C++ Software Engineering Portfolio

Mahdi Tavakol — Postdoctoral Researcher, University of Oxford

Selected C++17/20 projects demonstrating modern design, numerical algorithms, and parallel computing on Linux.

Focus Areas:

❑ RAII, Eigen, and exception-safe design
❑ Object Oriented code design
❑ OpenMP and MPI parallelism
❑ CMake and Git
❑ High-performance numerical algorithms and simulation frameworks

Project #1: Constant – pH simulation

LAMMPS plugin implementing protonation state transitions.

Goal : On-the-fly constant pH simulation

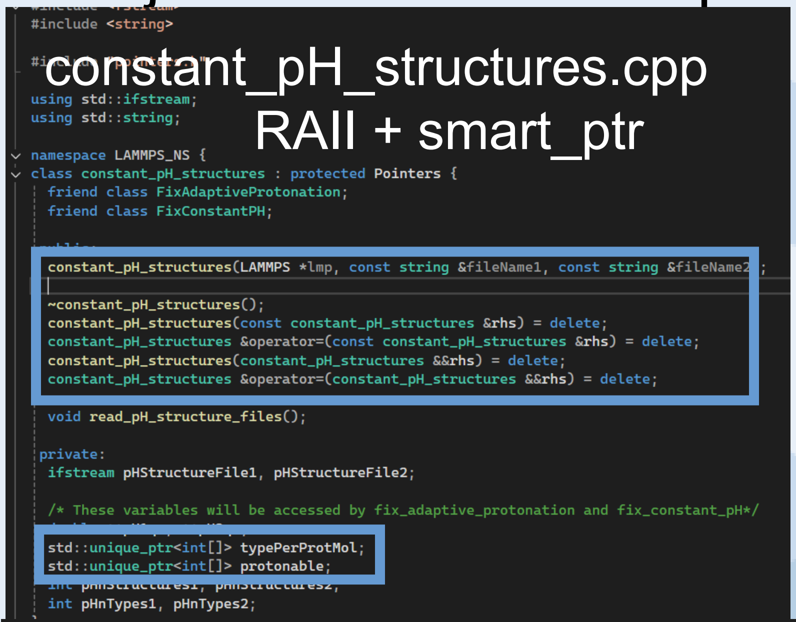
$$V\ (R,\lambda) = (1-\lambda) * V_A(R) * \lambda * V_B(R) + V^{pH}(\lambda) + V^{MM}(\lambda) + V^{bias}(\lambda)$$

Code architecture:

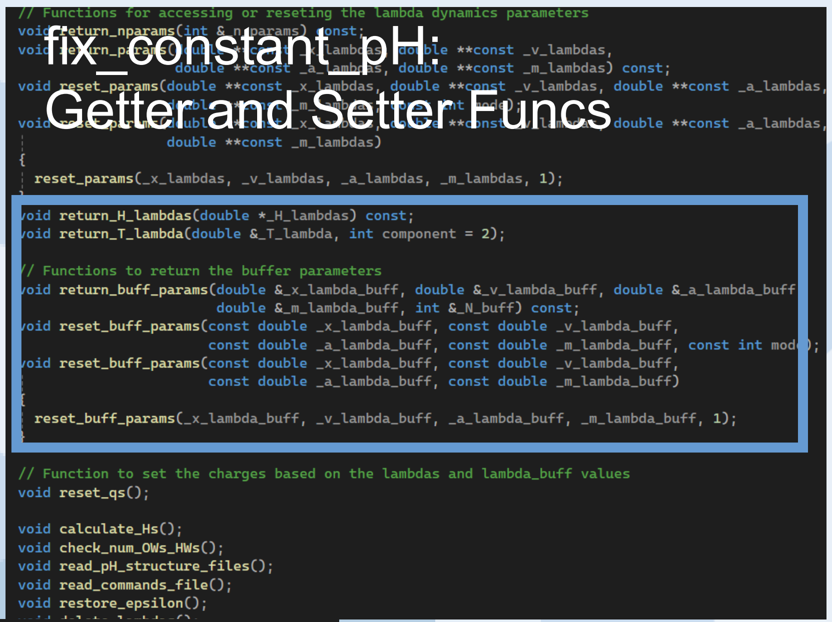
1. Data Layer: constant_pH_structures.cpp
2. State & physics: Fix_constant_pH.cpp (protonation state, buffer, potentials, commands hook)
3. Integration: Fix_nh_constant_pH.cpp (integrates DOFs, applies constrain on the total charge)
4. Logic : Fix_adaptive_protonation.cpp (environment, change qs)
5. Output state: Compute_temp_constant_pH.cpp (computes temp)
6. Analysis: Compute_solute_coordination.cpp (environment)

Input (1) – **Core Engine** (2-3) – **Adaptive Logic** (4) – **Analysis** (5-6)

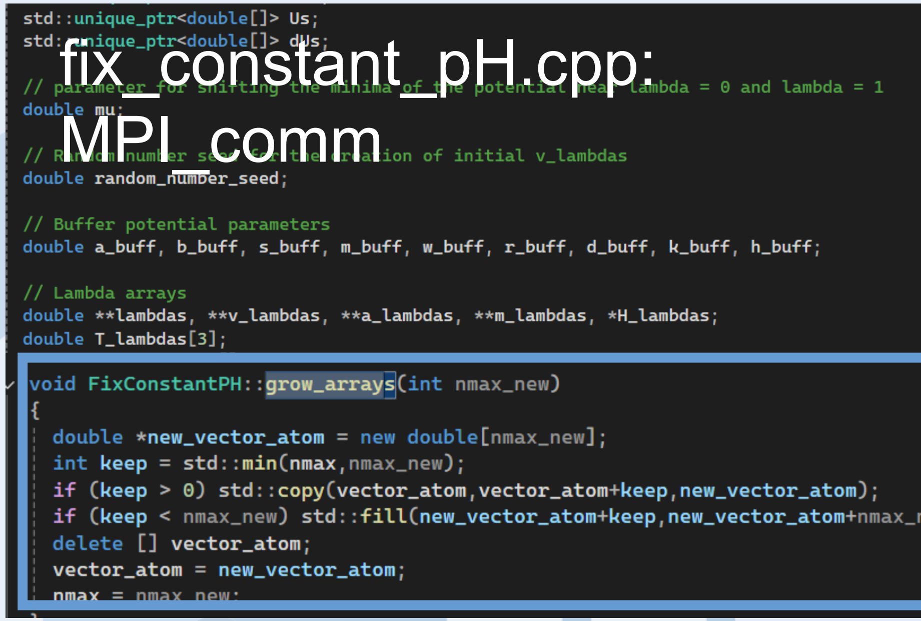
Project #1: Constant – pH simulation


constant_pH_structures.cpp
RAII + smart_ptr

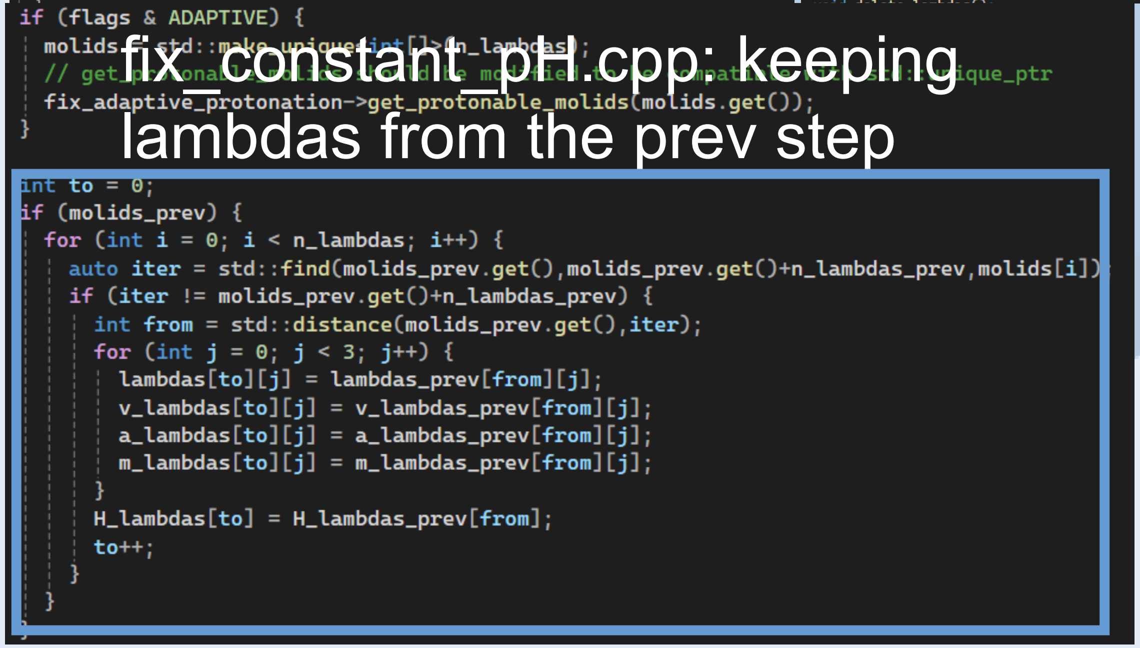
```cpp
#include <string>

#include "pointers.h"

using std::ifstream;
using std::string;

namespace LAMMPS_NS {
class constant_pH_structures : protected Pointers {
  friend class FixAdaptiveProtonation;
  friend class FixConstantPH;

 public:
  constant_pH_structures(LAMMPS *lmp, const string &fileName1, const string &fileName2);

  ~constant_pH_structures();
  constant_pH_structures(const constant_pH_structures &rhs) = delete;
  constant_pH_structures &operator=(const constant_pH_structures &rhs) = delete;
  constant_pH_structures(constant_pH_structures &&rhs) = delete;
  constant_pH_structures &operator=(constant_pH_structures &&rhs) = delete;

  void read_pH_structure_files();

 private:
  ifstream pHStructureFile1, pHStructureFile2;

  /* These variables will be accessed by fix_adaptive_protonation and fix_constant_pH*/
  std::unique_ptr<int[]> typePerProtMol;
  std::unique_ptr<int[]> protonable;
  int pHnStructures1, pHnStructures2;
  int pHnTypes1, pHnTypes2;
```


