A. Appendix A

A.1 src/pybind11_interface.cxx

```
#ifdef PYBIND11


#include "NISTfit/abc.h"
#include "NISTfit/optimizers.h"
#include "NISTfit/numeric_evaluators.h"
#include "NISTfit/examples.h"

#include <pybind11/pybind11.h>
#include <pybind11/stl.h>
#include <pybind11/functional.h>
#include <pybind11/eigen.h>

namespace py = pybind11;

using namespace NISTfit;

std::vector<double> time_LevenbergMarquardt(std::shared_ptr<AbstractEvaluator> &E,
     LevenbergMarquardtOptions &options, long Nrepeats)
{
    std::vector<double> times;
    for (auto i = 0; i < Nrepeats; ++i){
        auto startTime = std::chrono::high_resolution_clock::now();
        LevenbergMarquardt(E, options);
        auto endTime = std::chrono::high_resolution_clock::now();
        times.push_back(std::chrono::duration<double>(endTime - startTime).count()
            );
    }
    return times;
}

double fit_decaying_exponential(bool threading, std::size_t Nmax, short Nthreads,
    long N, long Nrepeat)
{
    double a = 0.2, b = 3, c = 1.3;
    std::vector<std::shared_ptr<AbstractOutput> > outputs;
    for (double i = 0; i < Nmax; ++i) {
        double x = i / ((double)Nmax);
        double y = exp(-a*x)*sin(b*x)*cos(c*x);
        auto in = std::shared_ptr<NumericInput>(new NumericInput(x, y));
        outputs.push_back(std::shared_ptr<AbstractOutput>(new
            DecayingExponentialOutput(N, in)));
    }
    std::shared_ptr<AbstractEvaluator> eval(new NumericEvaluator());
    eval->add_outputs(outputs);

    std::vector<double> c0 = { 1, 1, 1 };
    auto startTime = std::chrono::system_clock::now();
    auto opts = LevenbergMarquardtOptions();
    opts.c0 = c0; opts.threading = threading; opts.Nthreads = Nthreads;
    eval->set_coefficients(opts.c0);
    if(threading){
        for (auto jj  = 0; jj < Nrepeat; ++jj){
            eval->evaluate_parallel(Nthreads);
```

```
        }
    }
    else{
        for (auto jj  = 0; jj < Nrepeat; ++jj){
            eval->evaluate_serial(0,eval->get_outputs_size(),0);
        }
    }
    auto endTime = std::chrono::system_clock::now();
    return std::chrono::duration<double>(endTime - startTime).count();
}

void init_fitter(py::module &m){


    class PyFiniteDiffOutput : public FiniteDiffOutput {
    public:
        /* Inherit the constructors */
        using FiniteDiffOutput::FiniteDiffOutput;

        /* Trampoline (need one for each virtual function) */
        double call_func(const std::vector<double> &c) override {
            /* Release the GIL */
            py::gil_scoped_release release;
            {
                /* Acquire GIL before calling Python code */
                py::gil_scoped_acquire acquire;

                PYBIND11_OVERLOAD(
                    double,                         /* Return type */
                    FiniteDiffOutput,               /* Parent class */
                    call_func,                      /* Name of function in C++ (must
                        match Python name) */
                    c                               /* Argument(s) */
                );
            }
        }
    };

    py::class_<AbstractOutput, std::shared_ptr<AbstractOutput> >(m, "
        AbstractOutput")
        .def("get_error", &AbstractOutput::get_error)
        ;

    py::class_<PolynomialOutput, AbstractOutput, std::shared_ptr<PolynomialOutput>
         >(m, "PolynomialOutput")
        .def(py::init<std::size_t, const std::shared_ptr<NumericInput> &>())
        ;

    py::class_<DecayingExponentialOutput, AbstractOutput, std::shared_ptr<
        DecayingExponentialOutput> >(m, "DecayingExponentialOutput")
        .def(py::init<int, const std::shared_ptr<NumericInput> &>())
        ;

    py::class_<FiniteDiffOutput, AbstractOutput, PyFiniteDiffOutput /* trampoline
        */, std::shared_ptr<FiniteDiffOutput> >(m, "FiniteDiffOutput")
        .def(py::init<const std::shared_ptr<NumericInput> &,
                      const std::function<double(const std::vector<double> &)>,
                      const std::vector<double> &
```

```
                        >())
        ;

    py::class_<AbstractEvaluator, std::shared_ptr<AbstractEvaluator>>(m, "
        AbstractEvaluator")
        .def("evaluate_serial", &AbstractEvaluator::evaluate_serial)
        .def("evaluate_parallel", &AbstractEvaluator::evaluate_parallel)
        .def("get_outputs_size", &AbstractEvaluator::get_outputs_size)
        .def("get_times", &AbstractEvaluator::get_times)
        .def("time_evaluate_parallel", &AbstractEvaluator::time_evaluate_parallel)
        .def("time_evaluate_serial", &AbstractEvaluator::time_evaluate_serial)
        .def("add_outputs", &AbstractEvaluator::add_outputs)
        .def("get_error_vector", &AbstractEvaluator::get_error_vector, py::
            return_value_policy::copy)
        .def("get_affinity_scheme", &AbstractEvaluator::get_affinity_scheme)
        .def("set_affinity_scheme", &AbstractEvaluator::set_affinity_scheme)
        ;

    py::class_<NumericEvaluator, AbstractEvaluator, std::shared_ptr<
        NumericEvaluator> >(m, "NumericEvaluator")
        .def(py::init<>())
        .def("set_coefficients", &NumericEvaluator::set_coefficients)
        ;

    py::class_<NumericInput, std::shared_ptr<NumericInput> >(m, "NumericInput")
