Software and Tools for Computational Engineering

RWTH AACHEN UNIVERSITY

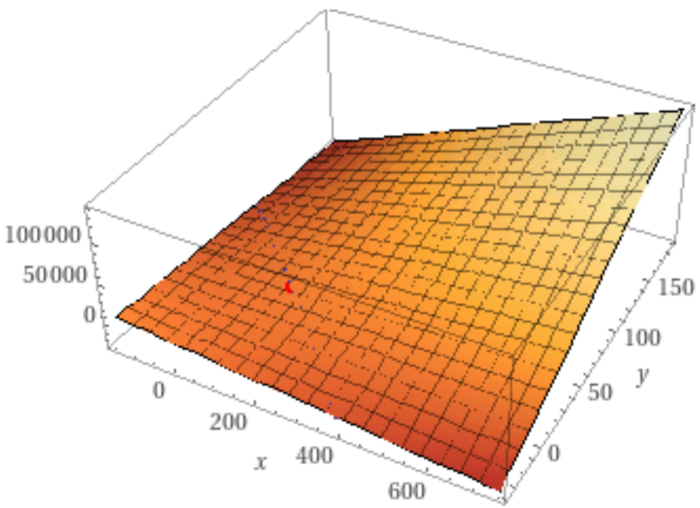
C++ UI design for NAG Optimization Modelling Suite

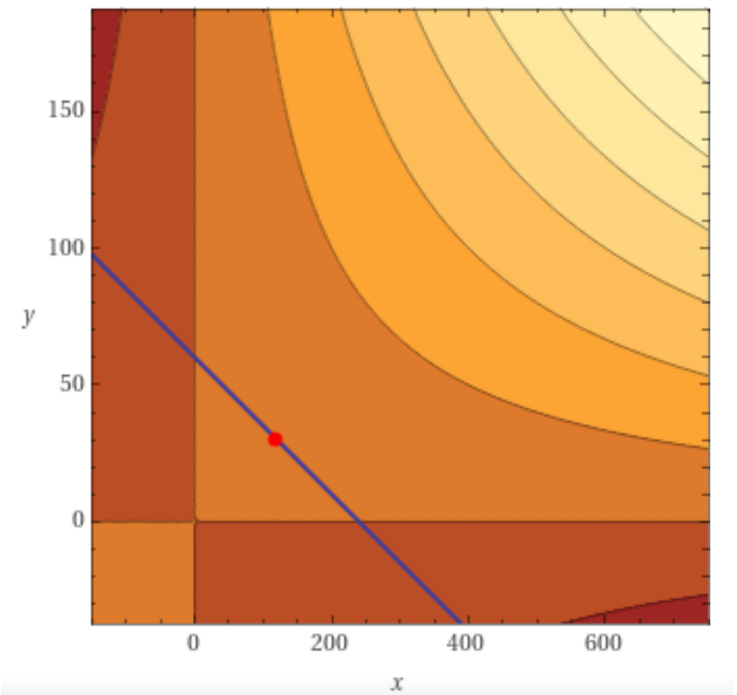
Group 6

Konstantin Korkin, Maksim Feldman, Tran Man Khang, Yifei Huang, Ziya Valiyev

Contents

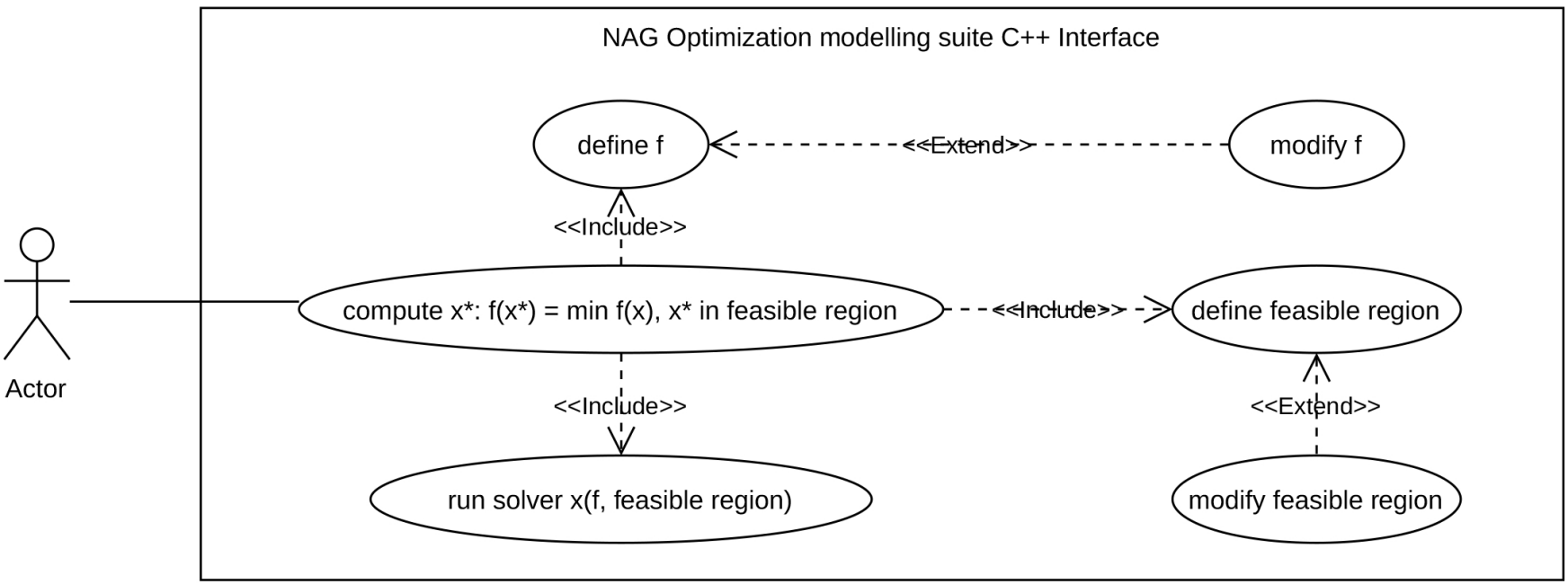
Maximize utility $f(x, y) = xy$ subject to the constraint $g(x, y) = x + y = 240$. Here the price of per unit x is 1, the price of y is 4 and the budget available to buy x and y is 240.





