: 0
Start coordinates: -0.01
End coordinates: 0.01
Name prefix: name
Index suffix: #t
Position suffix: #f
Iso-Surface 1 name: name-00001
/surface/iso-surface x-coordinate name-00001 () ()range [-0.02, 0.02]
-0.0033333333 ()

> Iso-Surface 2 name: name-00002
/surface/iso-surface x-coordinate name-00002 () ()range [-0.02, 0.02]
0.0033333333 ()

>
Created 2 iso-surfaces.

```

The output looks very busy and contains a lot of information that is not required, including the output of the text commands. This can be hidden easily using the procedure “quiet” from the main document. The code can also be cleaned up a bit by compressing all the of the closing parentheses and removing the lines with “display” that are no longer needed.

One final step is still missing, though. You might have wondered about the error messages already because they refer to something called (post-iso-n-docu). This procedure has to be written, too. It is good practice to provide some sort of documentation together with scripts if you target different users. Adding a compressed version of such a documentation right inside of Fluent can be very helpful. It contains just a bunch of display statements.

You can find all steps and the final version with the documentation procedure (2042682_Example4_19_final.scm) attached to solution 2042682. It is more convenient to look at the final Scheme file directly instead of checking a listing in this document that spans over many pages.

Summary

In this example you have seen a possible way to develop more complex scripts. Some preliminary testing might be required to understand the arguments of the text commands and how they behave under different conditions (e.g. with different models).

Good planning can help you write the code more easily. With charts you can see more easily which logic you need to implement that something works. Different philosophies exist about which charts are best and how much detail should be in these charts. This example shows only one possible way.

Making a script robust and user-friendly requires a lot of effort. In this example, only 5 to 10% of the code is required to perform the essential task of creating the iso-surfaces. The remaining code is required to capture invalid user input and allow flexibility of these inputs.

An alternative could be to provide a panel. Many incorrect inputs can be avoided in advance. But the code for a panel can be just as long as in this example.

References

- [1] ANSYS, Inc., "Solution 2042682: Introduction to ANSYS Fluent scripting - Part 2 - Accessing solver data," ANSYS, Inc., 2016.
- [2] ANSYS, Inc., "Solution 2042681: Introduction to ANSYS Fluent scripting - Part 1 - Introduction to Scheme," ANSYS, Inc., 2016.
- [3] ANSYS, Inc., "Solution 2038952: How to retrieve the case name in UDF?," ANSYS, Inc., 2015.
- [4] ANSYS, Inc., "Solution 2042751: How to remove all default interior zones from ANSYS Fluent post-processing?," ANSYS, Inc., 2016.
- [5] ANSYS, Inc., "Solution 2039572: How to change multiple ANSYS Fluent boundary conditions that follow a name pattern simultaneously?," ANSYS, Inc., 2016.
- [6] Ansys, Inc., "Solution 2042772: How to create many equally spaced (iso-) surfaces for ANSYS Fluent postprocessing?," ANSYS, Inc., 2016.
- [7] ANSYS, Inc., "Solution 2042683: Introduction to ANSYS Fluent scripting - Part 3 - Creating keyboard shortcuts, menu items and panels," ANSYS, Inc., 2016.
- [8] ANSYS, Inc., "Solution 912: How to create an iso-surface along an arbitrarily oriented plane?," ANSYS, Inc., 2015.
- [9] ANSYS, Inc., "Solution 1030: How to automatically create multiple point monitors?," ANSYS, Inc., 2015.