        .def(py::init<double, double>())
        .def("x", &NumericInput::x)
        .def("y", &NumericInput::y);

    py::class_<LevenbergMarquardtOptions>(m, "LevenbergMarquardtOptions")
        .def(py::init<>())
        .def_readwrite("c0", &LevenbergMarquardtOptions::c0)
        .def_readwrite("threading", &LevenbergMarquardtOptions::threading)
        .def_readwrite("Nthreads", &LevenbergMarquardtOptions::Nthreads)
        .def_readwrite("omega", &LevenbergMarquardtOptions::omega)
        .def_readwrite("tau0", &LevenbergMarquardtOptions::tau0)
        ;

    m.def("LevenbergMarquardt", &LevenbergMarquardt, "Fit");
    m.def("time_LevenbergMarquardt", &time_LevenbergMarquardt);
    m.def("fit_decaying_exponential", &fit_decaying_exponential);

    m.def("Eigen_nbThreads", [](){ return Eigen::nbThreads(); });
    m.def("Eigen_setNbThreads", [](int Nthreads) { return Eigen::setNbThreads(
        Nthreads); });
}

PYBIND11_PLUGIN(NISTfit) {
    py::module m("NISTfit", "NISTfit module");

    init_fitter(m);

    return m.ptr();
}

#endif
```

A.2 include/NISTfit/abc.h

```
#ifndef NISTFIT_ABC_
#define NISTFIT_ABC_

#include <vector>
#include <future>
#include <memory>
#include <iterator>
#include <iostream>
#include <numeric>      // std::accumulate
#include <queue>

#if defined(_WIN32)
#define WIN32_LEAN_AND_MEAN
#define NOMINMAX
#include <windows.h>
#undef NOMINMAX
#undef WIN32_LEAN_AND_MEAN
#else
#include <pthread.h>
#endif

#include "Eigen/Dense"

#include "ThreadPool.h"

namespace NISTfit{

    // Forward definitions
    struct ThreadData;
    class AbstractEvaluator;

    /// The abstract base class for the inputs
    class AbstractInput {
    };

    /// The abstract base class for the outputs
    class AbstractOutput{
    private:
        /// The AbstractEvaluator that this output is linked with.  This pointer *
            MUST* be set
        /// when the AbstractOutput (or derived class thereof) is added to the
            AbstractEvaluator.
        /// The AbstractEvaluator::add_output() function takes case of this
            automatically.
        AbstractEvaluator *m_evaluator;
    public:
        virtual double get_error() const = 0;
        virtual const std::vector<double> &get_Jacobian_row() const { throw std::
            exception(); };
        /// Evaluate one input and cache the output variables internally
        /// This class should already be holding a pointer to the input to which
            it is connected
        virtual void evaluate_one() = 0;
        /// Return the linked input state
        virtual AbstractInput & get_input() const = 0;
        /// A pure-virtual function that is used to handle ANY exception that is
```

```
            caught in the
        /// evaluate_one function.  You might want to consider re-throwing the
            exception in the function
        /// and then setting an error flag/message, etc.
        virtual void exception_handler() = 0;
        /// Get the pointer to the AbstractEvaluator linked with this output
        virtual const AbstractEvaluator & get_AbstractEvaluator() const { return *
            m_evaluator; }
        /// Set the pointer to the AbstractEvaluator linked with this output
        virtual void set_AbstractEvaluator(AbstractEvaluator *evaluator) {
            m_evaluator = evaluator; }
    };

    /// The abstract base class for the evaluator
    class AbstractEvaluator
    {
    private:
        std::vector<std::shared_ptr<AbstractOutput> > m_outputs;
        std::vector<int> m_affinity_scheme; ///< A vector of processor indices
            that shall be used for each thread spun up, 0-based
        std::unique_ptr<ThreadPool> m_pool; ///< A ThreadPool of threads
        std::vector<double> m_times; ///< Elapsed times for each parallel payload
    protected:
        Eigen::MatrixXd J;
        Eigen::VectorXd r;

        /**
        * @brief Setup the threads that will be used to do the evaluations
        *
        */
        void setup_threads(short Nthreads) {

            if (!m_pool || Nthreads != m_pool->get_threads().size()){
                // Make a thread pool for the workers
                m_pool = std::unique_ptr<ThreadPool>(new ThreadPool(Nthreads));
                m_times.resize(Nthreads);

                // Set the thread affinity if desired
#if defined(WIN32)
                auto &threads = m_pool->get_threads();
                for (long i = 0; i < Nthreads; ++i) {
                    std::thread &td = threads[i];
                    if (!m_affinity_scheme.empty() && i <= m_affinity_scheme.size
                        ()) {
                        // See http://stackoverflow.com/a/41574964/1360263
                        auto affinity_mask = (static_cast<DWORD_PTR>(1) <<
                            m_affinity_scheme[i]); //core number starts from 0
                        SetThreadAffinityMask(td.native_handle(), affinity_mask);
                    }
                }
#else
                // See https://github.com/eliben/code-for-blog/blob/master/2016/
                    threads-affinity/set-affinity.cpp (code in public domain)
                // Create a cpu_set_t object representing a set of CPUs. Clear it
                    and mark
                // only CPU i as set.