fix_constant_pH:
Getter and Setter Funcs

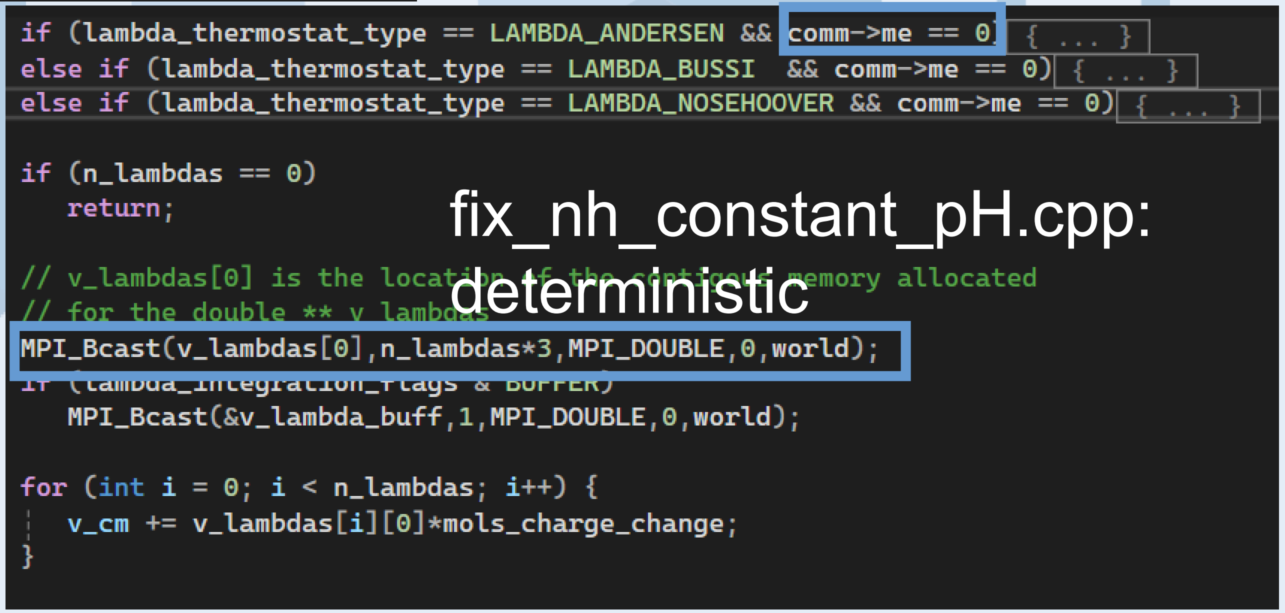
```cpp
// Functions for accessing or reseting the lambda dynamics parameters
void return_nparams(int & n_params) const;
void return_params(double **const _x_lambdas, double **const _v_lambdas,
                   double **const _a_lambdas, double **const _m_lambdas) const;
void reset_params(double **const _x_lambdas, double **const _v_lambdas, double **const _a_lambdas,
                  double **const _m_lambdas, const int mode);
void reset_params(double **const _x_lambdas, double **const _v_lambdas, double **const _a_lambdas,
                  double **const _m_lambdas)
{
  reset_params(_x_lambdas, _v_lambdas, _a_lambdas, _m_lambdas, 1);
}
void return_H_lambdas(double *_H_lambdas) const;
void return_T_lambda(double &_T_lambda, int component = 2);

// Functions to return the buffer parameters
void return_buff_params(double &_x_lambda_buff, double &_v_lambda_buff, double &_a_lambda_buff
                        double &_m_lambda_buff, int &_N_buff) const;
void reset_buff_params(const double _x_lambda_buff, const double _v_lambda_buff,
                       const double _a_lambda_buff, const double _m_lambda_buff, const int mode);
void reset_buff_params(const double _x_lambda_buff, const double _v_lambda_buff,
                       const double _a_lambda_buff, const double _m_lambda_buff)
{
  reset_buff_params(_x_lambda_buff, _v_lambda_buff, _a_lambda_buff, _m_lambda_buff, 1);
}

// Function to set the charges based on the lambdas and lambda_buff values
void reset_qs();

void calculate_Hs();
void check_num_OWs_HWs();
void read_pH_structure_files();
void read_commands_file();
void restore_epsilon();
```


fix_constant_pH.cpp:
MPI_comm

```cpp
std::unique_ptr<double[]> Us;
std::unique_ptr<double[]> dUs;

// parameter for shifting the minima of the potential near lambda = 0 and lambda = 1
double mu;

// Random number seed for the creation of initial v_lambdas
double random_number_seed;

// Buffer potential parameters
double a_buff, b_buff, s_buff, m_buff, w_buff, r_buff, d_buff, k_buff, h_buff;

// Lambda arrays
double **lambdas, **v_lambdas, **a_lambdas, **m_lambdas, *H_lambdas;
double T_lambdas[3];

void FixConstantPH::grow_arrays(int nmax_new)
{
  double *new_vector_atom = new double[nmax_new];
  int keep = std::min(nmax,nmax_new);
  if (keep > 0) std::copy(vector_atom,vector_atom+keep,new_vector_atom);
  if (keep < nmax_new) std::fill(new_vector_atom+keep,new_vector_atom+nmax_
  delete [] vector_atom;
  vector_atom = new_vector_atom;
  nmax = nmax_new;
```


fix_constant_pH.cpp: keeping
lambdas from the prev step

```cpp
if (flags & ADAPTIVE) {
  molids = std::make_unique<int[]>(n_lambdas);
  // get_protonable_molids should be modified to be compatible with std::unique_ptr
  fix_adaptive_protonation->get_protonable_molids(molids.get());
}

int to = 0;
if (molids_prev) {
  for (int i = 0; i < n_lambdas; i++) {
    auto iter = std::find(molids_prev.get(),molids_prev.get()+n_lambdas_prev,molids[i]);
    if (iter != molids_prev.get()+n_lambdas_prev) {
      int from = std::distance(molids_prev.get(),iter);
      for (int j = 0; j < 3; j++) {
        lambdas[to][j] = lambdas_prev[from][j];
        v_lambdas[to][j] = v_lambdas_prev[from][j];
        a_lambdas[to][j] = a_lambdas_prev[from][j];
        m_lambdas[to][j] = m_lambdas_prev[from][j];
      }
      H_lambdas[to] = H_lambdas_prev[from];
      to++;
    }
  }
```


fix_nh_constant_pH.cpp:
deterministic

```cpp
if (lambda_thermostat_type == LAMBDA_ANDERSEN && comm->me == 0) { ... }
else if (lambda_thermostat_type == LAMBDA_BUSSI  && comm->me == 0) { ... }
else if (lambda_thermostat_type == LAMBDA_NOSEHOOVER && comm->me == 0) { ... }

if (n_lambdas == 0)
  return;

// v_lambdas[0] is the location of the contiguous memory allocated
// for the double ** v_lambdas
MPI_Bcast(v_lambdas[0],n_lambdas*3,MPI_DOUBLE,0,world);
if (lambda_integration_flags & BUFFER)
  MPI_Bcast(&v_lambda_buff,1,MPI_DOUBLE,0,world);

for (int i = 0; i < n_lambdas; i++) {
  v_cm += v_lambdas[i][0]*mols_charge_change;
}
```

Modern C++ 17 – RAII – MPI – Deterministic Integration – Modular Design

Project #2: C++17 Plugin Architecture for Adaptive Biasing (LAMMPS Extension)

Adaptive bias potential framework for free-energy surface exploration.

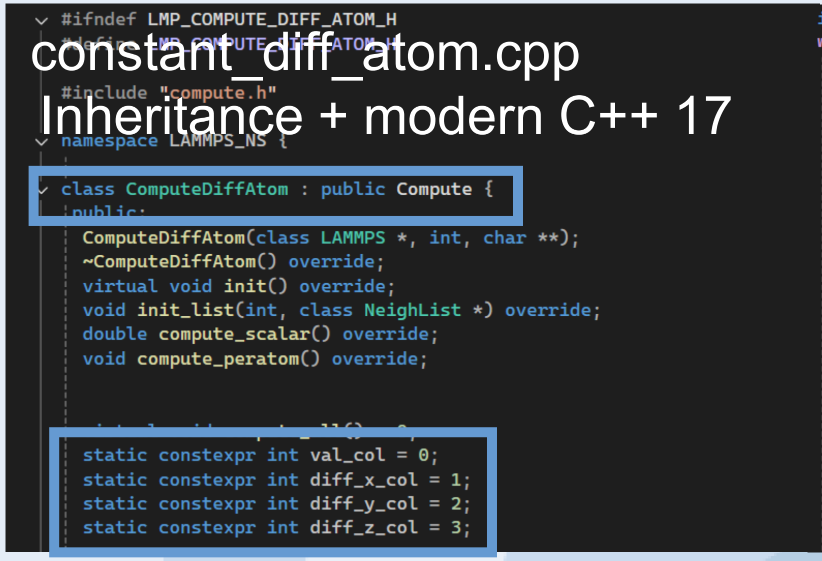
Goal : Apply adaptive biasing forces to escape metastable states

Code architecture:

1. Parent abstract class: compute_diff_atom.cpp (pure abstract class)
   a) compute_entropy_atom.h (user defined CVs)
   b) compute_cparam_atom.h (user defined CVs)
   c) compute_enthalpy_atom.h (user defined CVs)
   d) compute_q6_smooth_atom.h (complex math, many ghost communications, on-the-fly computation due to large memory demand)
2. State & physics: Fix_metaAR.h (Integration logic, file I/O, history management, bias update scheme)
3. Integration: fix_nvt.h (integrates DOFs, applies constraint on the total charge)

