Introduction to CENSO

A tutorial for downloading, installing and using CENSO, with example problems.

June 3, 2015

Contents

1 Introduction

1.1 Intent and reader prerequisites

This manual intends to give the reader a quick introduction to the CENSO framework, so (s)he will be able set up and solve optimization problems. It will focus on the practical aspects rather than the theory behind the concepts used in the solver algorithm. Therefore, the reader should at least be acquainted with the following topics within the field of optimization:

- Mathematical formulation of mixed-integer nonlinear programming (MINLP) optimization problems.
- Convexity, and the importance of convexity within optimization. A good introduction to this topic can be found in chapters 2-4 of Boyd and Vandenberghe's *Convex Optimization* [2].
- Interior-point algorithms. Ipopt is an interior-point algorithm, and it is the default solver for local optimization problems in CENSO. Good introductions on interior-point algorithms are found in chapter 11 of *Convex Optimization* [2] and chapters 14 and 19 of Nocedal and Wright's *Numerical Optimization* [5].
- Branch-and-Bound algorithms. An introduction can be found in e.g. [3].

TODO: Describe CENSO!!!!

1.2 Types of problems solved

CENSO is a framework for solving general mixed-integer nonlinear programs (MINLPs) on the form

$$\begin{aligned} \text{minimize} \quad & f(x) = c^\top x, && (1a)\\ \text{subject to} \quad & c_i^L \leq c_i(x) \leq c_i^U, \quad i = \{1, \dots, m\} && (1b)\\ & x^L \leq x \leq x^U, && (1c) \end{aligned}$$

where $x = \begin{bmatrix} x_\mathrm{c}^\top & x_\mathrm{i}^\top \end{bmatrix}^\top$. $x_\mathrm{c} \in \mathbb{R}^{n_\mathrm{c}}$ is a vector of continuous optimization variables and $x_\mathrm{i} \in \mathbb{Z}^{n_\mathrm{i}}$ is a vector of integer optimization variables. The total number of optimization variables is $n = n_\mathrm{c} + n_\mathrm{i}$. The presence of integer variables and nonlinear constraints makes this problem a MINLP.

- Equation (1a) shows the objective function $f : \mathbb{R}^{n_\mathrm{c}} \times \mathbb{Z}^{n_\mathrm{i}} \mapsto \mathbb{R}$, which is assumed to be linear in the optimization variables. This does not lead to any loss of generality, since any optimization problem can be written this way by converting it to its epigraph form (see section 1.4.1). $c \in \mathbb{R}^n$ is a vector of constants.

- Equation (1b) shows the constraints of the MINLP. Each $c_i : \mathbb{R}^{n_\mathrm{c}} \times \mathbb{Z}^{n_\mathrm{i}} \mapsto \mathbb{R}$ is a constraint function, $c_i^L \in (\mathbb{R} \cup \{-\infty\})$ is the lower bound for the constraint function and $c_i^U \in (\mathbb{R} \cup \{\infty\})$ is the upper bound for the constraint function. An equality constraint is specified by setting $c_i^L = c_i^U = c_i(x)$. The number of constraints is m. No particular assumptions are made about the constraint functions; they can be linear or nonlinear, convex or nonconvex. However, to ensure predictable solver behaviour, they should be twice continuously differentiable.

- Equation (1c) shows the domain bounds of each variable. $x^L \in (\mathbb{R} \cup \{-\infty\})^n$ is a vector of lower bounds and $x^U \in (\mathbb{R} \cup \{\infty\})^n$ is a vector of upper bounds. Even though these constraints could be included in (1b), they are stated explicitly here for the sake of simplicity.

1.3 Supported constraint classes

Even though any constraint can be implemented in CENSO, a few common classes of constraints have been added to the framework for simple implementation in the MINLP. The variables used in these constraints can be any of the optimization variables, continuous and/or integer. Some of the constraints specify relationships between a subset of the optimization variables. In these cases we will use the notation $\tilde{x} \in \mathbb{R}^q \times \mathbb{Z}^r, q \leq n_\mathrm{c}, r \leq n_\mathrm{i}$ for a subset of the optimization variables. The constraint classes which are currently supported are:

- Linear constraints on the form

$$\tilde{c}^L \leq A\tilde{x} \leq \tilde{c}^U, \tag{2}$$

  Recall that a linear equality constraint on the form $A\tilde{x} = b$ can be specified by letting $\tilde{c}^L = \tilde{c}^U = b$. Here, $\tilde{c}^L \in (\mathbb{R} \cup \{-\infty\})^{q+r}$ and $\tilde{c}^U \in (\mathbb{R} \cup \{\infty\})^{q+r}$ are vectors of upper and lower bounds corresponding to the subset of the optimization variables.