                if (!m_affinity_scheme.empty()) {
                    auto &threads = m_pool->get_threads();
```

```
                    for (long i = 0; i < Nthreads; ++i) {
                        cpu_set_t cpuset;
                        CPU_ZERO(&cpuset);
                        CPU_SET(m_affinity_scheme[i], &cpuset);
                        int rc = pthread_setaffinity_np(threads[i].native_handle()
                            ,
                                                        sizeof(cpu_set_t), &cpuset
                                                            );
                        if (rc != 0) {
                          std::cerr << "Error calling pthread_setaffinity_np: " <<
                              rc << "\n";
                        }
                    }
                }
#endif
            }
        };
        /**
        * @brief Kill the threads that have been spun up to do the evaluations
        *
        * This function is called when AbstractEvaluator is destroyed
        */
        void kill_threads() {
            if (m_pool){
                m_pool->JoinAll();
                m_pool.release();
            }
        };
        /// Add a single output to the list of outputs and connect pointer to
            AbstractEvaluator
        void add_output(const std::shared_ptr<AbstractOutput> &out) {
            m_outputs.push_back(out);
            m_outputs.back()->set_AbstractEvaluator(this);
        }
    public:
        virtual void set_coefficients(const std::vector<double> &) = 0;
        virtual const std::vector<double> & get_const_coefficients() const = 0;
        /// Get the size of the outputs
        std::size_t get_outputs_size() { return m_outputs.size(); };
        /// Add a vector of instances derived from AbstractOutput to this
            evaluator
        void add_outputs(const std::vector<std::shared_ptr<AbstractOutput> > &
            outputs) {
            for (auto &out : outputs) {
                add_output(out);
            }
        }
        /// Get a reference to the vector of outputs
        std::vector<std::shared_ptr<AbstractOutput> > & get_outputs() { return
            m_outputs; };
        /// Destructor
        ~AbstractEvaluator() {
            // auto startTime = std::chrono::system_clock::now();
            kill_threads();
            // auto endTime = std::chrono::system_clock::now();
            // double thread_kill_elap = std::chrono::duration<double>(endTime -
                startTime).count();
            //std::cout << "thread teardown:" << thread_kill_elap << " s\n";
```

```
    }
        std::vector<double> get_times(){ return m_times; }

    /**
     * @brief Evaluate the residual function in serial operation for the input
          vector indices in the range [iInputStart, iInputStop)
     * @param iInputStart The starting index (included in the output)
     * @param iInputStop The stopping index (NOT included in the output) (a la
          Python)
     * @param iOutputStart The starting index for where the outputs should be
          placed (probably equal to iInputStart)
     */
    void evaluate_serial(std::size_t iInputStart, std::size_t iInputStop, std
        ::size_t iOutputStart) const {
        int Nrepeat = 1;
        for (int rep = 0; rep < Nrepeat; ++rep) {
            std::size_t j = iOutputStart;
            for (std::size_t i = iInputStart; i < iInputStop; ++i) {
                try{
                    m_outputs[j]->evaluate_one();
                }
                catch(...){
                    m_outputs[j]->exception_handler();
                }
                j++;
            }
        }
    };

    // Return a vector of times for each repeat of calling evaluate_parallel
    std::vector<double> time_evaluate_parallel(short Nthreads, short Nrepeats)
        {
        std::vector<double> times;
        for (auto i = 0; i < Nrepeats; ++i){
            auto startTime = std::chrono::high_resolution_clock::now();
            evaluate_parallel(Nthreads);
            auto endTime = std::chrono::high_resolution_clock::now();
            times.push_back(std::chrono::duration<double>(endTime - startTime)
                .count());
        }
        return times;
    }
    // Return a vector of times for each repeat of calling evaluate_serial
    std::vector<double> time_evaluate_serial(short Nrepeats){
        std::vector<double> times;
        for (auto i = 0; i < Nrepeats; ++i){
            auto startTime = std::chrono::high_resolution_clock::now();
            evaluate_serial(0,get_outputs_size(),0);
            auto endTime = std::chrono::high_resolution_clock::now();
            times.push_back(std::chrono::duration<double>(endTime - startTime)
                .count());
        }
        return times;
    }

    /**
     * @brief Evaluate all the outputs in parallel
     * @param Nthreads The number of threads over which the calculations
```

```
            should be distributed
         */
        void evaluate_parallel(short Nthreads){

            // Set up threads but put them in holding pattern
            // no-op if threads are already initialized
            setup_threads(Nthreads);

            std::size_t Nmax = m_outputs.size();
            std::size_t Lchunk = Nmax / Nthreads;
            std::vector<std::size_t> chunksizes(Nthreads, Lchunk);
            auto remainder = Nmax-Lchunk*Nthreads;
            // Increase the first remainder chunk sizes
            for (auto i = 0; i < remainder; ++i){
                chunksizes[i]++;
            }
            assert(std::accumulate(chunksizes.begin(), chunksizes.end(),
                static_cast<std::size_t>(0)) == m_outputs.size());
            std::vector<double> times(Nthreads), summers(Nthreads);

            std::size_t isum = 0;
            for (auto i = 0; i < Nthreads; ++i)
            {
                auto cs = chunksizes[i];
                auto itStart = m_outputs.begin() + isum;
                auto itEnd = itStart + cs -1;// -1 because iEnd is NON-INCLUSIVE
                    !!!!!!!!!!