Input (1) – **Core Engine** (2-3) – **Adaptive Logic** (2) – **Analysis** (2)

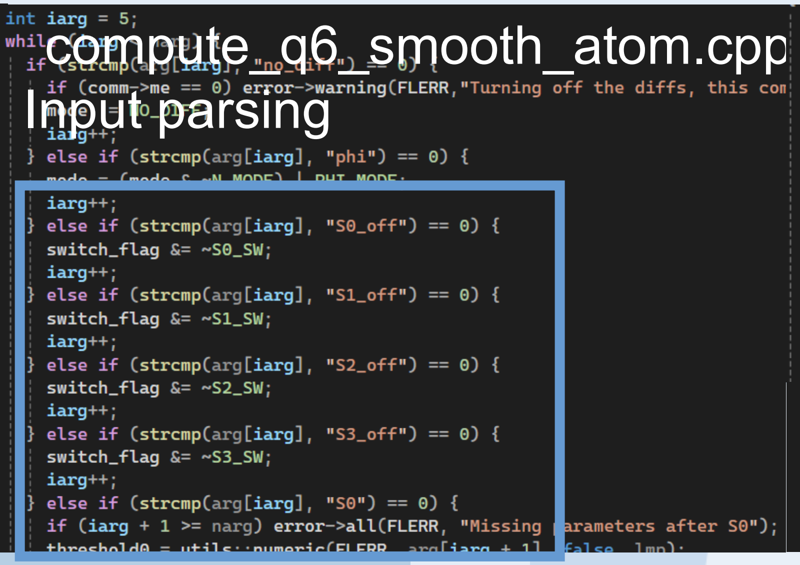
Project #2: C++17 Plugin Architecture for Adaptive Biasing (LAMMPS Extension)



constant_diff_atom.cpp

Inheritance + modern C++ 17

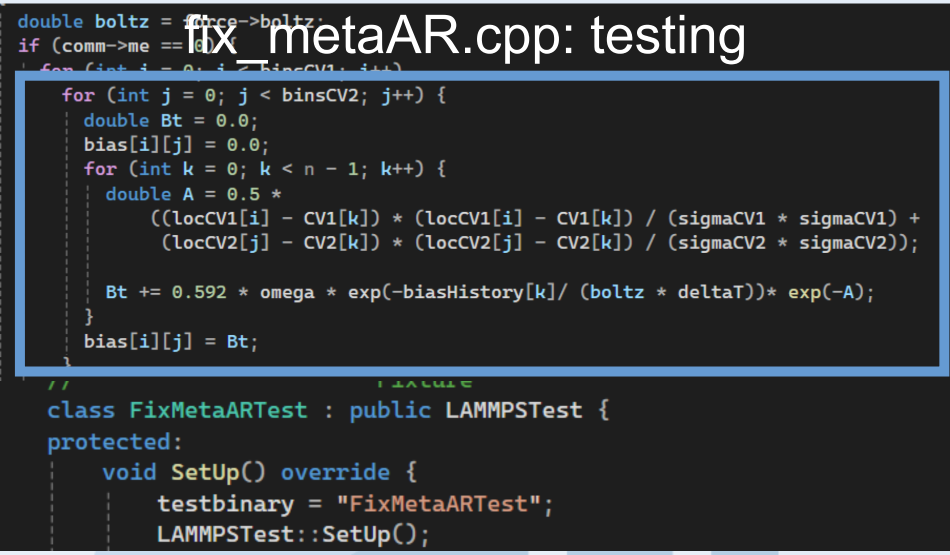
```cpp
#ifndef LMP_COMPUTE_DIFF_ATOM_H
[illegible]

#include "compute.h"

namespace LAMMPS_NS {

class ComputeDiffAtom : public Compute {
 public:
  ComputeDiffAtom(class LAMMPS *, int, char **);
  ~ComputeDiffAtom() override;
  virtual void init() override;
  void init_list(int, class NeighList *) override;
  double compute_scalar() override;
  void compute_peratom() override;


  [illegible]
  static constexpr int val_col = 0;
  static constexpr int diff_x_col = 1;
  static constexpr int diff_y_col = 2;
  static constexpr int diff_z_col = 3;
```



compute_q6_smooth_atom.cpp

Input parsing

```cpp
int iarg = 5;
while [illegible]
  if (strcmp(arg[iarg], "no_diff") == 0) {
    if (comm->me == 0) error->warning(FLERR,"Turning off the diffs, this com
    mode = NO_DIFF;
    iarg++;
  } else if (strcmp(arg[iarg], "phi") == 0) {
    mode = (mode & ~N_MODE) | PHI_MODE;
    iarg++;
  } else if (strcmp(arg[iarg], "S0_off") == 0) {
    switch_flag &= ~S0_SW;
    iarg++;
  } else if (strcmp(arg[iarg], "S1_off") == 0) {
    switch_flag &= ~S1_SW;
    iarg++;
  } else if (strcmp(arg[iarg], "S2_off") == 0) {
    switch_flag &= ~S2_SW;
    iarg++;
  } else if (strcmp(arg[iarg], "S3_off") == 0) {
    switch_flag &= ~S3_SW;
    iarg++;
  } else if (strcmp(arg[iarg], "S0") == 0) {
    if (iarg + 1 >= narg) error->all(FLERR, "Missing parameters after S0");
    threshold0 = utils::numeric(FLERR, arg[iarg + 1], false, lmp);
```



fix_metaAR.cpp: testing

```cpp
double boltz = force->boltz;
if (comm->me == 0) {
  for (int i = 0; i < binsCV1; i++)
    for (int j = 0; j < binsCV2; j++) {
      double Bt = 0.0;
      bias[i][j] = 0.0;
      for (int k = 0; k < n - 1; k++) {
        double A = 0.5 *
            ((locCV1[i] - CV1[k]) * (locCV1[i] - CV1[k]) / (sigmaCV1 * sigmaCV1) +
             (locCV2[j] - CV2[k]) * (locCV2[j] - CV2[k]) / (sigmaCV2 * sigmaCV2));

        Bt += 0.592 * omega * exp(-biasHistory[k]/ (boltz * deltaT))* exp(-A);
      }
      bias[i][j] = Bt;
    }
```

```cpp
  //                              Fixture
  class FixMetaARTest : public LAMMPSTest {
  protected:
      void SetUp() override {
          testbinary = "FixMetaARTest";
          LAMMPSTest::SetUp();
```

compute_q6_smooth_atom:

$$N[i] = s2\left(\sum_{j} s1(b_{ij}) * s0(r_{ij})\right)$$

$$\Phi[i] = \sum\sum s3(r_{ij}) * N[i] * N[j]$$

$$bij = \sum_{m=-6}^{m=6} q6[i][m] * q6[j][m]$$

$$\Phi' = N[i] * \left(\sum \frac{dNi}{drj}\right) + \frac{dNi}{dri} * (\sum Nj)$$

1. Central atom q6m, diffq6m and coeffs
2. Fill ghost atoms q6m, diffq6m, s2, ds2 and coeffs
3. dcij/dri = qj*dqi/dri(A) + qi*dqj/dri(B) (We do not have the B)
   1. Fill A in for the atom i and fill B for the atom j in h[j]
   2. Add from the h calculated by the ghost atoms
4. Add the diff related to s2 and s0 (distance)
5. We have Ni and diffNi/diffri
6. Send Ni to ghost atoms
7. Calculate the dNj/dri on-the-fly from qj and coeffs

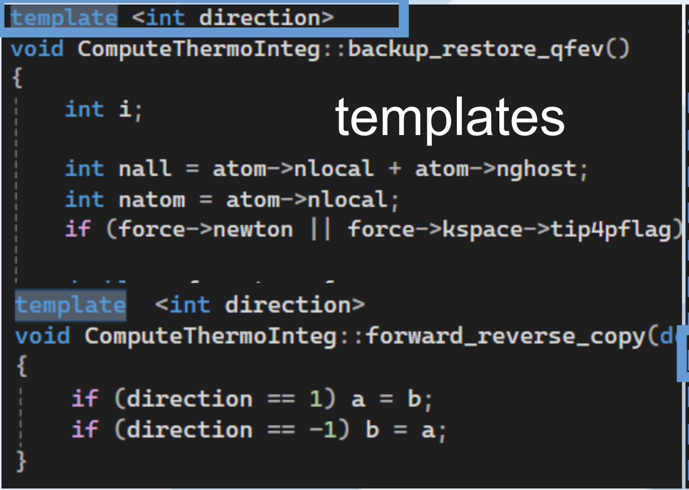
RAII-safe memory allocation (LAMMPS Memory API)- Constexpr – Virtual overrides – C++ smart pointers – On-the-fly calculations to avoid heap exhaustion

Project #3: Free energy Part A: C++ Thermodynamic Integration module

Custom C++ LAMMPS compute for thermodynamic-integration (TI) energy derivatives.