- Quadratic constraints on the form

$$c_i^L \leq \tilde{x}^\top P \tilde{x} + q^\top \tilde{x} + r \leq c_i^U, \tag{3}$$

  where P is a $(q+r) \times (q+r)$ matrix (not necessarily symmetric), and $q \in \mathbb{R}^{q+r}$ and $r \in \mathbb{R}$ are constant vectors.

- One-dimensional quadratic constraints on the form

$$c_i^L \leq a x_0^2 + b x_0 + c - x_1 \leq c_i^U, \tag{4}$$

  where a, b and c are constants, and x_0 and x_1 are any two optimization variables.

- Bilinear constraints on the form

$$c_i^L \leq a x_0 x_1 - x_2 \leq c_i^U, \tag{5}$$

  where a is a constant, and x_0, x_1 and x_2 are any three optimization variables.

- Exponential constraints on the form

$$c_i^L \leq a e^{b x_0} - x_1 \leq c_i^U, \tag{6}$$

  where a and b are constants, and x_0 and x_1 are any two optimization variables.

- Polynomial constraints on the form

$$c_i^L \leq p(\tilde{x}) \leq c_i^U. \tag{7}$$

  The polynomial $p(\tilde{x})$ consists of s monomials, and is specified by a vector $c \in \mathbb{R}^s$ and an $s \times (q+r)$ matrix $E = \{e_{i,j}\}$:

$$\begin{array}{rcccccccc} p(\tilde{x}) & = & c_0 & \cdot & \tilde{x}_0^{e_{0,0}} & \cdot & \tilde{x}_1^{e_{0,1}} & \cdot \ \ldots \ \cdot & \tilde{x}_{q+r-1}^{e_{0,q+r-1}} \\ & + & c_1 & \cdot & \tilde{x}_0^{e_{1,0}} & \cdot & \tilde{x}_1^{e_{1,1}} & \cdot \ \ldots \ \cdot & \tilde{x}_{q+r-1}^{e_{1,q+r-1}} \\ & + & \ldots & & & & & & \\ & + & c_{s-1} & \cdot & \tilde{x}_0^{e_{s-1,0}} & \cdot & \tilde{x}_1^{e_{s-1,1}} & \cdot \ \ldots \ \cdot & \tilde{x}_{q+r-1}^{e_{s-1,q+r-1}}. \end{array} \tag{8}$$

  To clarify with an example, the polynomial defined by

$$\tilde{x} = \begin{bmatrix} x_1 \\ x_3 \\ x_8 \end{bmatrix}, \quad c = \begin{bmatrix} 5 \\ 3 \\ 10 \\ 1 \end{bmatrix}, \quad E = \begin{bmatrix} 4 & 2 & 0 \\ 1 & 0 & 8 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix},$$

  with $s = 4$, would look like

$$p(\tilde{x}) = 5x_1^4 x_3^2 + 3x_1 x_8^8 + 10x_8 + x_1.$$

- Sine function constraints on the form

$$c_i^L \leq a \sin(b x_0) - x_1 \leq c_i^U, \tag{9}$$

  where a and b are constants, and x_0 and x_1 are any two optimization variables.

Constraints which do not fit into any of the classes above, can be implemented in one of two ways:

- Creating a new constraint class which matches the constraint exactly.
- Sampling the constraint in a grid and create a B-spline approximation of the constraint (see section 1.4.3 for details).