                isum += cs;
                double &elapsed = m_times[i];
                std::function<void(void)> f = [itStart, itEnd, &elapsed]() {
                    auto startTime = std::chrono::high_resolution_clock::now();
                    for (auto it = itStart; it != itEnd; ++it) {
                        try {
                            (*it)->evaluate_one();
                        }
                        catch (...) {
                            (*it)->exception_handler();
                        }
                    }
                    auto endTime = std::chrono::high_resolution_clock::now();
                    elapsed = std::chrono::duration<double>(endTime - startTime).
                        count();
                };
                m_pool->AddJob(f);
            }
            // Now we wait for all threads to finish
            m_pool->WaitAll();
        };

        /** @brief Construct the Jacobian matrix \f$\mathbf{J}\f$
         *
         * Each entry in the Jacobian matrix is given by
         * \f[ J_{ij} = \frac{\partial r_i}{\partial c_j} \f]
         * where \f$r_i\f$ is the \f$i\f$-th residue and \f$c_j\f$ is the \f$j\f$-
            th coefficient
         *
         * It is constructed by taking the rows of the Jacobian matrix stored in
            instances of AbstractOutput
```

```
         */
        const Eigen::MatrixXd &get_Jacobian_matrix() {
            std::size_t ncol = m_outputs[0]->get_Jacobian_row().size();
            J.resize(m_outputs.size(), ncol);
            int i = 0;
            for (auto &o : m_outputs) {
                const std::vector<double> &Jrow = o->get_Jacobian_row();
                Eigen::Map<const Eigen::VectorXd> Jrow_wrap(&Jrow[0], Jrow.size())
                    ;
                J.row(i) = Jrow_wrap;
                i++;
            }
            return J;
        };
        Eigen::MatrixXd build_Jacobian_matrix_numerically(double dx) {
            std::size_t ncol = m_outputs[0]->get_Jacobian_row().size();
            Eigen::MatrixXd Jfd(m_outputs.size(), ncol);

            // Initial values
            auto c0 = get_const_coefficients();
            set_coefficients(c0);
            evaluate_serial(0,m_outputs.size(),0);
            Eigen::VectorXd r0 = get_error_vector();

            // Iterate over the columns
            for (int icol = 0; icol < ncol; ++icol) {
                std::vector<double> c = c0;
                double dc = dx*c[icol];
                c[icol] += dc;
                set_coefficients(c);
                evaluate_serial(0, m_outputs.size(), 0);
                r = get_error_vector();
                Jfd.col(icol) = (r-r0)/dc;
            }
            return Jfd;
        };

        /** @brief Construct the residual vector of residuals for each data point
         *
         * \f[ r_i = (y_{\rm model} - y_{\rm given})_i\f]
         *
         * Internally, the AbstractOutput::get_error() function is called on each
             AbstractOutput managed by this evaluator.  One
         * of AbstractOutput::evaluate_serial() or AbstractOutput::
             evaluate_threaded() should have already been called before calling
             this function
         */
        const Eigen::VectorXd &get_error_vector() {
            r.resize(m_outputs.size());
            int i = 0;
            for (auto &o : m_outputs) {
                r(i) = o->get_error(); i++;
            }
            return r;
        }
        /**
         * @brief Set affinity scheme that is to be used to determine which thread
              is connected to which processor
```

```
         *
         * The indices in the vector are the indices for the first, second, third,
              etc. thread, 0-based.  For instance if you want
         * the affinity to go as the first thread on core 1, the first core on
              thread 2, etc., you might have: [0,2,4,6,1,3,5,7]
         */
        void set_affinity_scheme(const std::vector<int> &affinity_scheme){
            m_affinity_scheme = affinity_scheme; };
        /**
         * @brief Get the affinity scheme that is in use
         */
        const std::vector<int> & get_affinity_scheme() { return m_affinity_scheme;
             };
};

/// The data inputs
class NumericInput : public AbstractInput{
    protected:
        double m_x, m_y;
    public:
        NumericInput(double x, double y) : m_x(x), m_y(y) {};
        const double x() const { return m_x; };
        const double y() const { return m_y; };
};

/// The class for the evaluation of a single output value for a single input
    value
class NumericEvaluator : public AbstractEvaluator {
protected:
    std::vector<double> m_c;
public:
    void set_coefficients(const std::vector<double> &c){ m_c = c; };
    const std::vector<double> & get_const_coefficients() const { return m_c;
        };
};

/// The data structure for an output for the single y output variable
class NumericOutput : public AbstractOutput{
    protected:
        const std::shared_ptr<NumericInput> m_in;
        double m_y_calc;
        std::vector<double> Jacobian_row; // Partial derivative of calculated
            value with respect to each independent variable
    public:
        /// Copy constructor
        NumericOutput(const std::shared_ptr<NumericInput> &in) : m_in(in) {};
        /// Move constructor
        NumericOutput(const std::shared_ptr<NumericInput> &&in) : m_in(in) {};
        virtual double get_error() const override { return m_y_calc - m_in->y
            (); };