Users of the interface want to easily define and solve optimization problems with C++, including the ability to:

- define (and modify) the objective function
- define (and modify) the constraints
- find the minima/maxima of the objective function regarding the constraints without having to write code to find the derivative


NAG Optimization modelling suite C++ Interface
define f
<<Extend>>
modify f
<<Include>>
Actor
compute x*: f(x*) = min f(x), x* in feasible region
<<Include>>
define feasible region
<<Include>>
<<Extend>>
run solver x(f, feasible region)
modify feasible region

- The NAG Library is a comprehensive collection of routines for the solution of numerical and statistical problems, comparable to FICO Xpress or Gurobi.
- The NAG Optimization modelling suite is a part of the NAG Library, and is a suite of routines which allows the user to define and solve various optimization problems in a uniform manner.

Optimization problem is defined as:

$$\begin{aligned} \text{minimize} \quad & f(x) \\ \text{subject to} \quad & x \in F \end{aligned}$$

where x denotes the decision variables, f(x) the objective function and F the feasibility set. We assume that $F \subset \mathbb{R}^n$.

Optimization problem classification:

1. Type of objective function:
   - Linear: $f(x) = c^T x + c_0, c \in \mathbb{R}^n$
   - Non-linear: e.g. $f(x) = (1 - x_1)^2 + 100 \cdot (x_2 - {x_1}^2)^2$

2. Type of constraints:
   - Unconstrained
   - Simple bound: $l_x \leq x \leq u_x$, l_x and u_x are n-dimensional vectors.
   - Linear constraint: $l_B \leq B_x \leq u_B$, B is a general m_B x n rectangular matrix and l_x and u_x are n-dimensional vectors.
   - Non-linear constraint: $l_g \leq g(x) \leq u_g$
   - and more...

These combinations define specific optimization problems:

1. Linear programming problem: linear objective function, linear constraints and simple bounds.

$$\begin{aligned} \underset{x \in \mathbb{R}^n}{\text{minimize}} \quad & c^T x \\ \text{subject to} \quad & l_B \leq Bx \leq u_B, \\ & l_x \leq x \leq u_x. \end{aligned}$$

2. Nonlinear programming problem: general nonlinear objective function and any of the nonlinear, quadratic, linear or bound constraints.

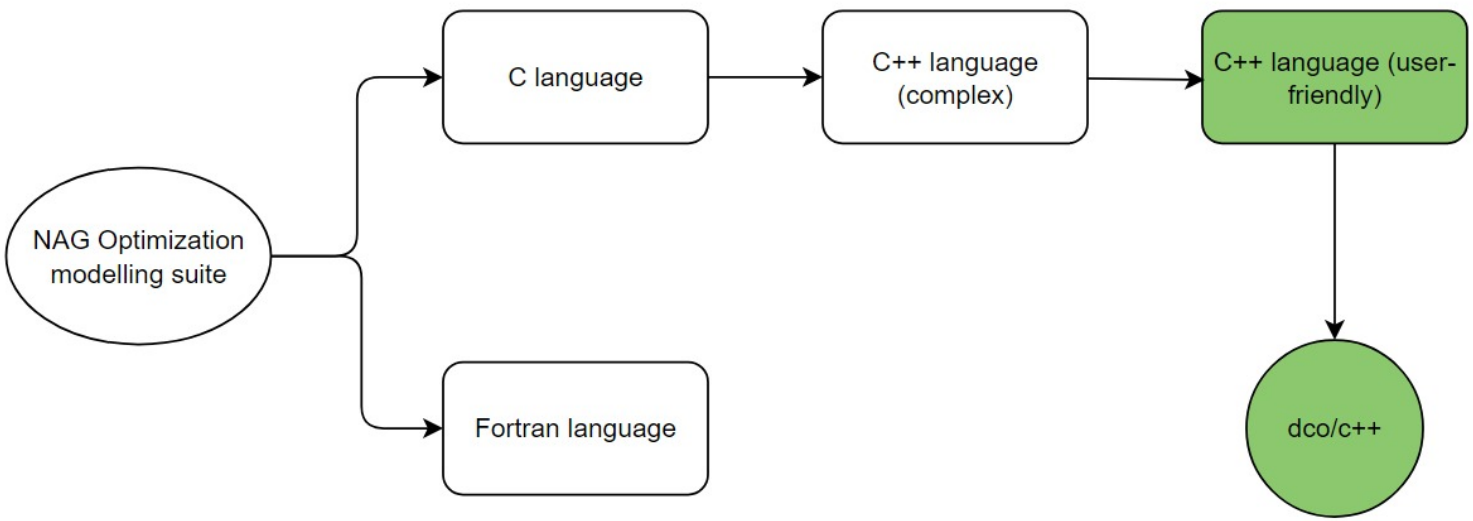
$$\begin{aligned} \underset{x \in \mathbb{R}^n}{\text{minimize}} \quad & f(x) \\ \text{subject to} \quad & l_g \leq g(x) \leq u_g, \\ & l_B \leq Bx \leq u_B, \\ & l_x \leq x \leq u_x. \end{aligned}$$

A lot more: Quadratically Constrained Quadratic Programming, Least Squares, Data Fitting...But we focus on these two for the project.

dco/c++:

- An AD software tool for computing sensitivities of C++ codes

- The NAG library (specifically the optimization modelling suite) is available in several languages, including Fortran, C, and an experimental C++ interface that is hard to use.
- A C++ layer is to be built on top of the given C++ interface, allowing the user to access the library by writing easier code. The C++ interface also utillizes dco/c++ for addition functions not existing in the given version.