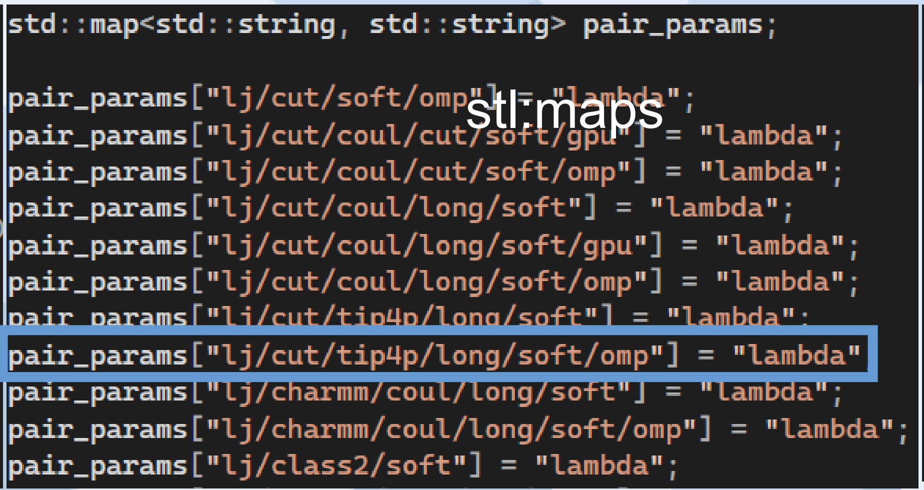
Code architecture:

1. Parent abstract class: compute.h (pure abstract class)
   compute_thermo_integ.h



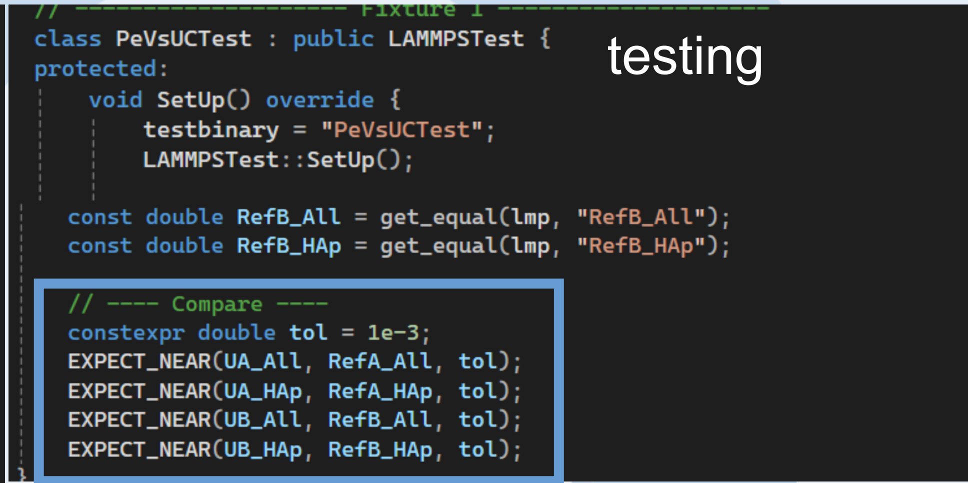
templates

```
template <int direction>
void ComputeThermoInteg::backup_restore_qfev()
{
    int i;

    int nall = atom->nlocal + atom->nghost;
    int natom = atom->nlocal;
    if (force->newton || force->kspace->tip4pflag)

template  <int direction>
void ComputeThermoInteg::forward_reverse_copy(d
{
    if (direction == 1) a = b;
    if (direction == -1) b = a;
}
```



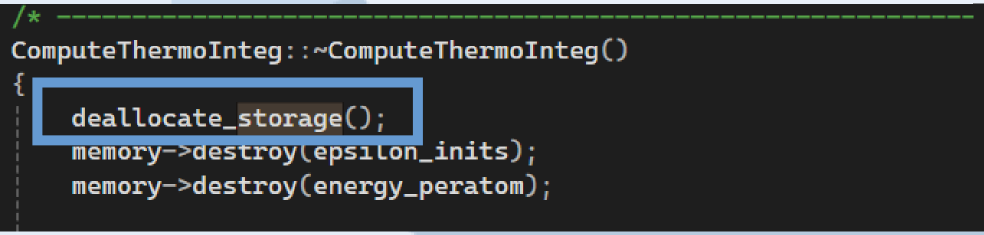
stl::maps

```
std::map<std::string, std::string> pair_params;

pair_params["lj/cut/soft/omp"] = "lambda";
pair_params["lj/cut/coul/cut/soft/gpu"] = "lambda";
pair_params["lj/cut/coul/cut/soft/omp"] = "lambda";
pair_params["lj/cut/coul/long/soft"] = "lambda";
pair_params["lj/cut/coul/long/soft/gpu"] = "lambda";
pair_params["lj/cut/coul/long/soft/omp"] = "lambda";
pair_params["lj/cut/tip4p/long/soft"] = "lambda";
pair_params["lj/cut/tip4p/long/soft/omp"] = "lambda"
pair_params["lj/charmm/coul/long/soft"] = "lambda";
pair_params["lj/charmm/coul/long/soft/omp"] = "lambda";
pair_params["lj/class2/soft"] = "lambda";
```

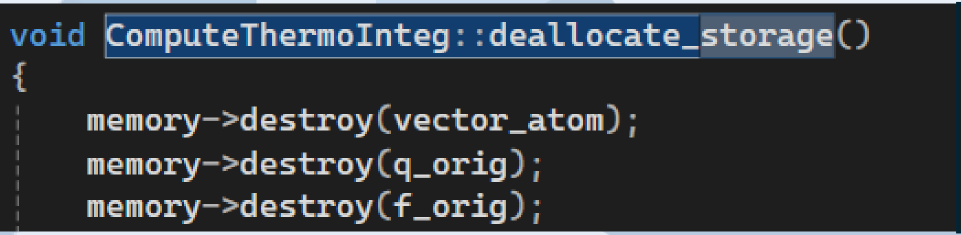


testing

```
// -------------------- Fixture 1 --------------------
class PeVsUCTest : public LAMMPSTest {
protected:
    void SetUp() override {
        testbinary = "PeVsUCTest";
        LAMMPSTest::SetUp();

    const double RefB_All = get_equal(lmp, "RefB_All");
    const double RefB_HAp = get_equal(lmp, "RefB_HAp");

    // ---- Compare ----
    constexpr double tol = 1e-3;
    EXPECT_NEAR(UA_All, RefA_All, tol);
    EXPECT_NEAR(UA_HAp, RefA_HAp, tol);
    EXPECT_NEAR(UB_All, RefB_All, tol);
    EXPECT_NEAR(UB_HAp, RefB_HAp, tol);
}
```

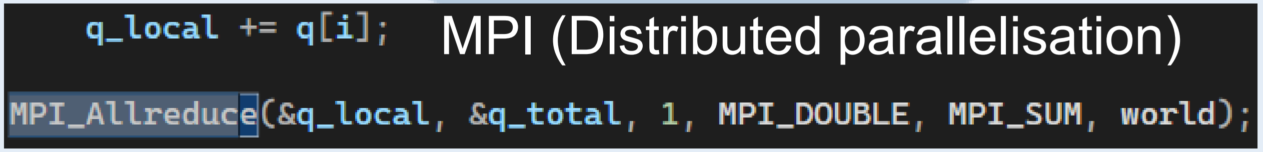


```
/* ----------------------------------------------------------------
ComputeThermoInteg::~ComputeThermoInteg()
{
    deallocate_storage();
    memory->destroy(epsilon_inits);
    memory->destroy(energy_peratom);
```



```
void ComputeThermoInteg::deallocate_storage()
{
    memory->destroy(vector_atom);
    memory->destroy(q_orig);
    memory->destroy(f_orig);
```

RAII implementation for double**
(mdspan was not available yet!)

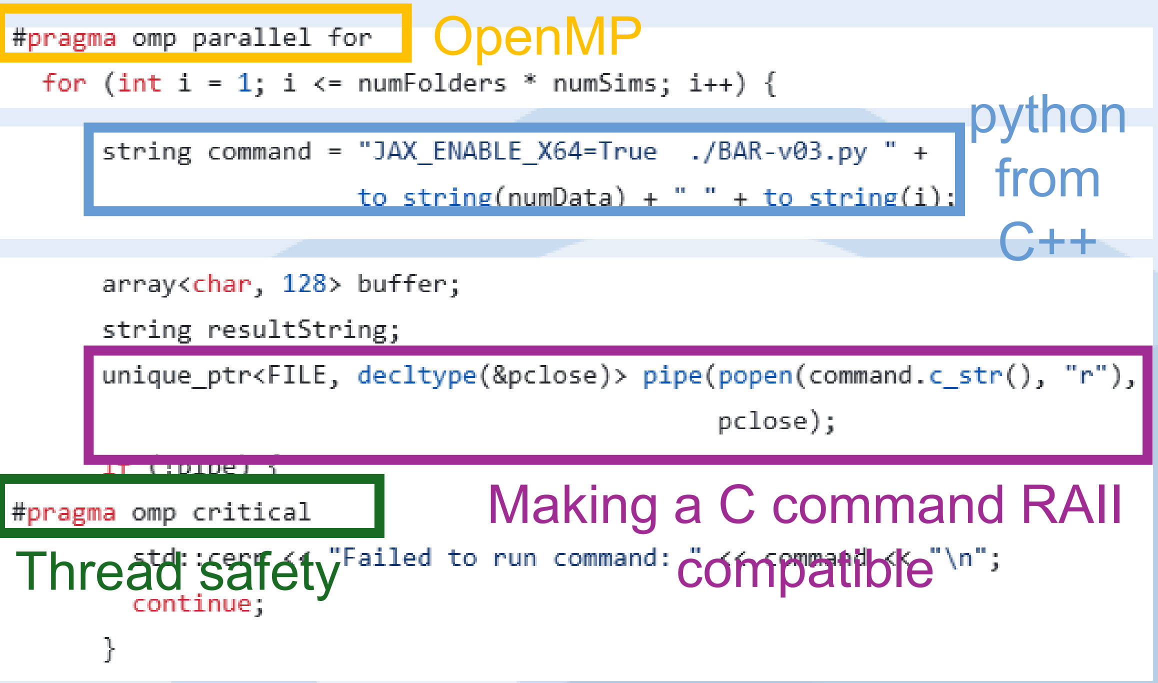


```
    q_local += q[i];
MPI_Allreduce(&q_local, &q_total, 1, MPI_DOUBLE, MPI_SUM, world);
```

MPI (Distributed parallelisation)

Verified, reproducible, and parallel-safe C++ compute module integrated with LAMMPS core