        virtual const std::vector<double> & get_const_coefficients() const {
            return get_AbstractEvaluator().get_const_coefficients(); }
        const std::vector<double> & get_Jacobian_row() const override  {
            return Jacobian_row; }
        AbstractInput & get_input() const override { return *static_cast<
            AbstractInput*>(m_in.get()); };
        void resize(std::size_t N){ Jacobian_row.resize(N); };
};
```

```
    /// The data structure for an output for the single y output variable
    class FiniteDiffOutput : public NumericOutput{

    protected:
        std::function<double(const std::vector<double> &)> m_f;
        std::vector<double> m_dc;
    public:
        /// Copy constructor w/ passed in model function
        FiniteDiffOutput(const std::shared_ptr<NumericInput> &in,
                         const std::function<double(const std::vector<double> &)>
                             &f,
                         const std::vector<double> &dc)
            : NumericOutput(in), m_f(f), m_dc(dc) {
                resize(dc.size());};
        /// Move constructor w/ passed in model function
        FiniteDiffOutput(const std::shared_ptr<NumericInput> &&in,
                         const std::function<double(const std::vector<double> &)>
                             &f,
                         const std::vector<double> &dc)
            : NumericOutput(in), m_f(f), m_dc(dc) {
                resize(dc.size());};
        virtual double call_func(const std::vector<double> &c){
            return m_f(c);
        }
        /// Evaluate the function, and the Jacobian row by numerical
            differentiation
        void evaluate_one() override{
            // Do the calculation
            const std::vector<double> &c = get_const_coefficients();
            m_y_calc = call_func(c);
            for (std::size_t i = 0; i < c.size(); ++i) {
                std::vector<double> cp = c, cm = c;
                cp[i] += m_dc[i]; cm[i] -= m_dc[i];
                Jacobian_row[i] = (call_func(cp) - call_func(cm))/(2*m_dc[i]);
            }
        };
        void exception_handler() override{ m_y_calc = 100000; }
    };

}; /* namespace NISTfit */

#endif
```

A.3 include/NISTfit/examples.h

```
#ifndef NISTFIT_EXAMPLES_
#define NISTFIT_EXAMPLES_

#include "NISTfit/abc.h"

namespace NISTfit{

/*
 * \brief The factorial function
 *
 * Not done efficiently, could certainly be improved, but the point is we want it
     to be slow!
 */
```

```
double factorial(int N){
    if (N == 0){ return 1; } // An identity
    double o = N; // output; as double to avoid overflow
    for (int i = N - 1; i > 0; --i) {
        o *= i;
    }
    return o;
}
/*
 * \brief The exponential function exp(x) expressed as series expansion
 *
 * This is an intentionally slow implementation of the exponential function, in
 * order to increase the amount of work per evaluation
 */
double exp_expansion(double x, int N) {
    if (N <= 0){ return exp(x); }
    double y = 0;
    for (int m=0; m < N; ++m){
        y += pow(x, m)/factorial(m);
    }
    return y;
}
double sin_expansion(double x, int N) {
    if (N <= 0){ return sin(x); }
    double y = 0;
    for (int m = 0; m < N; ++m) {
        y += pow(-1, m)*pow(x, 2*m+1)/factorial(2*m+1);
    }
    return y;
}
double cos_expansion(double x, int N) {
    if (N <= 0){ return cos(x); }
    double y = 0;
    for (int m = 0; m < N; ++m) {
        y += pow(-1, m)*pow(x, 2*m)/factorial(2*m);
    }
    return y;
}

class DecayingExponentialOutput : public NumericOutput {
protected:
    int N; ///< Order of Taylor series expansion
    std::vector<double> c_fix = {1,2,3};
public:
    DecayingExponentialOutput(int N,
                              const std::shared_ptr<NumericInput> &in)
        : NumericOutput(in), N(N) { resize(3); };
    /// In the highly unlikely case of an exception in this class,
    /// (implementation of this method is required), set the calculated value
    /// to something very large
    void exception_handler() override { m_y_calc = 100000; }
    //const std::vector<double> & get_const_coefficients() const override { return
        c_fix; };
    void evaluate_one() override {
        // Get a reference to the coefficients
        const std::vector<double> &c = get_const_coefficients();
        // Do the calculation
        const double x = m_in->x(), e = exp_expansion(-c[0]*x, N),
```

```
                    s1 = sin_expansion(c[1]*x, N), c2 = cos_expansion(c[2]*x, N);
        double y = e*s1*c2;
        Jacobian_row[0] = -x*y;
        Jacobian_row[1] = x*e*cos_expansion(c[1]*x,N)*c2;
        Jacobian_row[2] = -x*e*s1*sin_expansion(c[2]*x,N);
        m_y_calc = y;
    }
};

} /* namespace NISTfit */

#endif
```

A.4 include/NISTfit/optimizers.h

```
#ifndef NISTFIT_OPTIMIZERS_
#define NISTFIT_OPTIMIZERS_

#include <vector>

namespace NISTfit{

    /**
     * @brief A struct for holding options to be passed to LevenbergMarquardt
         function
     */
    struct LevenbergMarquardtOptions {

        std::vector<double> c0; ///< The initial coefficients that are being
            fitted
        bool threading = false; ///< True to use threaded evaluation, false for
            serial evaluation
        short Nthreads = -1; ///< Number of threads to use; -1 for std::thread::
            hardware_concurrency(), positive number otherwise
        double omega; ///< The relaxation to be applied to the step, in the range
            (0,infinity). A value of 1.0 is a full step from Levenberg-Marquardt.