Functional:

- defining optimization problems, including:
  - the ability to define and modify objective functions
- automatic computation of derivatives
- recognisation of linear contraints
  - automatic modification of problem to reflect linearity
- solve problems with either automatic solver choice or manual choice from the 3 solvers: e04mt, e04kf, e04st

Nonfunctional:

- backend: NAG Optimization Modelling Suite, C++ interface
- interface implementation in C++ using OOP
- testing with CTest and googletest
- documentation with doxygen
- automatic computation of derivatives and recognisation of linear contraints with dco/c++

How the pre-existing C++ interface works:

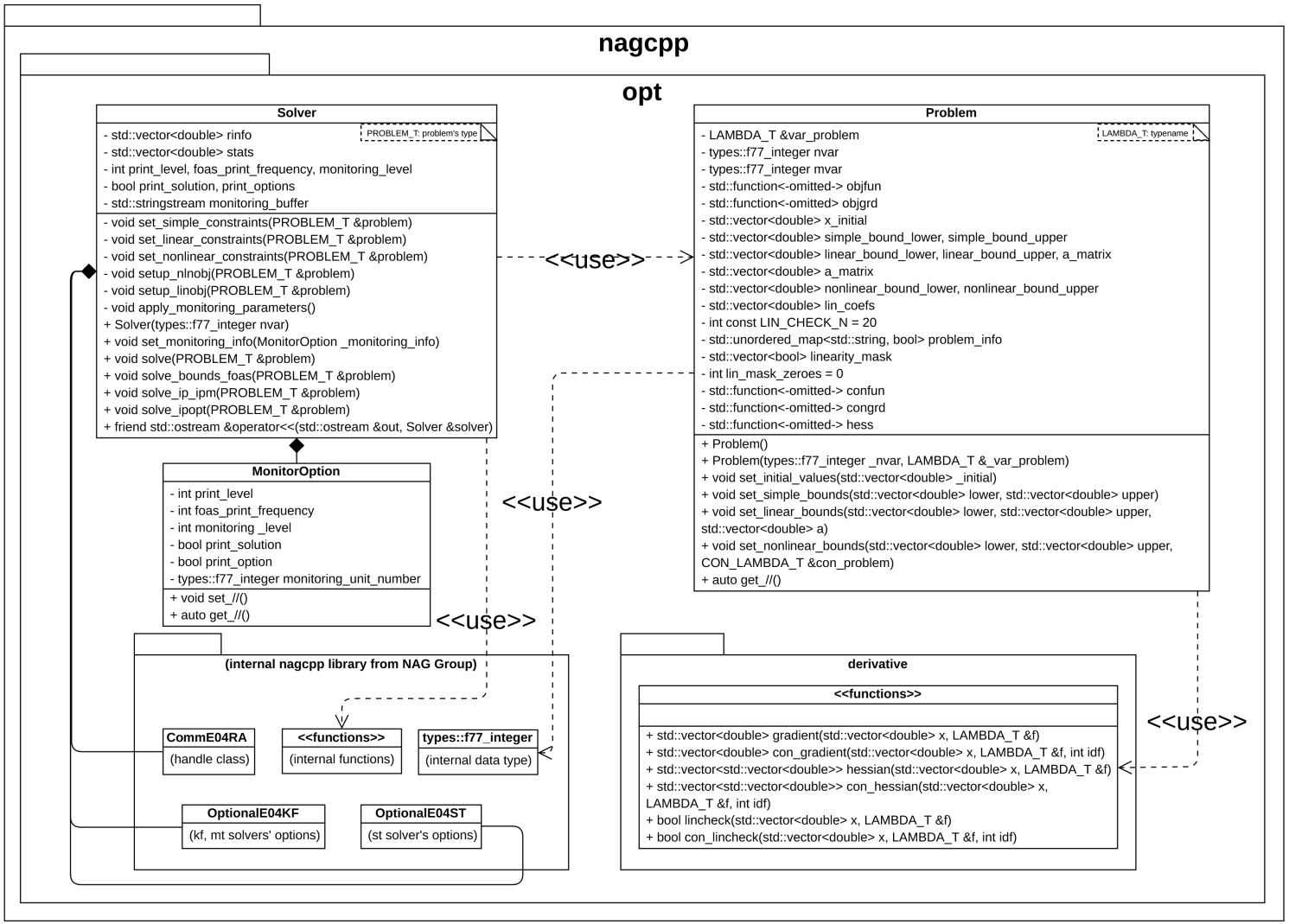
- Uses a single class called "handle", which contains everything: solver's parameters (accuracy, monitoring, etc.) and problem definition (objective function, bounds) and more.
- No classes and class method, only functions.
- If a solver requires derivative of a function, the user must write it themselves, which is prone to errors.

