3. Functional Programming – Scheme

For this part of the comprehensive assignment, we will ask you to implement the RANSAC algorithm following the functional paradigm. However, this time we will give you a reduced point set and we ask you to simply find the main dominant plane in each point cloud (just one plane then, not the three most dominant ones as before. You will use the following files:

- `Point_Cloud_1_No_Road_Reduced.xyz`
- `Point_Cloud_2_No_Road_Reduced.xyz`
- `Point_Cloud_3_No_Road_Reduced.xyz`

And the following function that reads the point cloud in a file and creates a list of 3D points:

```
(define (readXYZ fileIn)
  (let ((sL (map (lambda s (string-split (car s)))
                          (cdr (file->lines fileIn)))))
    (map (lambda (L)
           (map (lambda (s)
                  (if (eqv? (string->number s) #f)
                      s
                      (string->number s))) L)) sL)))
```

Process to Follow

We give you here the steps to follow to implement the RANSAC plane identification algorithm in Scheme. You must create the functions listed here. Test each function with simple input to make sure they produce the correct result.

The initial function to call will be:

```
(planeRANSAC filename confidence percentage eps)
```

This function will return a pair made of the dominant plane equation and the number of points that supports it. The filename should be read in a `let` statement such that the variable `Ps` will contain the list of 3D points in the rest of the function.

The first step is to be able to pick 3 random points from a list of points. The following code can be used for that, assuming the points are in a list `Ps`:

```
(list-ref Ps (random (length Ps)))
```

You then need a function that computes a plane equation from 3 points. The equation is described by a list of the four parameters, from the equation ax+by+cz=d:

```
(plane P1 P2 P3)
```

```
'(a b c d)
```

The next step is to count the support of a plane. Create this function such that it returns the support count and the plane parameter in a pair.

```
(support plane points)
n.(a b c d)
```

Finally you need to repeat the random sampling K times to find the dominant plane (the plane with the best support).

```
(dominantPlane Ps k)
```

To determine the number of iterations required, you must define the function that computes this number based on certain confidence and a certain percentage of points as explained in the main problem description.

```
(ransacNumberOfIteration confidence percentage)
```

Submit your project in a zip file containing the scheme functions file and a text file containing the output obtained for each of the point cloud file. Make sure to include the function call with the values used to produce the shown result.

Make sure all your Scheme functions have a header describing what the function does, the input parameters and the output. Your function must adhere to the functional paradigm principles. In particular you must not use the functions terminating by ! (such as the set! function).

A simple example illustrating the iterative process

To help you in the design of your solution, we give you here a simple example of a function that iteratively search for a solution. The idea is to find the maximum value of a function by simply testing it with random values and keeping the one that produces the largest result. Obviously there are much better ways to solve this, but this is simply to illustrate an iterative algorithm that proceeds in a similar way to RANSAC. Examine this function and use it to inspire you in building your RANSAC solution.

```
; the function to maximize
(define (fct x) (+ (* (- 55 x) (- x 55)) 20))

; the function that finds the maximum of function fct by randomly
generating
; a number of random values indicated by parameter try
; the call could be as follows: (maxfct fct 100)
(define (maxfct fct try)
  (maxfct-helper fct try -1000))

; this is the helper function doing the work
(define (maxfct-helper fct try best)
(let ((x (random 100)))
(cond
    ((= 0 try) best)
    ((> (fct x) best) (maxfct-helper fct (- try 1) (fct x)))
    (else (maxfct-helper fct (- try 1) best)))))
```