We are going to write a design document for the C++ implementation of your favorite algorithm. There are no restrictions on your choice of the algorithm you want to implement, hopefully you are going to implement something close to your research.

Here are the few points you want to address in the document. Small toy examples are provided below to help you think.

Note, the following is one way of thinking about solving a complicated problem, its neither the only nor the best way. The summary is: Promise yourself the existence of low-level and high-level libraries, assuming their functionality, write the high-level pseudo-code and go back and implement both the low-level and the high-level libraries, adding along the way, more functionality to both sets of libraries as and when needed.

Although, you are not supposed to implement in this problem set, its good to keep the following rules of thumb in mind while writing pseudo-code:

- break down a piece of work into as many functions as necessary
- good indicators that you need a function instead of raw code are:
  - when indented code has already reached 3-4 levels of indentation deep
  - when you find yourself writing at least two “essentially similar” chunks of code in your program
- when you find yourself writing a function which is more than a 100 lines long, its time to break its functionality down into pieces and write some helper functions

Problem 1 The R function

Write the prottype of the R function you are going to ultimately use. Think about the set of parameters needed to control the algorithm and which of those could have default values. Here is a simple example

```r
generateMultivariateT (nobs, 
degreesOfFreedom, 
meanVec = c(0, 0), 
dispMatrix = diag(2))
```

In the following, we will be coming back to the above example many times. So, we briefly mention here the theory of generating a sample
from a multivariate \((m\text{-dimensional})\) \(t_{m,\nu}(\mu, \Sigma)\) distribution. Let \(Z \sim \text{Normal}_m(0, I_m)\) and \(X \sim \chi^2_v\). Also, let \(A\) be the Cholesky decomposition of the \(\Sigma\) matrix, i.e., we have \(\Sigma = AA^T\). Then \(Y := \mu + AZ\sqrt{\nu/X} \sim t_{m,\nu}(\mu, \Sigma)\)

Problem 2 The \(C\) function

Think about the local variables you need to create to implement the \(C\) backend for the above \(R\) function. Assuming you are going to use the \(\text{.Call}\) interface with all the arguments passed packed in a list you might need e.g. the following variables in \(C\):

```c
SEXP generateMultivariateT (SEXP argsList)
{
    int nobs, degreesOf Freedom, nProtected = 0;
    double *meanVecVals, *dispMatrixVals;
    SEXP retSampleMatrix;
    / *
    * Lots of stuff.
    */
    return retSampleMatrix;
}
```

Problem 3 The data structure(s) needed from the “bottom-up” view

This is one of the most important part of any problem. The choice of this should mainly be guided by the functionality you want from its associated interface / library (remember the vector library!).

So, for the above example, if we want to do all the necessary computation, namely, Cholesky decomposition, matrix-vector multiplication, vector-addition, using a vector and a matrix library of our own, we might want think about how to implement the matrix library. Note, we have already implemented a vector library. Along those lines we may propose the following data structure:

```c
typedef struct Matrix Matrix;
struct Matrix {
    int n_row;
    int n_col;
    double *vec_form_value;
};
```

For dealing with the above data structure we might want to propose the following interface:
extern Matrix *
matrix_new (long n_row, long n_col, Boolean is_in_vec_form);

extern void
matrix_free (Matrix **AA);

extern long
matrix_get_n_row (Matrix *AA);

extern long
matrix_get_n_col (Matrix *AA);

extern double
matrix_get_element (Matrix *AA, long row, long col);

extern void
matrix_set_element (Matrix *AA, long row, long col, double value);

extern void
matrix_set_col (Matrix *AA, long col, double *value);

extern void
matrix_set_row (Matrix *AA, long row, double *value);

extern void
matrix_transpose (Matrix *AA, Matrix *AA_transp);

extern void
matrix_scal_mat_mult (Matrix *AA, double aa, Matrix *BB);

extern void
matrix_mat_vec_mult (Matrix *AA, Vector *xx, Vector *yy);

extern void
matrix_mat_mat_add (Matrix *AA, Matrix *BB, Matrix *CC);

extern void
matrix_mat_mat_mult (Matrix *AA, Matrix *BB, Matrix *CC);

extern int
matrix_is_symmetric (Matrix *AA);

extern void
matrix_chol (Matrix *AA, Matrix *LL,
            int *is_mat_symmetric, int *is_mat_nnd);
Problem 4  The “top-down” view

Here we need to take a high-level view and ask ourselves what data structure and a corresponding library would be needed to do the actual sampling and its high level control in a neat-n-clean fashion.

So, continuing to think about our previous example first we might want to write a clean logic like this.

-- extract the parameters from the ‘argList’ argument and process them

-- store the parameters in a variable, call it opts, of type Options, say, a convenient data structure

-- use the variable opts to create a variable, call it sampler, of type Sampler, say, a convenient data structure

-- call a function on the sampler variable to do the sampling

The corresponding code might look like the following and may live within the SEXP generateMultivariateT (SEXP argList); function above.

/*
 * Lots of stuff.
 */
Options *opts;
Sampler *sampler;
double *samples;

opts = options_new( );
load_parameters_into_options (argsList, opts);
sampler = sampler_new (opts);
sampler_do_multivariate_T_sampling(sampler, opts, &samples);
/*
 * Lots of stuff.
 */

Problem 5  The data structure(s) needed from the “top-down” view

Here one needs to think about the details of the data structures used in the “top-down” view of the problem.

So, again continuing along the lines we have been pursuing so far, at this stage, we need to ask ourselves what the data structures Options and Sampler should look like and how to implement the functions used in the “top-down” view which use these to data structures.
Problem 6 Merging the “top-down” view and the “bottom-up” approach

Now its time to use the low-level library functions to write the high-level functions.

In our continuing example, we now should start thinking how we can for example use functions like matrix_chol, matrix_mat_vec_mult to implement say, sampler_do_multivariate_T_sampling. In the process, we might find that we would need some other low-level / high-level functions to make the code-flow transparent and clean. For example, how about a function like the following:

```c
extern void
vector_fill_with_standard_normals (Vector *zz);
```

Do you see how this convenience function might be useful?

—Happy (Pseudo-)Coding!—