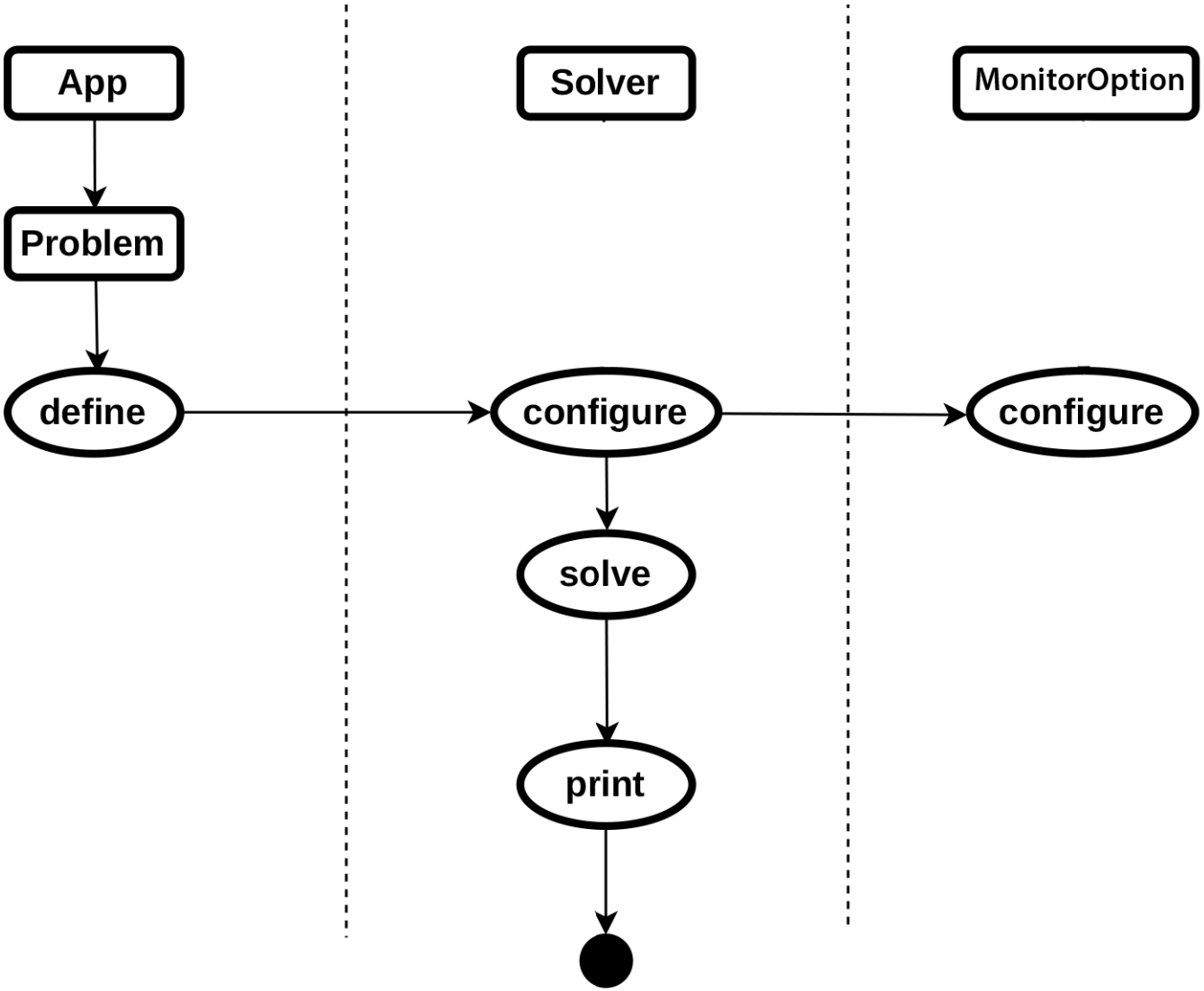
```
// create handle (e04ra)
nagcpp::opt::CommE04RA handle(nvar);

// define simple box bound (e04rh)
std::vector<double> b_lower = {-1.0, -2.0, -3.0};
std::vector<double> b_upper = {0.8, 2.0, 4.0};
nagcpp::opt::handle_set_simplebounds(handle, b_lower, b_upper);

// solver arguments
std::vector<double> rinfo;
std::vector<double> stats;
nagcpp::opt::OptionalE04KF opt;

// call solver (e04kf)
nagcpp::opt::handle_solve_bounds_foas(handle, objfun, objgrd, nullptr, x,
                                      rinfo, stats, opt);
```

```
// printing
handle.PrintLevel(1);
handle.PrintSolution("yes");
handle.PrintOptions("no");
handle.FOASPrintFrequency(5);

// monitoring
handle.MonitoringLevel(3);
handle.MonitoringFile(monitoring_unit_number);
```

Some keywords/classes from the existing C++ interface which could be used as classes in ours:

- problem
- solver
- dco/derivative
- bound
- function
  - objectiveFunction
  - constraintFunction
- linearInfo


nagcpp
opt
Solver
PROBLEM_T: problem's type
- std::vector<double> rinfo
- std::vector<double> stats
- int print_level, foas_print_frequency, monitoring_level
- bool print_solution, print_options
- std::stringstream monitoring_buffer
- void set_simple_constraints(PROBLEM_T &problem)
- void set_linear_constraints(PROBLEM_T &problem)
- void set_nonlinear_constraints(PROBLEM_T &problem)
- void setup_nlnobj(PROBLEM_T &problem)
- void setup_linobj(PROBLEM_T &problem)
- void apply_monitoring_parameters()
+ Solver(types::f77_integer nvar)
+ void set_monitoring_info(MonitorOption _monitoring_info)
+ void solve(PROBLEM_T &problem)
+ void solve_bounds_foas(PROBLEM_T &problem)
+ void solve_ip_ipm(PROBLEM_T &problem)
+ void solve_ipopt(PROBLEM_T &problem)
+ friend std::ostream &operator<<(std::ostream &out, Solver &solver)
<<use>>
Problem
LAMBDA_T: typename
- LAMBDA_T &var_problem
- types::f77_integer nvar
- types::f77_integer mvar
- std::function<-omitted-> objfun
- std::function<-omitted> objgrd
- std::vector<double> x_initial
- std::vector<double> simple_bound_lower, simple_bound_upper
- std::vector<double> linear_bound_lower, linear_bound_upper, a_matrix
- std::vector<double> a_matrix
- std::vector<double> nonlinear_bound_lower, nonlinear_bound_upper
- std::vector<double> lin_coefs
- int const LIN_CHECK_N = 20
- std::unordered_map<std::string, bool> problem_info
- std::vector<bool> linearity_mask
- int lin_mask_zeroes = 0
- std::function<-omitted-> confun
- std::function<-omitted-> congrd
- std::function<-omitted-> hess
+ Problem()
+ Problem(types::f77_integer _nvar, LAMBDA_T &_var_problem)
+ void set_initial_values(std::vector<double> _initial)
+ void set_simple_bounds(std::vector<double> lower, std::vector<double> upper)
+ void set_linear_bounds(std::vector<double> lower, std::vector<double> upper, std::vector<double> a)
+ void set_nonlinear_bounds(std::vector<double> lower, std::vector<double> upper, CON_LAMBDA_T &con_problem)
+ auto get_//()
MonitorOption
- int print_level
- int foas_print_frequency
- int monitoring _level
- bool print_solution
- bool print_option
- types::f77_integer monitoring_unit_number
+ void set_//()
+ auto get_//()
<<use>>
<<use>>
(internal nagcpp library from NAG Group)
CommE04RA
(handle class)
<<functions>>
(internal functions)
types::f77_integer
(internal data type)
OptionalE04KF
(kf, mt solvers' options)
OptionalE04ST
(st solver's options)
derivative
<<functions>>
+ std::vector<double> gradient(std::vector<double> x, LAMBDA_T &f)
+ std::vector<double> con_gradient(std::vector<double> x, LAMBDA_T &f, int idf)
+ std::vector<std::vector<double>> hessian(std::vector<double> x, LAMBDA_T &f)
+ std::vector<std::vector<double>> con_hessian(std::vector<double> x, LAMBDA_T &f, int idf)
+ bool lincheck(std::vector<double> x, LAMBDA_T &f)
+ bool con_lincheck(std::vector<double> x, LAMBDA_T &f, int idf)
<<use>>


App
Solver
MonitorOption
Problem
define
configure
configure
solve
print

- Target platform: Linux devices, specifically the Virtual Machine provided by Prof. Naumann
- Programming language: C++ 20
- Compiler: gcc
- Build system: CMake
- Runtime library: NAG Library with C++ interface, dco/c++, the standard ones (std, math.h, etc.)

