1 UTILITIES

The library provides some utilities for writing Monte Carlo simulation programs. For some of them, such as command line option processing, there are more advanced, dedicated libraries out there. The library only provides some basic functionality that is sufficient for simple cases.

1.1 ALIGNED MEMORY ALLOCATION

The standard library class `std::allocator` is used by containers to allocate memory. It works fine in most cases. However, sometime it is desirable to allocate memory aligned by a certain boundary. The library provides the class template,

```
template <typename T, std::size_t Alignment = AlignmentTrait<T>::value,
    typename Memory = AlignedMemory>
class Allocator;
```

which conforms to the `std::allocator` interface. The address of the pointer returned by the `allocate` method will be a multiple of `Alignment`. The value of alignment has to be positive, larger than `sizeof(void *)`, and a power of two. Violating any of these conditions will result in compile-time error. The last template parameter `Memory` shall have two static methods,

```
static void *aligned_malloc(std::size_t n, std::size_t alignment);
static void aligned_free(void *ptr);
```

The method `aligned_malloc` shall behave similar to `std::malloc` with the additional alignment requirement. It shall return a null pointer if it fails to allocate memory. In any other case, including zero input size, it shall return a reachable non-null pointer. The method `aligned_free` shall behave similar to `std::free`. It shall be able to handle a null pointer as its input. The library provides three implementations, discussed below. The default argument `AlignedMemory` is an alias to one of them by default. It can be changed by defining the macro `MCKL_ALIGNED_MEMORY_TYPE`.

`AlignedMemoryTBB` This class uses `scalable_aligned_malloc` and `scalable_aligned_free` from the TBB library. This is the default method if `MCKL_USE_TBB_MALLOC` is true.

AlignedMemorySYS This class uses `posix_memalign` and `free` on POSIX platforms. It uses `_aligned_malloc` and `_aligned_free` if the compiler is MSVC. Otherwise, this class is not defined. If this class is define, it is the default method if `MCKL_USE_TBB_MALLOC` is false.

AlignedMemorySTD This class uses only `std::malloc` and `std::free`. It is the last resort method the library will use.

The default alignment depends on the type `T`. If it is a scalar type (`std::is_scalar<T>`), then the alignment is `MCKL_ALIGNMENT`, whose default is 32. This alignment is sufficient for modern SIMD operations, such as AVX. For other types, the alignment is the maximum of `alignof(T)` and `MCKL_ALIGNMENT_MIN`, whose default is 16. Two container types are defined for convenience,

```
template <typename T>
using Vector = std::vector<T, Allocator<T>>;

template <typename T, std::size_t N,
    std::size_t Alignment = AlignmentTrait<T>::value>
class alignas(Alignment) Array;
```

The first can be used as a drop-in replacement of `std::vector<T>` since it is merely a type alias with a different default allocator. The second can is derived from `std::array<T, N>` and thus can be a drop-in replacement in most situations.

1.2 SAMPLE COVARIANCE

The library provides a class template to estimate sample covariance,

```
template <typename RealType>
class Covariance;
```

At the time of writting, only `float` and `double` are supported. The class has the following operator as its interface,

```
void operator()(MatrixLayout layout, std::size_t n, std::size_t p,
    const RealType *x, const RealType *w, RealType *mean,
    RealType *cov, MatrixLayout cov_layout = RowMajor,
    bool cov_upper = false, bool cov_packed = false)
```

It computes the sample covariance matrix Σ,

$$\Sigma_{i,j} = \frac{\sum_{i=1}^{N} w_i}{(\sum_{i=1}^{N} w_i)^2 - \sum_{i=1}^{n} w_i^2} \sum_{k=1}^{N} w_k (x_{k,i} - \bar{x}_i)(x_{k,j} - \bar{x}_j)$$
$$\bar{x}_i = \frac{1}{\sum_{i=1}^{N} w_i} \sum_{k=1}^{N} w_k x_{k,i}$$

where x is the n by p matrix of samples, and w is the n-vector of weights. Below we given detailed description of each of the parameters,

`layout` The storage layout of sample matrix x. It is assumed to be an n by p matrix.

`n` The number of samples.

`p` The dimension of each sample.

`x` The sample matrix. If it is a null pointer, then no computation is carried out.

`w` The weight vector. If it is a null pointer, then all samples are assigned weight 1.

`mean` Output storage of the mean. If it is a null pointer, then it is ignored.

`cov` Output storage of the covariance matrix. If it is a null pointer, then it is ignored.

`cov_layout` The storage layout of the covariance matrix.

`cov_upper` If `true`, then the upper triangular of the covariance matrix is packed, otherwise the lower triangular is packed. Ignored if `cov_pack` is `false`.

`cov_packed` If `true`, then the covariance matrix is packed.

The last three parameters specify the storage scheme of the covariance matrix. See any reference of BLAS or LAPACK for explanation of the scheme. Below is an example of the class in use,

```
using T = StateMatrix<RowMajor, Dynamic, double>;
Sampler<T> sampler(n, p);
// Configure and iterate the sampler
double mean[p];
double cov[p * (p + 1) / 2];
Covariance eval;
auto x = sampler.particle().value().data();
auto w = sampler.particle().weight().data();
eval(RowMajor, n, p, x, w, mean, cov, RowMajor, false, true);
```

One can later compute the Cholesky decomposition using LAPACK or other linear algebra libraries. Below is an example of using the covariance matrix to generate multivariate Normal proposals,

```
double chol[p * (p + 1) / 2];
double y[p];
LAPACKE_dpptrf(LAPACK_ROW_MAJOR, 'L', p, chol);
NormalMVDistribution<double, p> normal_mv(mean, chol);
normal_mv(rng, y);
```