Project #3: Free energy Part B: C++ and Python Pipeline for high throughput data



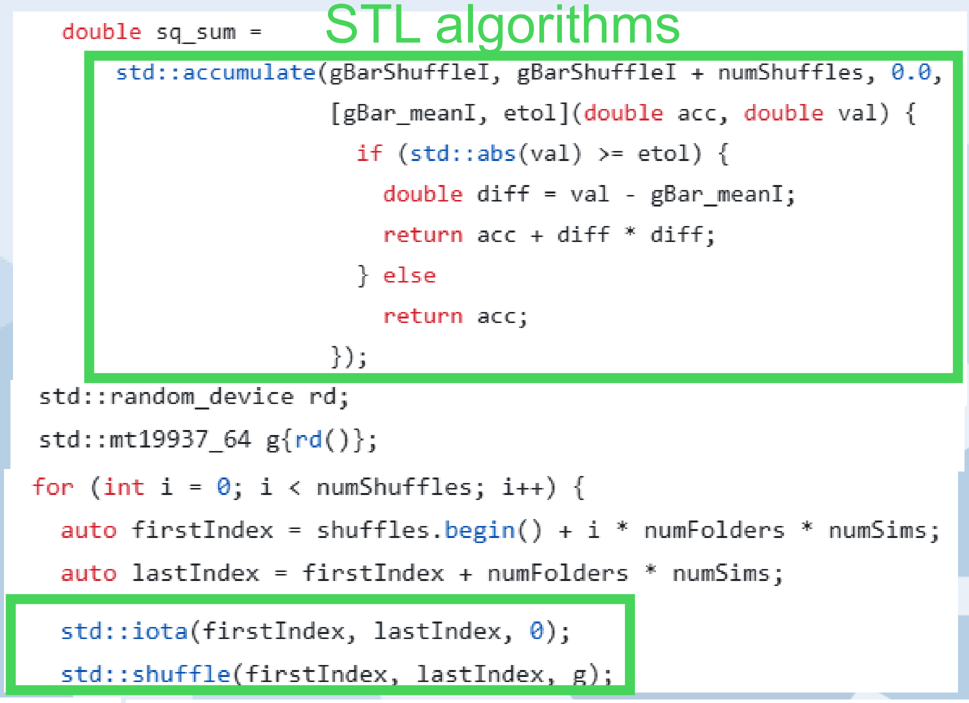
OpenMP

python from C++

Making a C command RAII compatible

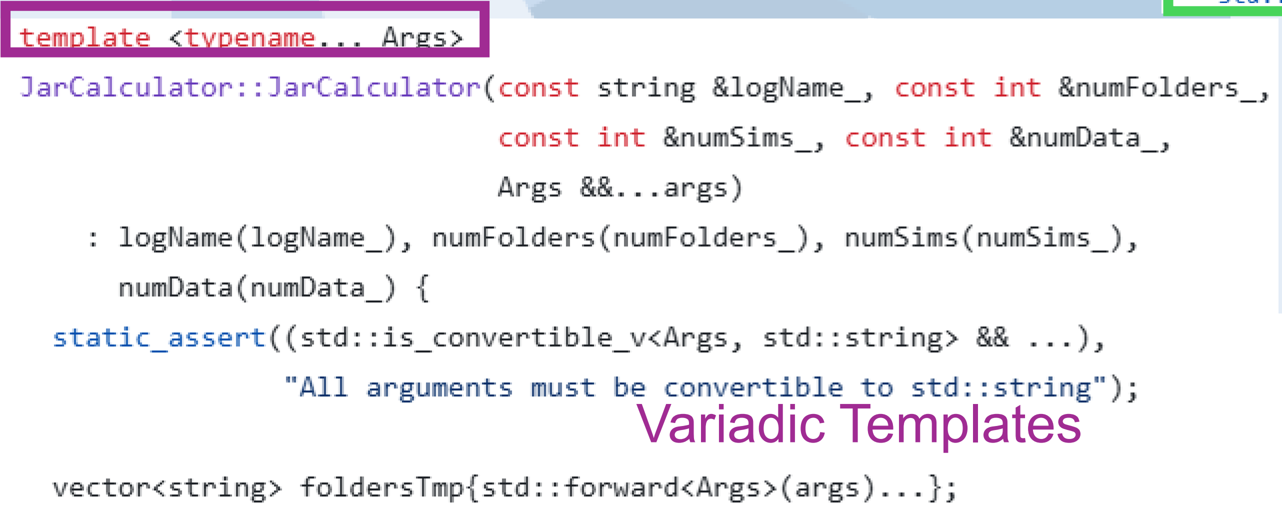
Thread safety

```
#pragma omp parallel for
  for (int i = 1; i <= numFolders * numSims; i++) {

    string command = "JAX_ENABLE_X64=True  ./BAR-v03.py " +
                     to_string(numData) + " " + to_string(i);

    array<char, 128> buffer;
    string resultString;
    unique_ptr<FILE, decltype(&pclose)> pipe(popen(command.c_str(), "r"),
                                              pclose);
    if (!pipe) {
#pragma omp critical
      std::cerr << "Failed to run command: " << command << "\n";
      continue;
    }
```



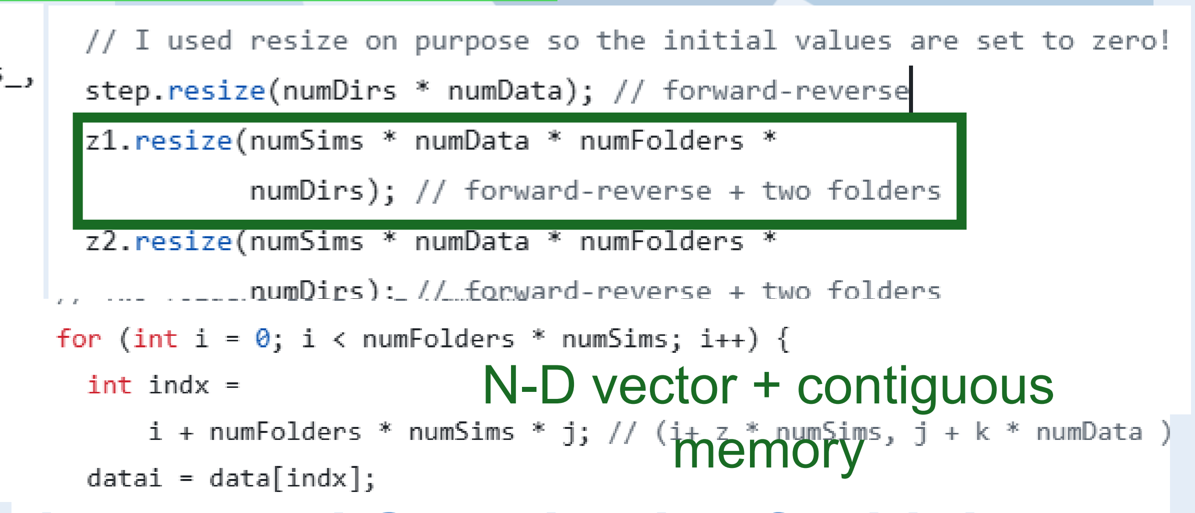
STL algorithms

```
  double sq_sum =
      std::accumulate(gBarShuffleI, gBarShuffleI + numShuffles, 0.0,
                      [gBar_meanI, etol](double acc, double val) {
                        if (std::abs(val) >= etol) {
                          double diff = val - gBar_meanI;
                          return acc + diff * diff;
                        } else
                          return acc;
                      });
std::random_device rd;
std::mt19937_64 g{rd()};
for (int i = 0; i < numShuffles; i++) {
  auto firstIndex = shuffles.begin() + i * numFolders * numSims;
  auto lastIndex = firstIndex + numFolders * numSims;

  std::iota(firstIndex, lastIndex, 0);
  std::shuffle(firstIndex, lastIndex, g);
```



Variadic Templates

```
template <typename... Args>
JarCalculator::JarCalculator(const string &logName_, const int &numFolders_,
                             const int &numSims_, const int &numData_,
                             Args &&...args)
    : logName(logName_), numFolders(numFolders_), numSims(numSims_),
      numData(numData_) {
  static_assert((std::is_convertible_v<Args, std::string> && ...),
                "All arguments must be convertible to std::string");

  vector<string> foldersTmp{std::forward<Args>(args)...};
```



N-D vector + contiguous memory

```
  // I used resize on purpose so the initial values are set to zero!
  step.resize(numDirs * numData); // forward-reverse
  z1.resize(numSims * numData * numFolders *
            numDirs); // forward-reverse + two folders
  z2.resize(numSims * numData * numFolders *
            numDirs); // forward-reverse + two folders

for (int i = 0; i < numFolders * numSims; i++) {
  int indx =
      i + numFolders * numSims * j; // (i+ z * numSims, j + k * numData )
  datai = data[indx];
```

Parallel, memory-efficient, and language-integrated C++ pipeline for high-throughput free-energy computation