Implementation

Source Code


```
.
├── include (C++ header files)
│   ├── derivative.hpp
│   ├── nag_cpp.hpp
│   ├── problem.hpp
│   └── solver.hpp
├── caseStudies (directories containing various problem cases)
├── cmake (auxiliary files required for building and testing with CMake)
│   ├── FindNAG_dco_cpp.cmake
│   └── FindNAG_Library.cmake
├── CMakeLists.txt (used for building and testing)
├── doc
│   ├── Latex_talk (slides and report)
│   ├── briefing.pdf (assignment)
│   ├── todo.txt
│   └── user guide.txt
├── examples (directories containing various example programs)
├── license
│   └── nag.key
├── README.md
├── tests (w/ google test and ctest)
│   ├── CMakeLists.txt
│   ├── test_dco
│   │   └── test_dco.cpp
│   ├── test_linearity
│   │   └── test_linearity.cpp
│   └── test_set_get
│       └── test_set_get.cpp
├── thirdParty (NAG C++ interface from NAG group)
└── └── nag_cpp
```

problem.hpp:

1. contains the objective functions, constraint functions and various kinds of bounds

```
class Problem
{
private:
    LAMBDA_T &var_problem; //!< Objective function in form of lambda function.
    types::f77_integer nvar; //!< Size of variable x.
    types::f77_integer mvar; //!< Amount of nonlinear constraints.

    std::function<void(const std::vector<double> &, double &, types::f77_integer &)> objfun; //!< Objective function in form of
std::function.
    std::function<void(const std::vector<double> &, std::vector<double> &, types::f77_integer &)> objgrd; //!< Gradient of the
objective function.

    std::vector<double> x_initial; //!< Start values of vector x.
    std::vector<double> simple_bound_lower, simple_bound_upper; //!< Borders of box constraints.
    std::vector<double> linear_bound_lower, linear_bound_upper; //!< Borders of linear constraints.
    std::vector<double> a_matrix; //!< Sparse matrix of linear constraints.
    std::vector<double> nonlinear_bound_lower, nonlinear_bound_upper; //!< Borders of nonlinear constraints.
    std::vector<double> lin_coefs; //!< Linear coefficients of objective function, if it is linear.
```

2. contains the set_//_() functions

```
//! Function for setting initial values of variable x.
/*! …
*/
void set_initial_values(std::vector<double> _initial)
{…
}

//! Function for setting box constraints.
/*! …
*/
void set_simple_bounds(std::vector<double> lower, std::vector<double> upper)
{…
}

//! Function for setting linear constraints.
/*! …
*/
void set_linear_bounds(std::vector<double> lower, std::vector<double> upper, std::vector<double> a)
{…
}

//! Function for setting nonlinear constraints.
/*! …
*/
template <typename CON_LAMBDA_T>
void set_nonlinear_bounds(std::vector<double> lower, std::vector<double> upper, CON_LAMBDA_T &con_problem)
{…
}
```

3. set_nonlinear_bounds() detects whether bounds are non-linear or not and sets them accordingly

```
/*! Check on linearity for every component of nonlinear constraints.…
for (int i = 0; i < this->mvar; ++i)
{
    bool con_is_linear = true;
    for (int num = 1; num <= LIN_CHECK_N; ++num)
    {…
    }

    if (con_is_linear)
    {
        this->linear_bound_lower.push_back(lower[i]);
        this->linear_bound_upper.push_back(upper[i]);

        std::vector<double> x(this->nvar, 1.0);
        auto con_lin_coefs = nagcpp::opt::derivative::con_gradient(x, con_wrap_problem_not_adjusted, i);
        for (int j = 0; j < this->nvar; ++j)
        {…
        }

        this->problem_info["linear_bounds_declared"] = true;
    }
    else
    {
        this->nonlinear_bound_lower.push_back(lower[i]);
        this->nonlinear_bound_upper.push_back(upper[i]);

        this->problem_info["nonlinear_bounds_declared"] = true;

        this->lin_mask_zeroes++; //!< Counting nonlinear components.
    }

    this->linearity_mask.push_back(con_is_linear); //!< Filling in linearity mask.
}
```

derivative.hpp:

- uses dco/c++
- checks linearity by examining whether the derivative is zero or not

```
template <typename LAMBDA_T>
std::vector<double> gradient(std::vector<double> x, LAMBDA_T &f)
{
    std::vector<double> grad(x.size());
    using type = dco::gt1s<double>::type;
    std::vector<type> ax(x.begin(), x.end());
    for (std::size_t i = 0; i < x.size(); ++i)
    {
        dco::derivative(ax[i]) = 1.0;
        type ay = f(ax);
        grad[i] = dco::derivative(ay);
        dco::derivative(ax[i]) = 0.0;
    }
    return grad;
}
```

monitor.hpp:

MonitorOption: a simple class used to hold monitoring options

```
class MonitorOption
{
private:
    //! Monitoring parameters
    int print_level;
    int foas_print_frequency;
    int monitoring_level;
    bool print_solution;
    bool print_options;
    nagcpp::types::f77_integer monitoring_unit_number;
```

solver.hpp:

1. contains CommE04RA handle, the internal NAG C++ interface's communicator class, as well as solver's options
2. contains set_//() wrapper functions to pass problem definition into the handle

```
class Solver
{
private:
    //! Arguments for solvers.
    CommE04RA handle; //!< Main object of problem solving. Contains all information about problem and output.
    std::vector<double> rinfo; //!< Error measures and various indicators at the end of the final iteration.
    std::vector<double> stats; //!< Solver statistics at the end of the final iteration.
    OptionalE04KF opt; //!< Optional parameter container (for solvers e04kf, e04mt).
    OptionalE04ST opt_st; //!< Optional parameter container (for solver e04st).

    //! Helper vars for monitoring.
    std::stringstream monitoring_buffer;
    nagcpp::opt::MonitorOption monitoring_info;

    //! Function for setting box constraints.
    /*! …
    */
    template <typename PROBLEM_T>
    void set_simple_constraints(PROBLEM_T &problem)
    { …
    }

    //! Function for setting linear constraints.
    /*! …
    */
    template <typename PROBLEM_T>
    void set_linear_constraints(PROBLEM_T &problem)
    { …
    }
```