             Making omega < 1 yields partial steps and makes the algorithm slower
            but more stable.
        double tau0; ///< Madsen et al. recommends setting tau0 to 1e-6 if the
            initial guess
                     ///< is believed to be a good estimate of the final solution,
                         or
                     ///< larger values like 1e-3 or 1 if the guess value is less
                        certain
        short debug_level = 0; ///< The higher this is, the more will be output to
             screen; 0 is no additional output

        LevenbergMarquardtOptions() : omega(1.0), tau0(1.0) {};
    };

    /**
     * @brief The Levenberg-Marquardt sum-of-squares minimizer
     * @param E The derived instance of AbstractEvaluator used to evaluate the
         terms in the sum-of-squares
     * @param options The options to be passed to this function
     */
    std::vector<double> LevenbergMarquardt(std::shared_ptr<AbstractEvaluator> &E,
        LevenbergMarquardtOptions &options);
```

```
} /* NISTfit */

#endif
```

A.5 src/optimizers.cpp

```
#include "NISTfit/abc.h"
#include "NISTfit/optimizers.h"
#include "Eigen/Dense"
#include <cfloat>

std::vector<double> NISTfit::LevenbergMarquardt(std::shared_ptr<AbstractEvaluator>
     &E,
                                                 LevenbergMarquardtOptions &options
                                                     )
{
    double F_previous = 8888888;
    double lambda = 1;
    double nu = 2;

    std::vector<double> c =options.c0;

    Eigen::Map<Eigen::VectorXd> c_wrap(&c[0], c.size());

    auto Nthreads = (options.Nthreads < 0) ? std::thread::hardware_concurrency() :
         options.Nthreads;

    for (int counter = 0; counter < 100; ++counter) {

        E->set_coefficients(c);

        if (options.threading){ // Check if threading
            E->evaluate_parallel(Nthreads); // Using threading
        }
        else{
            E->evaluate_serial(0, E->get_outputs_size(), 0); // Not using
                threading
        };
        const Eigen::MatrixXd &J = E->get_Jacobian_matrix();
        const Eigen::VectorXd &r = E->get_error_vector();
        const Eigen::MatrixXd A = J.transpose()*J;
        //printf("r(min,max,mean): %g %g %g\n", r.minCoeff(), r.maxCoeff(), r.mean
            ());

        // This is actually the sum of squares of the entries in the error vector
        double F = r.squaredNorm();
        if (counter == 0) {
            double maxDiag = A.diagonal().maxCoeff();
            // Madsen recommends setting tau0 to 1e-6 if the initial guess
            // is believed to be a good estimate of the final solution, or
            // larger values like 1e-3 or 1 if the guess value is less certain
            lambda = maxDiag*options.tau0;
        }

        // Levenberg-Marquardt with LHS*DELTAc = RHS
        const Eigen::MatrixXd LHS = A + lambda*A.diagonal().asDiagonal().
            toDenseMatrix();
        const Eigen::MatrixXd RHS = -J.transpose()*r;
```

```
        // Calculate the step
        const Eigen::VectorXd DELTAc = options.omega*LHS.colPivHouseholderQr().