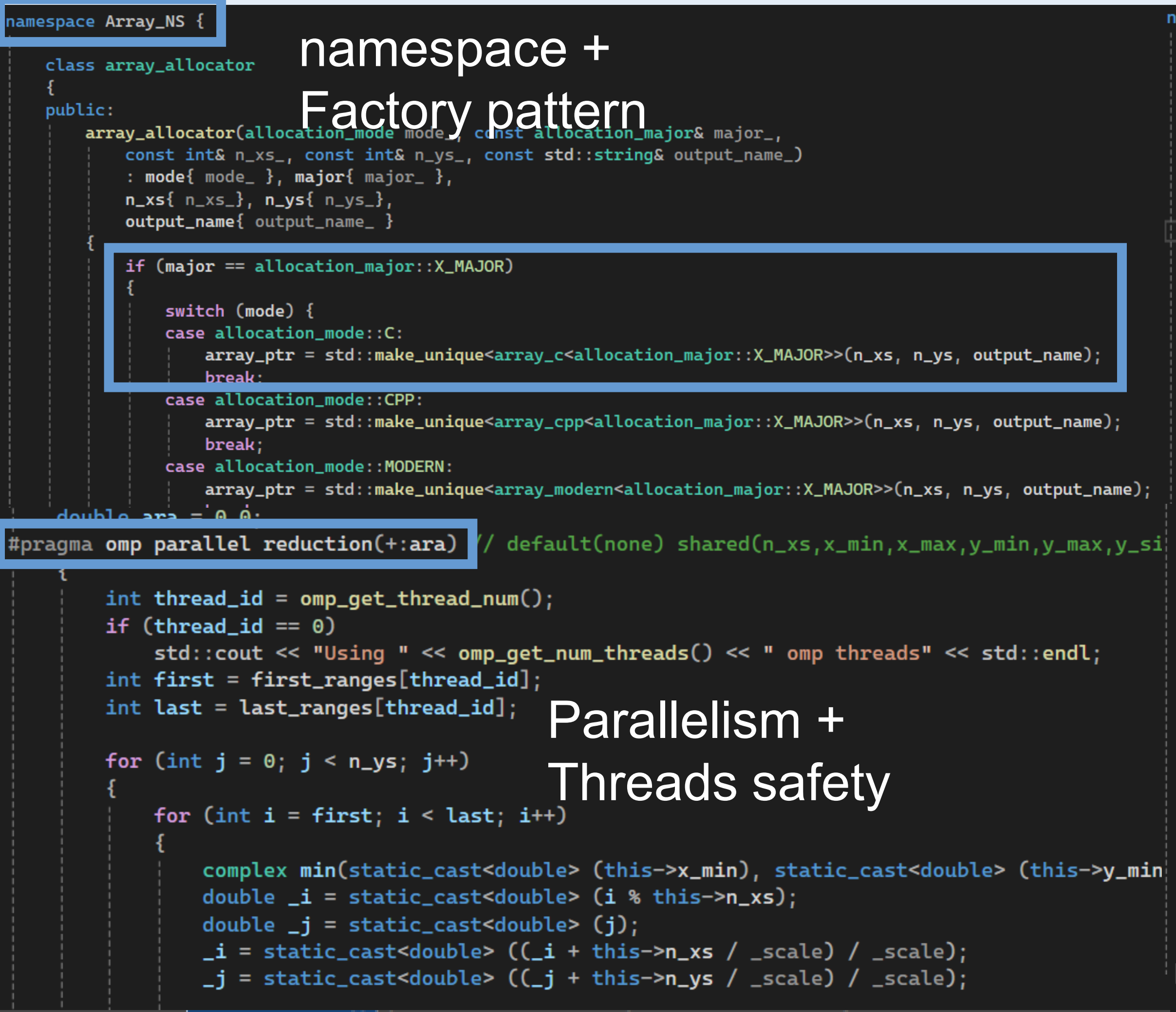
Project #4: Mandelbrot Parallel Renderer

A Modular C++ Framework for Parallel Simulation and Rendering

```
(1) Main.cpp
  |
  |── (2) Runner/ (run_mandelbrot_timing, run_mandelbrot_animation)
  |       └── Factory-like controller
  |
  |── (3) Algorithm/
  |       └── mandelbrot.h
  |              |── mandelbrot_xmesh(_inner/outerloop).h (inheritance)
  |              └── mandelbrot_xmesh(_inner/outerloop).h
  |── (4) Array/
  |       |── array.h (Base interface)
  |       |       └── array_c / array_cpp / array_modern / array_mdspan
  |       └── array_allocator (RAII factory)
  |
  |── (5) Numerical/
  |         └── complex.h (templates)
  |
  └── (6) Plotting/
           └──Mandelbrot_plot_frames.py
```

Input (1) – **Control Layer (2)** – **Core Engine (3)** – **Memory & Data Layer (4)** – **Numerical Library (5)** – **Post-processing (6)**

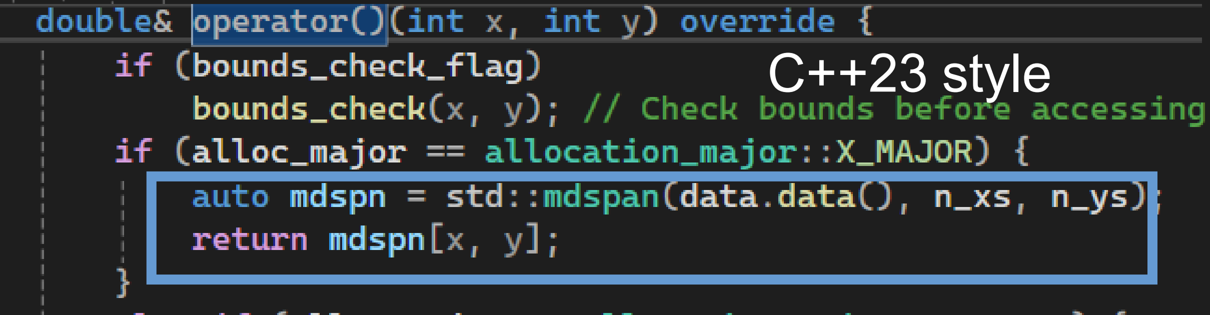
Project #4: Mandelbrot Parallel Renderer



namespace + Factory pattern

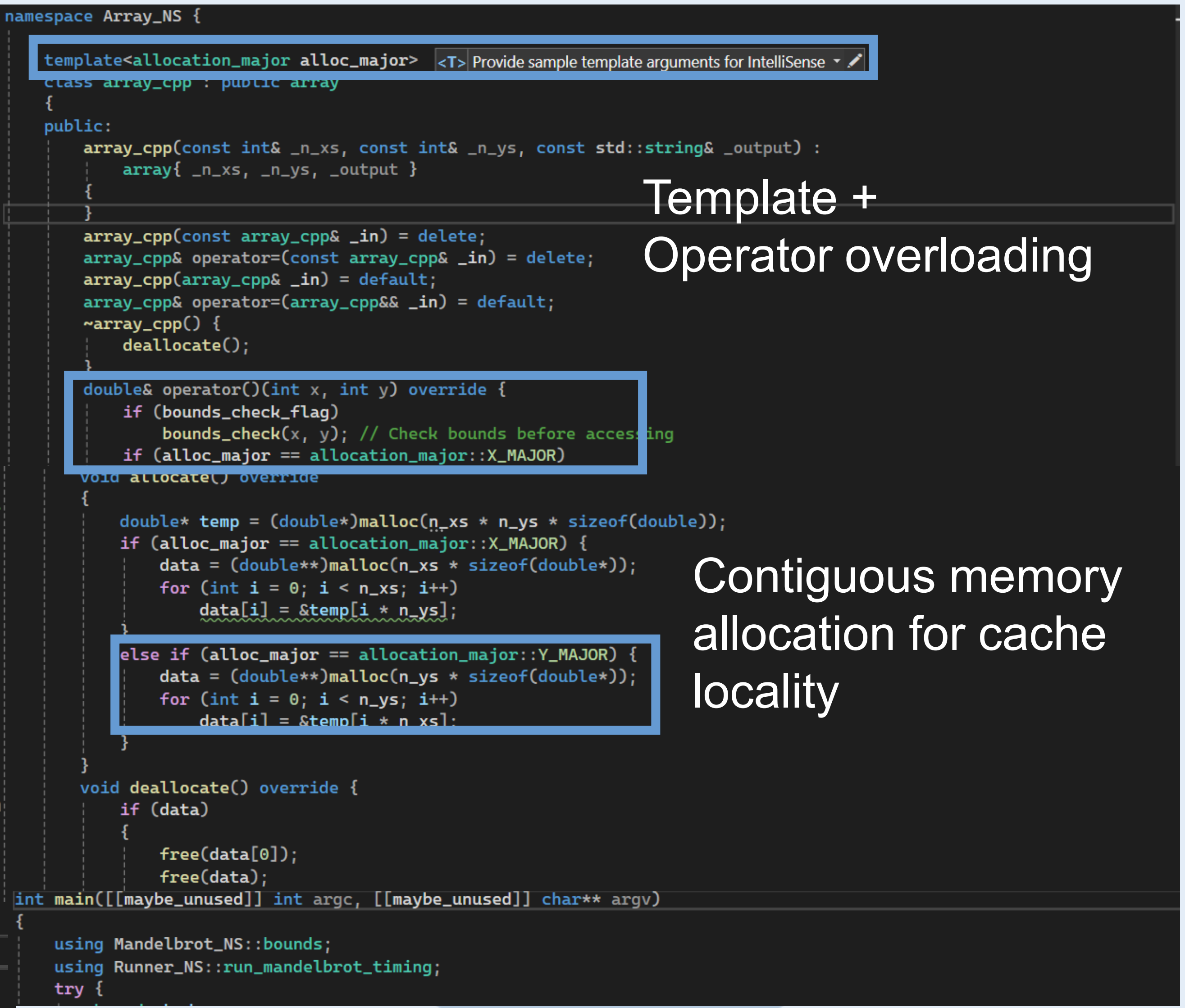
Parallelism + Threads safety

```cpp
namespace Array_NS {

    class array_allocator
    {
    public:
        array_allocator(allocation_mode mode_, const allocation_major& major_,
            const int& n_xs_, const int& n_ys_, const std::string& output_name_)
            : mode{ mode_ }, major{ major_ },
            n_xs{ n_xs_ }, n_ys{ n_ys_},
            output_name{ output_name_ }
        {
            if (major == allocation_major::X_MAJOR)
            {
                switch (mode) {
                case allocation_mode::C:
                    array_ptr = std::make_unique<array_c<allocation_major::X_MAJOR>>(n_xs, n_ys, output_name);
                    break;
                case allocation_mode::CPP:
                    array_ptr = std::make_unique<array_cpp<allocation_major::X_MAJOR>>(n_xs, n_ys, output_name);
                    break;
                case allocation_mode::MODERN:
                    array_ptr = std::make_unique<array_modern<allocation_major::X_MAJOR>>(n_xs, n_ys, output_name);
    double ara = 0.0;
#pragma omp parallel reduction(+:ara) // default(none) shared(n_xs,x_min,x_max,y_min,y_max,y_si
    {
        int thread_id = omp_get_thread_num();
        if (thread_id == 0)
            std::cout << "Using " << omp_get_num_threads() << " omp threads" << std::endl;
        int first = first_ranges[thread_id];
        int last = last_ranges[thread_id];

        for (int j = 0; j < n_ys; j++)
        {
            for (int i = first; i < last; i++)
            {
                complex min(static_cast<double> (this->x_min), static_cast<double> (this->y_min
                double _i = static_cast<double> (i % this->n_xs);
                double _j = static_cast<double> (j);
                _i = static_cast<double> ((_i + this->n_xs / _scale) / _scale);
                _j = static_cast<double> ((_j + this->n_ys / _scale) / _scale);
```