3. solve() automatically chooses from one of the 3 solvers: e04kf, e04mt, and e04st which is based on the problem's type of constraints and their linearity.

```
template <typename PROBLEM_T>
void solve(PROBLEM_T &problem)
{
    apply_monitoring_parameters();

    auto problem_info = problem.get_problem_info();

    bool kf_is_applicable = true; //!< Check, if e04kf can be applied.
    bool mt_is_applicable = true; //!< Check, if e04mt can be applied.

    /*! Row of checks on applicability of different solvers,…
    if (problem_info["is_linear"])
    {…
    }
    else
    {…
    }

    if (problem_info["simple_bounds_declared"])
    {…
    }

    if (problem_info["linear_bounds_declared"])
    {…
    }

    if (problem_info["nonlinear_bounds_declared"])
    {…
    }

    std::vector<double> u; //!< Lagrange multipliers (dual variables).

    //! Calls of the respective solvers.
    if (kf_is_applicable)
    {
        handle_solve_bounds_foas(this->handle, problem.get_objfun(), problem.get_objgrd(), nullptr, problem.get_initial(), this->rinfo, this->stats, this->opt);
    }
    else if (mt_is_applicable)
    {
        handle_solve_lp_ipm(this->handle, problem.get_initial(), u, this->rinfo, this->stats, nullptr);
    }
    else
    {
        handle_solve_ipopt(this->handle, problem.get_objfun(), problem.get_objgrd(),
                           problem.get_confun(), problem.get_congrd(), problem.get_hess(),
                           nullptr, problem.get_initial(), u, this->rinfo, this->stats, this->opt_st);
    }
}
```

Documentation:

Every function and its parameters are documented thoroughly.

```
//! Function for setting linear constraints.
/*!
    \param lower lower border of a * x.
    \param upper upper border of a * x.
    \param a matrix of linear constraints.
*/
void set_linear_bounds(std::vector<double> lower, std::vector<double> upper, std::vector<double> a)
{
    this->problem_info["linear_bounds_declared"] = true;

    for (int i = 0; i < lower.size(); ++i)
    {
        this->linear_bound_lower.push_back(lower[i]);
        this->linear_bound_upper.push_back(upper[i]);
        for (int j = 0; j < this->nvar; ++j)
        {
            this->a_matrix.push_back(a[i * this->nvar + j]);
        }
    }
}
```

Example of a simple program by the user:

```
#include "nag_cpp.hpp"
using namespace std;

// Minimizes f(x) = x_1^2 + x_2^2
// with respect to x_1 in [-1, 0.8], x_2 in [-2, 2]

// Objective function
auto test_problem = [](auto const &x, auto &y)
{
        y = pow(x[0], 2) + pow(x[1], 2);
};

int main()
{
        try
        {
                // Problem parameter
                nagcpp::types::f77_integer nvar = 2;
                std::vector<double> x = {-5, 3.9};
                std::vector<double> b_lower = {-1.0, -2.0};
                std::vector<double> b_upper = {0.8, 2.0};

                // Problem definition
                nagcpp::opt::Problem problem(nvar, test_problem);
                problem.set_initial_values(x);
                problem.set_simple_bounds(b_lower, b_upper);

                // Solve
                nagcpp::opt::Solver solver(nvar);
                solver.solve(problem);

                // Monitoring
                std::cout << solver;
        }
        catch (nagcpp::error_handler::Exception &error)
        {
                std::cout << error.msg << std::endl;
        }
}
```

- Focus on clean, readable, and well documented C++ code without memory leaks or invalid memory access (tested with Valgrind).
- Modern C++ features are used, such as class template type deduction via construction from 2017.
- Mathematically less complex for the user than the existing interface – automatic derivation, no worry about sparse matrix by e04mt, etc.
- The whole point is to be user-friendly. The previous example program written with the original interface was approx. 90 lines of code, which is now 40.
- In total we wrote 1400+ lines of code, most of it in `problem.hpp`.
- Points for future developers to improve upon:
  - Instead of building off the existing C++ interface, base directly off the C interface.
  - More solvers.
  - Detection of the linearity of individual term in a constraint function. (Currently if one term is non-linear, the entire constraint function is treated as non-linear.)

Testing was done with Valgrind and GoogleTest/Ctest.

Tests of various granularities:

1. Basic set/get routines (test_set_get.cpp):

```
TEST_F(ProblemTest, SimpleBoundsEQ)
{
        auto problem = make_problem(test_problem);
        auto getSimBounds = problem.get_simple_bounds();
        EXPECT_EQ(getSimBounds.first, b_lower) << "b_lower and problem.get_simple_bounds().first are unequal";
        EXPECT_EQ(getSimBounds.second, b_upper) << "b_upper and problem.get_simple_bounds().second are unequal";
}
TEST_F(ProblemTest, NVarEQ)
{
        auto problem = make_problem(test_problem);
        ASSERT_EQ(nvar, problem.get_nvar()) << "nvar and problex.get_nvar are unequal";
}

TEST_F(ProblemTest, ObjFunEQ)
{
        auto problem = make_problem(test_problem);
        double y1, y2;
        long int _inform;
        long int &inform = _inform;
        test_problem(x, y1);
        problem.get_objfun()(x, y2, inform);
        ASSERT_EQ(y1, y2) << "test_problem and problem.get_objfun return unequal value";
}
```