            solve(RHS);
        // Take the step
        c_wrap += DELTAc;

        // Resize the step (See Madsen document)
        if (counter > 0) {
            double DELTAL = 0.5*DELTAc.transpose()*(lambda*DELTAc - J.transpose()*
                r);
            double rho = (F_previous - F) / DELTAL;

            // Madsen Eq. 2.21
            if (rho > 0) {
                lambda *= std::max(1.0/3.0, 1 - pow(2 * rho - 1, 3));
                nu = 2;
            }
            else {
                lambda *= nu;
                nu *= 2;
            }
        }
        if (options.debug_level > 0){
            printf("r(min,max,mean): %g %g %g; F: %g\n", r.minCoeff(), r.maxCoeff
                (), r.mean(), F);
            std::cout << c_wrap << std::endl;
        }

        // If the residual has stopped changing, stop, no sense to keep evaluating
        // with the same coefficients
        if (counter > 1 && std::abs(F / F_previous - 1) < 1e-10) {
            break;
        }

        // Copy the residual
        F_previous = F;

        // Check whether to stop
        if (F < DBL_EPSILON || std::abs(lambda) < DBL_EPSILON) {
            break;
        }
    }
    return c;
}
```

A.6 evaluators.py

```
import NISTfit
import numpy as np

def get_eval_poly(Npoints):
    x = np.linspace(0,1,Npoints)
    y = 1 + 2*x + 3*x**2 + 4*x**6
    order = 6
    outputs = [NISTfit.PolynomialOutput(order, NISTfit.NumericInput(_x, _y))
               for _x,_y in zip(x, y)]
    eva = NISTfit.NumericEvaluator()
    eva.add_outputs(outputs)
    return eva, [1.5]*(order+1)
```

```
def get_eval_decaying_exponential(Norder):
    a = 0.2; b = 3; c = 1.3;
    x = np.linspace(0, 2, 1200)
    y = np.exp(-a*x)*np.sin(b*x)*np.cos(c*x)
    outputs = [NISTfit.DecayingExponentialOutput(Norder, NISTfit.NumericInput(_x,
        _y))
               for _x,_y in zip(x, y)]
    eva = NISTfit.NumericEvaluator()
    eva.add_outputs(outputs)
    return eva, [0.5, 2, 0.8]

def get_eval_decaying_exponential_finite_diff(Norder):
    a = 0.2; b = 3; c = 1.3;
    x = np.linspace(0, 2, 1000)
    y = np.exp(-a*x)*np.sin(b*x)*np.cos(c*x)
    dc = [0.01]*3 # epsilon used for each coefficient in the finite difference

    outputs = []

    for _x, _y in zip(x, y):
        # def f(c, x):
        #     np.exp(-c[0]*self.x)*np.sin(c[1]*self.x)*np.cos(c[2]*self.x)
        # o = pf.FiniteDiffOutput(pf.NumericInput(_x, _y), functools.partial(f, x=
            _x),dc)

        class Output(pf.FiniteDiffOutput):
            def __init__(self, input, x):
                super(Output, self).__init__(input,lambda c: c[0]+c[1], dc)
                self.x = x

            def call_func(self, c):
                return np.exp(-c[0]*self.x)*np.sin(c[1]*self.x)*np.cos(c[2]*self.x
                    )

        o = Output(pf.NumericInput(_x, _y), _x)
        outputs.append(o)

    eva = pf.NumericEvaluator()
    eva.add_outputs(outputs)
    return eva, [0.5, 2, 0.8]
```

A.7 time_NISTfit.py

```
from __future__ import division, print_function

# Standard libraries (always available)
import json
import sys
import timeit

# Conda-installable packages
import numpy as np
import pandas
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt

# Common module with the generators
```

```
import evaluators
import NISTfit

def generate_results(get_eva, args, ofname, method = 'evaluate', Nthreads_max = 8,
                     affinity = True, Nrepeats = 100, Eigen_threads = True):

    o = NISTfit.LevenbergMarquardtOptions()
    o.tau0 = 1

    windows_scheme = list(range(0,Nthreads_max,2)) + list(range(1,Nthreads_max,2))
    linux_scheme = list(range(0,Nthreads_max))
    if sys.platform.startswith('win'):
        scheme = windows_scheme
    else:
        scheme = linux_scheme

    if affinity:
        affinity_options = [(True,()),(False,[2,2])]
    else:
        affinity_options = [(False,())]

    data = {'filename': ofname, 'times':[], 'Nthreads_max': Nthreads_max,
            'affinity': affinity, 'affinity_options': affinity_options,
            'args' : args}

    for arg in args:
        for affinity, affinity_dashes in affinity_options:
            print(arg,affinity)

            reject = int(0.0*Nrepeats)

            # Serial evaluation
            eva, o.c0 = get_eva(arg)
            if affinity:
                eva.set_affinity_scheme(scheme)
            eva.set_coefficients(o.c0)
            N = eva.get_outputs_size()
            if method == 'evaluate':
                times = eva.time_evaluate_serial(Nrepeats)
            elif method == 'LM':
                o.threading = False
                times = NISTfit.time_LevenbergMarquardt(eva, o, Nrepeats)
            else:
                raise ValueError("Bad method")
            elap = np.min(np.sort(times)[reject:Nrepeats-reject])

            data['times'].append(dict(arg=arg, type='serial', Nthreads = 0,
                                      time = elap, affinity = affinity))

            # Parallel evaluation
            o.threading = True