C++23 style

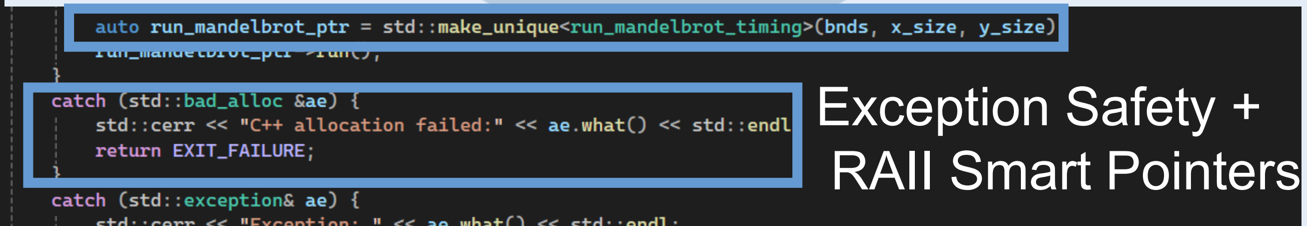
```cpp
double& operator()(int x, int y) override {
    if (bounds_check_flag)
        bounds_check(x, y); // Check bounds before accessing
    if (alloc_major == allocation_major::X_MAJOR) {
        auto mdspn = std::mdspan(data.data(), n_xs, n_ys);
        return mdspn[x, y];
    }
```



Template + Operator overloading

Contiguous memory allocation for cache locality

```cpp
namespace Array_NS {

    template<allocation_major alloc_major>   <T> Provide sample template arguments for IntelliSense
    class array_cpp : public array
    {
    public:
        array_cpp(const int& _n_xs, const int& _n_ys, const std::string& _output) :
            array{ _n_xs, _n_ys, _output }
        {
        }
        array_cpp(const array_cpp& _in) = delete;
        array_cpp& operator=(const array_cpp& _in) = delete;
        array_cpp(array_cpp& _in) = default;
        array_cpp& operator=(array_cpp&& _in) = default;
        ~array_cpp() {
            deallocate();
        }
        double& operator()(int x, int y) override {
            if (bounds_check_flag)
                bounds_check(x, y); // Check bounds before accessing
            if (alloc_major == allocation_major::X_MAJOR)
        void allocate() override
        {
            double* temp = (double*)malloc(n_xs * n_ys * sizeof(double));
            if (alloc_major == allocation_major::X_MAJOR) {
                data = (double**)malloc(n_xs * sizeof(double*));
                for (int i = 0; i < n_xs; i++)
                    data[i] = &temp[i * n_ys];
            }
            else if (alloc_major == allocation_major::Y_MAJOR) {
                data = (double**)malloc(n_ys * sizeof(double*));
                for (int i = 0; i < n_ys; i++)
                    data[i] = &temp[i * n_xs];
            }
        }
        void deallocate() override {
            if (data)
            {
                free(data[0]);
                free(data);
int main([[maybe_unused]] int argc, [[maybe_unused]] char** argv)
{
    using Mandelbrot_NS::bounds;
    using Runner_NS::run_mandelbrot_timing;
    try {
```



Exception Safety + RAII Smart Pointers

```cpp
        auto run_mandelbrot_ptr = std::make_unique<run_mandelbrot_timing>(bnds, x_size, y_size)
        run_mandelbrot_ptr->run();
    }
    catch (std::bad_alloc &ae) {
        std::cerr << "C++ allocation failed:" << ae.what() << std::endl
        return EXIT_FAILURE;
    }
    catch (std::exception& ae) {
        std::cerr << "Exception: " << ae.what() << std::endl;
```

Project #5: Neural Network: A Modular C++ Framework for Training Feed-Forward Nets

(1) Input/Configuration

❖ main.cpp	
❖ InputArgs.h	--- Parsing CLI
❖ InputFile.h	--- CSV reader
❖ InputFileMPI.h	--- Parallel CSV reader

(2) Data Processing & Logging

❖ Scaler.h	--- MinMaxScaler,…
❖ Logger.h	--- log to cout, log file, multi-stream

(3) Core Engine

❖ NeuralNetwork.h	--- Core training/Feed-forward
❖ NeuralNetworkMPI.h	--- MPI accelerator
❖ NeuralNetworkOpenMP.h	---OpenMP accelerator

(4) Model Components

❖ Layer.h	--- Layers
❖ Activation.h	--- Relu/Sigmoid/Tanh
❖ Dropout.h	--- Regularization

(5) Numerical Library

- ❖ Eigen

(6) Optimization

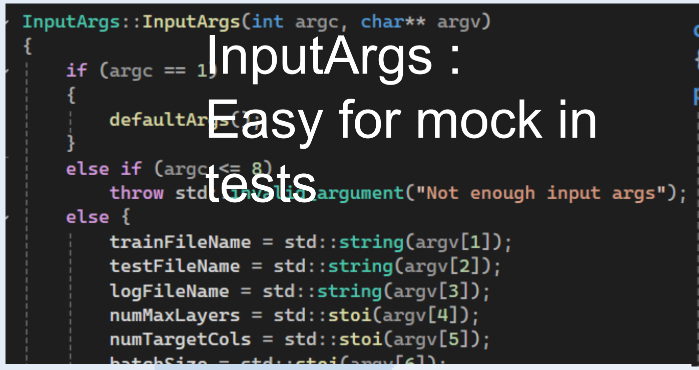
❖ Loss.h	--- MSE, MAE, Huber
❖ Optimizers.h	--- SGD, RMSProp, …

(7) Builds and tests

❖ CMakeLists.txt	--- Debug/release, MPI/OpenMP
❖ tests/	--- regression test with catch

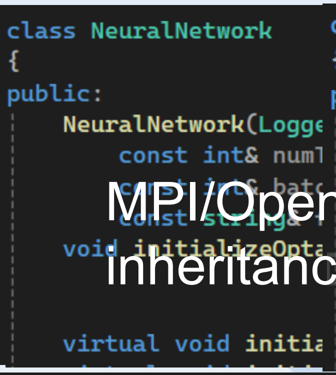
Input (1) ➔ Data (2) ➔ Core/Model (3-4) ➔ Math/Opt (5-6) ➔ Builds/tests (7)

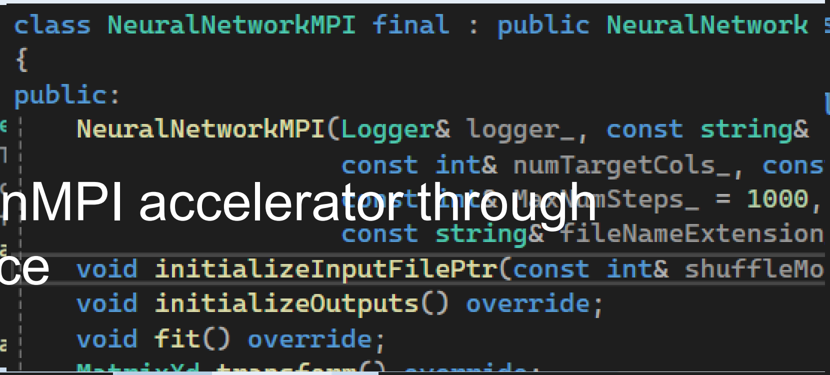
Project #5: Neural Network

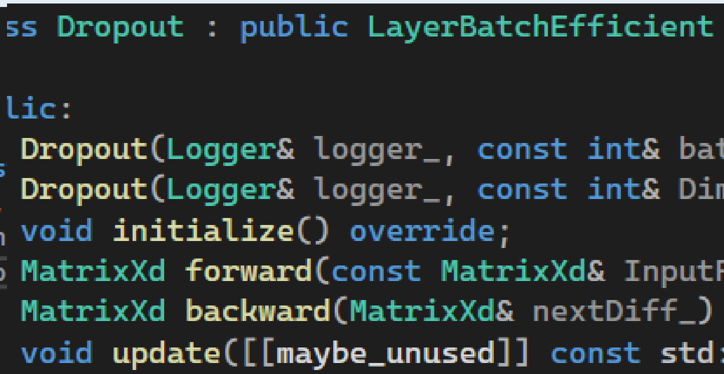


InputArgs : Easy for mock in tests

```
InputArgs::InputArgs(int argc, char** argv)
{
    if (argc == 1)
    {
        defaultArgs();
    }
    else if (argc <= 8)
        throw std::invalid_argument("Not enough input args");
    else {
        trainFileName = std::string(argv[1]);
        testFileName = std::string(argv[2]);
        logFileName = std::string(argv[3]);
        numMaxLayers = std::stoi(argv[4]);
        numTargetCols = std::stoi(argv[5]);
```