1.3 STORE OBJECTS IN HDF5 5 FORMAT

If the HDF5 library is available (`MCKL_HAS_HDF5`), it is possible to store `Sampler<T>` objects, etc., in the HDF5 format. For example,

```
hdf5store(sampler, "pf.h5", "sampler", false);
```

creates an HDF5 file named `pf.h5` with the sampler stored as a list in the group `sampler`. If the last argument is `true`, the data is inserted to an existing file. Otherwise a new file is created. In R it can be processed as the following,

```
library(rhdf5)
pf <- as.data.frame(h5read("pf.h5", "sampler"))
```

This creates a `data.frame` similar to that shown in section **??**. The `hdf5store` function is overloaded for `StateMatrix`, `Sampler<T>` and `Monitor<T>`. It is also overloaded for `Particle<T>` if an overload for `T` is defined. The all follow the format as above. In addition, the following creates a new empty HDF5 file,

```
hdf5store(filename);
```

while the following create a new group named `dataname`,

```
hdf5store(filename, dataname, append);
```

1.4 RAII CLASSES FOR OPENCL POINTERS

The library provides a few classes to manager OpenCL pointers. It provides RAII idiom on top of the OpenCL C interface. For example, below is a small program,

Class	OpenCL pointer type
`CLPlatform`	`cl_platform_id`
`CLContext`	`cl_context`
`CLDevice`	`cl_device_id`
`CLCommandQueue`	`cl_command_queue`
`CLMemory`	`cl_mem`
`CLProgram`	`cl_program`
`CLKernel`	`cl_kernel`
`CLEvent`	`cl_event`

TABLE 1.1 RAII classes for OpenCL pointers

```
auto platform = cl_get_platform().front();
auto device = platform.get_device(CL_DEVICE_TYPE_DEFAULT).front();
CLContext context(CLContextProperties(platform), 1, &device);
CLCommandQueue command_queue(context, device);
CLMemory buffer(context, CL_MEM_READ_WRITE, size);
std::string source = /* read source */;
CLProgram program(context, 1, &source);
program.build(1, &device);
CLKernel kernel(program, "kernel_name");
kernel.set_arg(0, buffer);
command_queue.enqueue_nd_range_kernel(kernel, 1, CLNDRange(), CLNDRange(N),
    CLNDRange());
```

In the above program, each class type object manages an OpenCL C type, such as `cl_platform`. The resources will be released when the object is destroyed. Note that, the copy constructor and assignment operator perform shallow copy. This is particularly important for `CLMemory` type objects. In Appendix **??**, an OpenCL implementation of the simple particle filter example in section **??** is shown. Table 1.1 lists the classes defined by the library and their corresponding OpenCL C pointer types.

1.5 PROCESS COMMAND LINE PROGRAM OPTIONS

The library provides some basic support for processing command line options. Here we show a minimal example. The complete program is shown in Appendix 1.5. First, we define variables to store values of options,

```
int n;
std::string str;
std::vector<double> vec;
```

All types that support standard library I/O stream operations are supported. In addition, for any type `T` that supports such operations, `std::vector<T, Alloc>`, is also supported. Then,

```
ProgramOptionMap option_map;
```

constructs the container of options. Options can be added to the map,

```
option_map
    .add("str", "A string option with a default value", &str, "default")
    .add("n", "An integer option", &n)
    .add("vec", "A vector option", &vec);
```

The first argument is the name of the option, the second is a description, and the third is a pointer to where the value of the option shall be stored. The last optional argument is a default value. The options on the command line can be processed as the following,

```
option_map.process(argc, argv);
```

where `argc` and `argv` are the arguments of the `main` function. When the program is invoked, each option can be passed to it like below,

```
./program_option --vec 1 2 1e-1 --str "abc" --vec 8 9 --str "def hij" --n 2 4
```

The results of the option processing is displayed below,

```
n: 4
str: def hij
vec: 1 2 0.1 8 9
```

To summarize these output, the same option can be specified multiple times. If it is a scalar option, the last one is used (`--str`, `--n`). The value of a string option can be grouped by quotes. For a vector option (`--vec`), all values are gathered together and inserted into the vector.

1.6 DISPLAY PROGRAM PROGRESS

Sometime it is desirable to see how much progress of a program has been made. The library provides a `Progress` class for this purpose. Here we show a minimal example.

```
Progress progress;
progress.start(n * n);
for (std::size_t i = 0; i != n; ++i) {
    progress.message("i = " + std::to_string(i));
    for (std::size_t j = 0; j != n; ++j) {
        // Do some computation
        progress.increment();
    }
}
progress.stop();
```

When invoked, the program output something similar the following,

```
[  4%][00:07][  49019/1000000][i = 49]
```

The method `start` starts the printing of the progress. The argument specifies how many iterations there will be before it is stopped. The method `message` direct the program to print a message. This is optional. Each time after we finish n iterations (there are n^3 total iterations of the inner-most loop), we increment the progress count by calling `increment`. And after everything is finished, the method `stop` is called.

1.7 STOP WATCH

Performance can only be improved after it is first properly benchmarked. There are advanced profiling programs for this purpose. However, sometime simple timing facilities are enough. The library provides a simple class `StopWatch` for this purpose. As its name suggests, it works much like a physical stop watch. Here is a simple example

```
StopWatch watch;
for (std::size_t i = 0; i != n; ++i) {
    // Some computation
    watch.start();
```

```
    // Computation to be benchmarked;
    watch.stop();
    // Some other computation
}
double t = watch.seconds(); // The time in seconds
```

The above example demonstrate that timing can be accumulated between loop iterations, function calls, etc. It shall be noted that, the timing is only accurate if the computation between `start` and `stop` is non-trivial.