            Nthreads_list = range(1, Nthreads_max+1)
            for Nthreads in Nthreads_list:
                if Eigen_threads:
                    NISTfit.Eigen_setNbThreads(Nthreads)
                eva, o.c0 = get_eva(arg)
                o.Nthreads = Nthreads
                eva.set_coefficients(o.c0)
```

```
                if affinity:
                    eva.set_affinity_scheme(scheme)
                elap = None
                cfinal = eva.evaluate_parallel(Nthreads)
                if method == 'evaluate':
                    times = eva.time_evaluate_parallel(Nthreads, Nrepeats)
                elif method == 'LM':
                    times = NISTfit.time_LevenbergMarquardt(eva, o, Nrepeats)
                else:
                    raise ValueError("Bad method")
                elap = np.min(np.sort(times)[reject:Nrepeats-reject])

                data['times'].append(dict(arg=arg, type='parallel',
                                          Nthreads=Nthreads, time=elap,
                                          affinity=affinity))

    with open('timing-'+ofname+'.json','w') as fp:
        fp.write(json.dumps(data, indent =2))

def plot_results(ofname):

    with open('timing-'+ofname+'.json') as fp:
        _data = json.load(fp)
        Nthreads_max = _data['Nthreads_max']
        Nthreads_list = range(1, Nthreads_max+1)
        affinity_options = _data['affinity_options']
        args = _data['args']
        df = pandas.DataFrame(_data['times'])

    fig1, ax1 = plt.subplots(1,1,figsize=(4,3))
    fig2, ax2 = plt.subplots(1,1,figsize=(4,3))

    if np.sum(df.affinity)>0 or len(affinity_options) > 1:
        ax1.plot([2,2.9],[7,7],lw=1,color='grey')
        ax1.plot([2,2.9],[6,6],lw=1,color='grey',dashes = [2,2])
        ax1.text(3,7,'Affinity',ha='left',va='center')
        ax1.text(3,6,'No affinity',ha='left',va='center')

    for arg, c in zip(args,['b','r','c','k']):
        for affinity, dashes in affinity_options:

            # Extract data for this arg from the pandas DataFrame
            time_serial = float(df[(df.arg == arg) & (df.Nthreads == 0) & (df.
                affinity == affinity)].time)
            times = np.array(df[(df.arg == arg) & (df.Nthreads >= 1) & (df.
                affinity == affinity)].time)

            line, = ax1.plot(Nthreads_list,time_serial/np.array(times),
                             color=c,dashes=dashes)

            if arg < 0:
                lbl = 'native'
            else:
                lbl = 'N: '+str(arg)

            ax2.plot(Nthreads_list, np.array(times)/times[0],label = lbl,
                     color=c,dashes=dashes)
            if affinity or len(affinity_options) == 1:
```

```
                ax1.text(len(times)-0.25, (time_serial/np.array(times))[-1], lbl,
                         ha='right', va='center',
                         color=c,
                         bbox = dict(facecolor='w',
                                     edgecolor=line.get_color(),
                                     boxstyle='round')
                         )

    ax1.plot([1,Nthreads_max],[1,Nthreads_max],'k',lw=3,label='linear speedup')
    ax1.set_xlabel(r'$N_{\rm threads}$ (-)')
    ax1.set_ylabel(r'Speedup $t_{\rm serial}/t_{\rm parallel}$ (-)')
    fig1.tight_layout(pad=0.3)
    fig1.savefig(ofname + '.pdf', transparent = True)

    # NN = np.linspace(1,Nthreads_max)
    # ax2.plot(NN,1/NN,'k',lw=3,label='linear speedup')
    # ax2.set_xlabel(r'$N_{\rm threads}$ (-)')
    # ax2.set_ylabel(r'Total time $t_{\rm parallel}/t_{\rm 1 thread}$ (-)')
    # ax2.legend(loc='best',ncol=2)
    # fig2.tight_layout(pad=0.3)
    # fig2.savefig('abs-'+ofname + '.pdf', transparent = True)

    plt.close('all')

if __name__=='__main__':

    import argparse
    parser = argparse.ArgumentParser(description='Run the timing tests for NISTfit
        ')
    parser.add_argument('Nthreads_max', metavar='Nthreads_max', type=int, nargs=1,
         help="The maximum number of threads")
    parser.add_argument('--affinity-too', nargs='?', const=True, default=False,
        help="If defined, the affinity tests will also be run (windows only)")
    args = parser.parse_args()

    for method in ['evaluate','LM']:
        # Many fewer evaluations for LM than for normal evaluation
        divisor = 1
        if method == 'LM':
            divisor = 10
        ofname = method+'-speedup_polynomial'
        generate_results(evaluators.get_eval_poly, [120,12000], ofname,
                         Nthreads_max = args.Nthreads_max[0], method = method,
                         Nrepeats = 500//divisor, affinity = args.affinity_too)
        plot_results(ofname)

        ofname = method+'-speedup_decaying_exponential'
        generate_results(evaluators.get_eval_decaying_exponential, [50,5,-1],
                         ofname, Nthreads_max = args.Nthreads_max[0], method =
                             method,
                         Nrepeats = 200//divisor, affinity= args.affinity_too)
        plot_results(ofname)
```