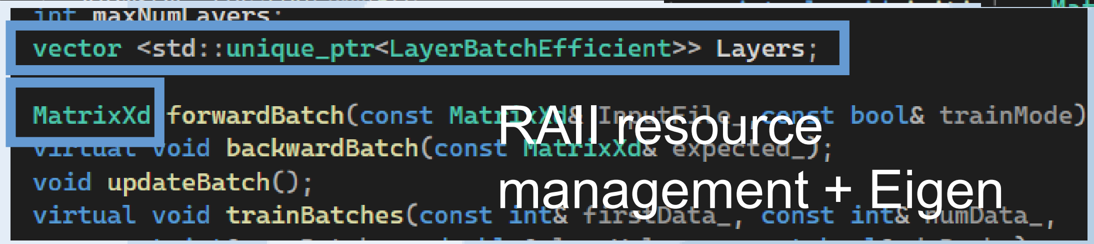


MPI/OpenMPI accelerator through inheritance

```
class NeuralNetwork
{
public:
    NeuralNetwork(Logge
        const int& numT
        const int& batc
        const string& 
    void initializeOpta

    virtual void initia
```

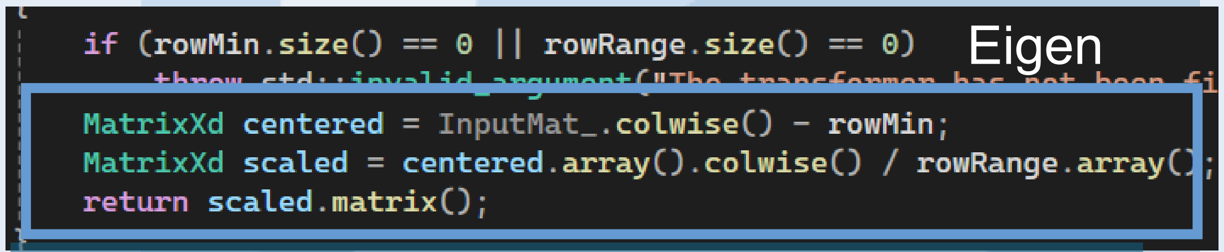
```
class NeuralNetworkMPI final : public NeuralNetwork
{
public:
    NeuralNetworkMPI(Logger& logger_, const string&
                     const int& numTargetCols_, cons
                     const int& MaxNumSteps_ = 1000,
                     const string& fileNameExtension
    void initializeInputFilePtr(const int& shuffleMo
    void initializeOutputs() override;
    void fit() override;
```

```
ss Dropout : public LayerBatchEfficient

lic:
 Dropout(Logger& logger_, const int& bat
 Dropout(Logger& logger_, const int& Dim
 void initialize() override;
 MatrixXd forward(const MatrixXd& InputF
 MatrixXd backward(MatrixXd& nextDiff_)
 void update([[maybe_unused]] const std:
```



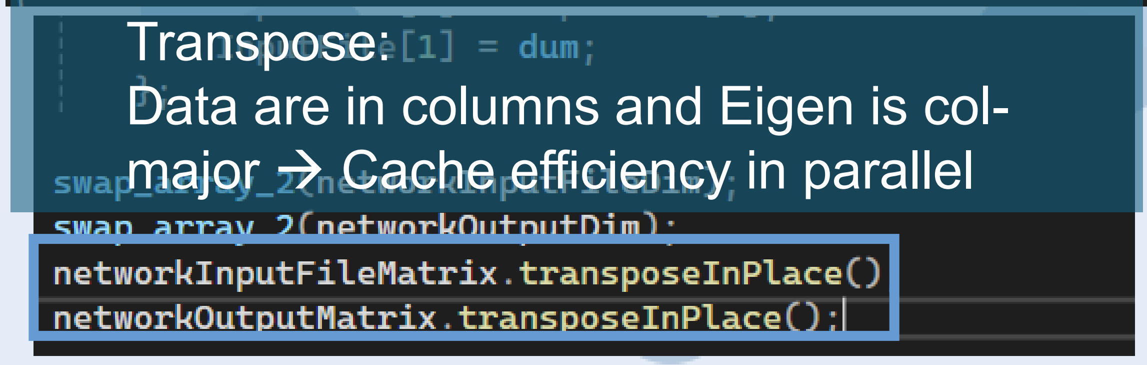
RAII resource management + Eigen

```
int maxNumLayers;
vector <std::unique_ptr<LayerBatchEfficient>> Layers;

MatrixXd forwardBatch(const MatrixXd& InputFile, const bool& trainMode)
virtual void backwardBatch(const MatrixXd& expected_);
void updateBatch();
virtual void trainBatches(const int& firstData_, const int& numData_,
```



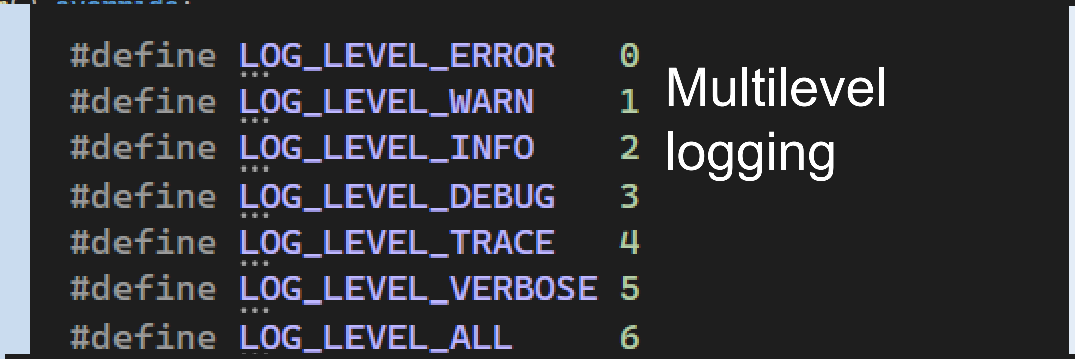
Eigen

```
if (rowMin.size() == 0 || rowRange.size() == 0)
    throw std::invalid_argument("The transformer has not been fi
MatrixXd centered = InputMat_.colwise() - rowMin;
MatrixXd scaled = centered.array().colwise() / rowRange.array();
return scaled.matrix();
```



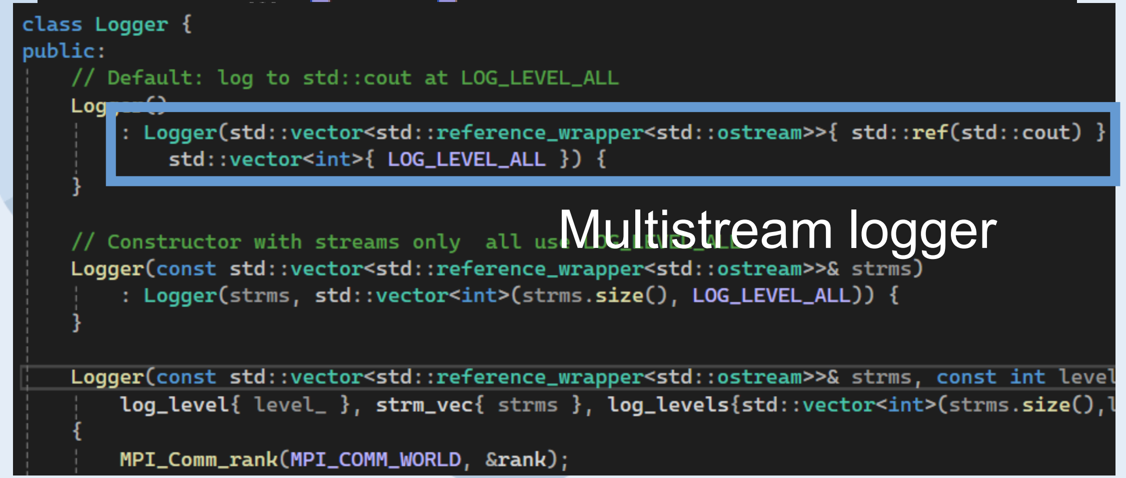
Transpose:
Data are in columns and Eigen is col-major → Cache efficiency in parallel

```
swap_array_2(networkInputFileDim);
swap_array_2(networkOutputDim);
networkInputFileMatrix.transposeInPlace()
networkOutputMatrix.transposeInPlace();
```



Multilevel logging

```
#define LOG_LEVEL_ERROR    0
#define LOG_LEVEL_WARN     1
#define LOG_LEVEL_INFO     2
#define LOG_LEVEL_DEBUG    3
#define LOG_LEVEL_TRACE    4
#define LOG_LEVEL_VERBOSE  5
#define LOG_LEVEL_ALL      6
```



Multistream logger

```
class Logger {
public:
    // Default: log to std::cout at LOG_LEVEL_ALL
    Logger()
        : Logger(std::vector<std::reference_wrapper<std::ostream>>{ std::ref(std::cout) }
          std::vector<int>{ LOG_LEVEL_ALL }) {
    }

    // Constructor with streams only  all use LOG_LEVEL_ALL
    Logger(const std::vector<std::reference_wrapper<std::ostream>>& strms)
        : Logger(strms, std::vector<int>(strms.size(), LOG_LEVEL_ALL)) {
    }

    Logger(const std::vector<std::reference_wrapper<std::ostream>>& strms, const int level
        log_level{ level_ }, strm_vec{ strms }, log_levels{std::vector<int>(strms.size(),l
    {
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

Modern C++ Features: RAII · Inheritance · Eigen · Parallel I/O · Logging Design – Test Design

Summary and Discussions

- ❖ Designed and implemented multiple C++17/20 scientific frameworks using RAII, Eigen, OpenMP, MPI and CUDA.
- ❖ Focused on numerical stability, deterministic parallelism and cache-efficient computation.
- ❖ Modular, testable and cross-compiler code with unit testing.
- ❖ Applied design patterns (Factory, Interface, Strategy) for extensibility.