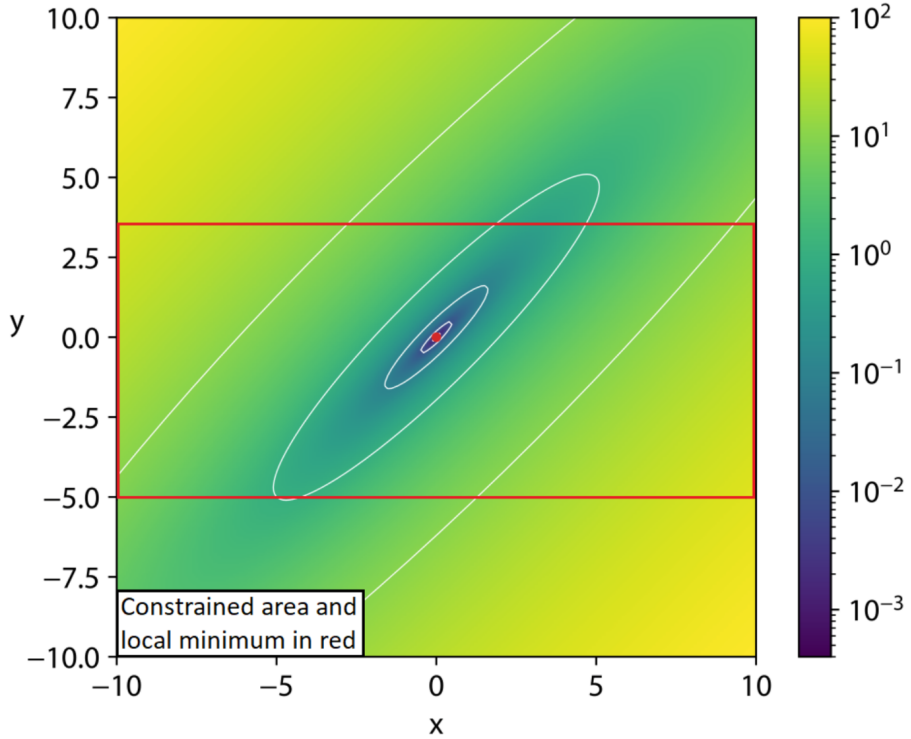
2. Derivatives computation (test_dco.cpp):

```
TEST_F(ProblemTest, ProblemObjGrdEQ)
{
        auto problem = make_problem(test_problem_1);
        std::vector<double> y1 = {-8, 14};
        std::vector<double> y2;
        long int _inform;
        long int &inform = _inform;
        problem.get_objgrd()(x, y2, inform);
        ASSERT_EQ(y1, y2) << "problem.get_objfun returns wrong values";
}

TEST_F(ProblemTest, ProblemConGrdEQ)
{
        auto problem = make_problem(test_problem_1);
        std::vector<double> y1 = {13.986254132645684, 8.6827035119133704};
        y1.resize(2);
        std::vector<double> y2;
        long int _inform;
        long int &inform = _inform;
        problem.get_congrd()(x, y2, inform);
        ASSERT_EQ(y1.size(), y2.size()) << "Vector y1 and vector y2 are of unequal length";
        for (int i = 0; i < nvar; ++i)
        {
                EXPECT_DOUBLE_EQ(y1[i], y2[i]) << "Vector y1 and vector y2 differ at index " << i << ", problem.get_confun returns wrong
values";
        }
}
```

3. More complex linearity detection logic (test_linearity.cpp):

```
TEST_F(ProblemTest, NonLinearProblemCheck)
{
        auto problem = make_problem(test_problem_1);
        auto getProbInfo = problem.get_problem_info();
        EXPECT_FALSE(getProbInfo["is_linear"]) << "Problem is linear";
}

TEST_F(ProblemTest, LinearProblemCheck)
{
        auto problem = make_problem(test_problem_2);
        auto getProbInfo = problem.get_problem_info();
        EXPECT_TRUE(getProbInfo["is_linear"]) << "Problem is not linear";
}

TEST_F(ProblemTest, LinearBoundsCheck)
{
        auto problem = make_problem(test_problem_1);
        std::vector<double> l_lower = {5.0, 1.0};
        std::vector<double> l_upper = {1e19, 1.0};
        std::vector<double> a = {2.3, 5.6, 11.1, 1.3,
                                                    1.0, 1.0, 1.0, 1.0};
        problem.set_linear_bounds(l_lower, l_upper, a);
        auto getProbInfo = problem.get_problem_info();
        EXPECT_TRUE(getProbInfo["linear_bounds_declared"]) << "No linear bounds were declared";
}

TEST_F(ProblemTest, NonLinearBoundsCheck)
{
        auto problem = make_problem(test_problem_1);
        std::vector<double> n_lower = {21.0}, n_upper = {100};
        problem.set_nonlinear_bounds(n_lower, n_upper, test_constraint_problem_2);
        auto getProbInfo = problem.get_problem_info();
        EXPECT_TRUE(getProbInfo["nonlinear_bounds_declared"]) << "No non-linear bounds were declared";
}
```

e04kf solver:



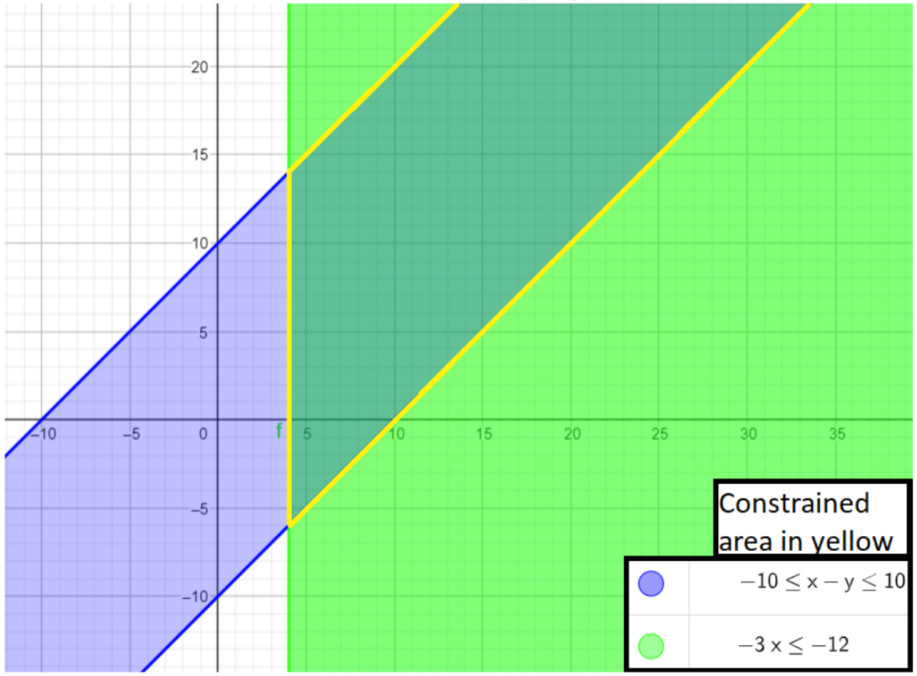
(Matyas function)[1]

$$\mathbb{R}^2 \to \mathbb{R},\ f(x) = 0.26(x^2 + y^2) - 0.48xy$$

$$l_x = \begin{pmatrix} -10 \\ -5 \end{pmatrix},\ u_x = \begin{pmatrix} 10 \\ 3.5 \end{pmatrix}$$

$$min(f(x_{min})) = 0, \quad x_{min} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

e04mt solver:



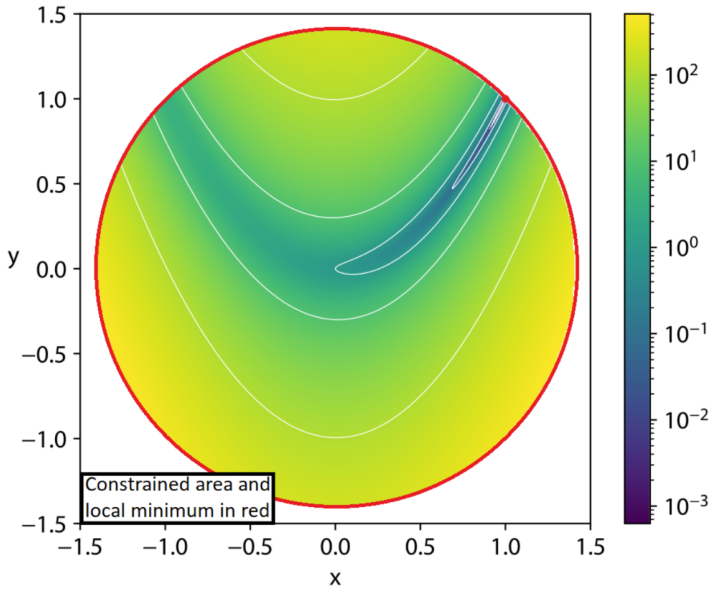
[2]

$$c = \begin{pmatrix} 1 \\ 3 \\ -2 \end{pmatrix}, l_x = \begin{pmatrix} -5 \\ -5 \\ 0 \end{pmatrix}, u_x = \begin{pmatrix} 15 \\ 2 \\ 0 \end{pmatrix}$$

$$A = \begin{pmatrix} 1 & -1 & 0 \\ -3 & 0 & 10 \end{pmatrix}, l_A = \begin{pmatrix} -10 \\ -\infty \end{pmatrix}, u_A = \begin{pmatrix} 10 \\ -12 \end{pmatrix}$$

$$min(c^T x_{min}) = -11, \quad x_{min} = \begin{pmatrix} 4 \\ -5 \\ 0 \end{pmatrix}$$

e04st solver:



(Rosenbrock on disk) [1]

$$f(x) = (1-x)^2 + 100 \cdot (y - x^2)^2$$

$$g(x) = x_1^2 + x_2^2,\ l_g = -\infty, u_g = 2$$

$$B = 0, l_B = 0,\ u_B = 0$$

$$l_x = \begin{pmatrix} -1.5 \\ -1.5 \end{pmatrix},\ u_x = \begin{pmatrix} 1.5 \\ 1.5 \end{pmatrix}$$

$$min(f(x_{min})) = 0, \quad x_{min} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

Live Software Demo

- make a build folder `mkdir build && cd build`
- run `cmake ${REPO_DIR} -DNAG_dco_cpp_DIR=${HOME}/NAG/dcl6i37ngl_v370/ -DNAG_Library_DIR=${HOME}/NAG/nll6i285bl/`
- run `make caseStudies.xxx`
- run `./caseStudies/caseStudies.xxx`
- Inspect output
- Modify in `caseStudies/xxx/*.cpp`
  - objective functions
  - constraint functions
  - bounds
  - solver printing parameters
  - automatic switch by changing `con_function` from non-linear to linear

Project Management

	Task	Members	Deadline
Phase 1	Requirement analysis	Everyone	25. 05
	Technical background (getting used to NAG and dco/c++)	Everyone	03. 07
	Slides for first presentation	Yifei	03. 07
Phase 2	Interface: write examples	Khang, Konstantin, Yifei	21. 07
	dco/c++ : example of linearity-check	Max, Ziya	28. 07
	Exam period		
	Interface: prototypes for e04kf solver	Khang, Konstantin	25. 08
	dco/c++: linearity-check program	Max, Ziya	25. 08
	Integrate dco to cpp interface	Khang, Konstantin	09. 09
	Prototypes for e04mt & e04st solvers	Khang, Konstantin	09 .09
Phase 3	Build system with CMAKE	Yifei, Khang	23. 09
	Unit test w/GoogleTest + test w/examples	Yifei, Khang, Max	07. 10
Phase 4	Doxygen & user/developer documentation	Khang, Konstantin	07. 10
	Slides for final presentation	Yifei, Khang	13. 10
Phase 5	Final Report	Everyone	Mid/End Nov.

- Group meetings every week to discuss work.
- Group meetings with supervisor every 2 week to have them check on progress.
- Tasks were always assigned to pair, if one person is unavailable there is still another, which proved much of use.

Summary and Conclusion

- We successfully developed a new object oriented C++ UI for NAG optimization modelling suite (NOMS) using the NAG library and dco/c++ for Automatic Diffrentiation that allows users to define and solve various optimization problems in a uniform manner.
- Our C++ interface is able to separate the definition of the optimization problem and the call to the solver so that it is possible to set up a problem in the same way for different solvers.
- We found at least three examples for each solver which are used in testing for the interface.
- We worked as a group, kept our communication both internal and with our supervisor Johannes, managed the time and overcame exceptional situations such as sickness, which was precious experience in software development for all of us.

Reference

1. image source:
   https://en.wikipedia.org/wiki/Test_functions_for_optimization
2. image made with desmos.com
3. NAG-Library manual:
   https://www.nag.com/numeric/nl/nagdoc_latest/nlhtml/
   -frontmatter/manconts.html

Thank you for listening!