Help

```
#ifndef _WORKSPACE_HPP
#define _WORKSPACE_HPP

#ifdef _OPENMP
#include <omp.h>
#endif
#include <ctime>
#include <algorithm>

#include "pnl/pnl_vector.h"
#include "pnl/pnl_matrix.h"
#include "pnl/pnl_random.h"
#ifdef HAVE_MPI
#include <mpi.h>
#endif

namespace mcam {
template <typename _T, typename _Alloc> class Workspace
{
protected:
    _T *data; /// The array of data
    int nb_elements; /// Number of elements in #data
    _Alloc allocator; /// An instance of the allocator class

public:
    Workspace()
    {
#ifdef HAVE_MPI
        int useMPI = false;
        MPI_Initialized(&useMPI);
        if (useMPI)
        {
            nb_elements = 1;
            data = new _T[1];
            MPI_Comm_rank(MPI_COMM_WORLD, &(allocator.num_thread));
            data[0] = allocator.create();
            return;
        }
#endif
```

```
#ifdef _OPENMP
        nb_elements = omp_get_max_threads();
#else
        nb_elements = 1;
#endif
        data = new _T[nb_elements];
        for (int i = 0; i < nb_elements; i++)
        {
            allocator.num_thread = i;
            data[i] = allocator.create();
        }
    }


    ~Workspace()
    {
        for (int i = 0 ; i < nb_elements ; i++)
        {
            allocator.destroy(&(data[i]));
        }
        delete [] data;
    }

    /**
     * With OpenMP on, return the instance indexed by the thread number.
     * With OpenMP off, returns the single instance.
     *
     * @return the current workspace
     */
    _T& operator()() const
    {
#ifdef _OPENMP
        return operator()(omp_get_thread_num());
#else
        return operator()(0);
#endif
    }

    /**
     * @param i an integer between 0 and nb_threads
     *
```

```
     * @return data[i]
     */
    _T& operator()(int i) const
    {
#ifndef PNL_RANGE_CHECK_OFF
        if (i > nb_elements)
        {
            printf("Accessing too many workspaces\ n");
            abort();
        }
#endif
        return data[i];
    }
};

/**
 * Allocator strcuture for PnlVect*
 */
struct AllocPnlVect
{
    int num_thread;
    PnlVect *create()
    {
        return pnl_vect_new();
    }
    void destroy(PnlVect **v)
    {
        pnl_vect_free(v);
    }
};

/**
 * Allocator strcuture for PnlMat*
 */
struct AllocPnlMat
{
    int num_thread;
    PnlMat *create()
    {
        return pnl_mat_new();
    }
```

```
    void destroy(PnlMat **M)
    {
        pnl_mat_free(M);
    }
};

/**
 * Allocator strcuture for PnlRng*. The created random number generators are
 * independent and suitable for usage with OpenMP.
 */
struct AllocPnlRng
{
    int num_thread;
    PnlRng *create()
    {
        PnlRng *rng = pnl_rng_dcmt_create_id(num_thread, 3128);
        pnl_rng_sseed(rng, 0);
        return rng;
    }
    void destroy(PnlRng **rng)
    {
        pnl_rng_free(rng);
    }
};

typedef Workspace<PnlVect *, AllocPnlVect> PnlVect_Workspace;
typedef Workspace<PnlMat *, AllocPnlMat> PnlMat_Workspace;

class PnlRng_Workspace: public Workspace<PnlRng *, AllocPnlRng>
{
public:
    PnlRng_Workspace()
        : Workspace() { }
    void set_seed(unsigned long s)
    {
        for (int i = 0 ; i < nb_elements ; i++)
        {
            pnl_rng_sseed(data[i], s);
        }
    }
};
```

```
}


#endif /* _WORKSPACE_HPP */
```