1.4 Mathematical background

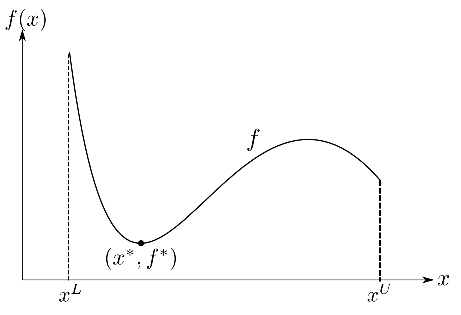
This section discusses a few of the mathematical concepts needed to better understand how CENSO works. The *epigraph form* is a transformation of an optimization problem to a form with a linear objective, which we need in CENSO. *Convexity* and *convex relaxations* are important topics in optimization theory in general, and in Branch-and-Bound algorithms such as CENSO, good convex relaxations are essential for a global solution to be found. The *B-spline* is a powerful approximation and interpolation technique that can be used in CENSO to approximate nonlinear functions with piecewise polynomials. Also, we can easily find good convex relaxations of B-splines, which is a very useful property. We also introduce global optimization and the Branch-and-Bound algorithm.

1.4.1 The epigraph form

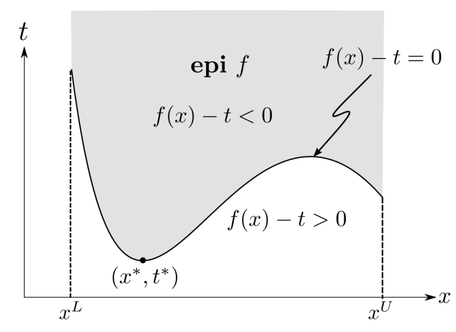
Before implementing the optimization problem in CENSO, we state the problem in its *epigraph form.* This is done by taking the original minimization problem (1), introducing an additional slack variable t, and minimizing t subject to an additional constraint which says that t should be greater than or equal to the original objective $f(x)$:

$$\begin{aligned} \text{minimize} \quad & t, && \text{(10a)} \\ \text{subject to} \quad & f(x) - t \leq 0, && \text{(10b)} \\ & c_i^L \leq c_i(x) \leq c_i^U, \quad i = \{1, \dots, m\} && \text{(10c)} \\ & x^L \leq x \leq x^U, && \text{(10d)} \end{aligned}$$

That is, we want to find the smallest t that lies on or above the graph space of $f(x)$. This is illustrated in Figure 1. We now have a new objective function which is linear (and convex). This conversion is always possible, meaning we can assume that we only have to deal with nonconvexity and nonlinearity in the constraints. From Figure 1 we see that (x^*, t^*) is optimal for (10) if and only if x^* is optimal for (1) and $t = f(x)$.



(a) Standard form



(b) Epigraph form. Note that (10b) constrains feasible points (x, t) to lie on or above the graph space of $f(x)$, which we denote **epi** f.

Figure 1: Graphical representation of a standard form optimization problem and an epigraph form optimization problem.

1.4.2 Convexity and convex relaxations

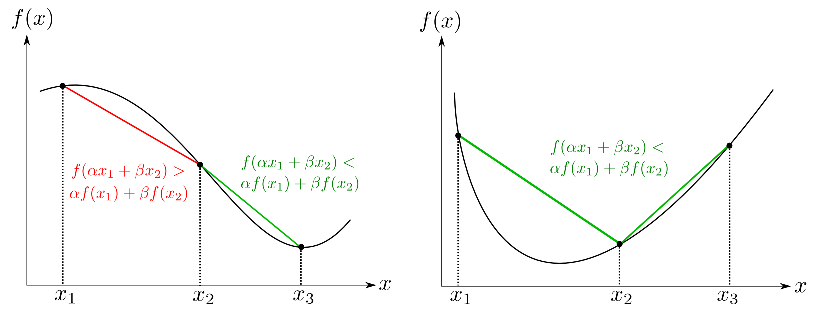
Convex optimization problems. It is well known that a solution to an optimization problem obtained by a local solver such as Ipopt is not necessarily the global solution. However, if the optimization problem is convex, we can guarantee that the solution is the global solution. For an optimization problem to be convex, the following conditions must be satisfied:

- The objective function $f(x)$ must be convex. This is always the case in (10), since the objective function (10a) is linear.
- The inequality constraints must be convex.
- The equality constraint must be linear.

Convex functions. A function $f : \mathbb{R}^n \mapsto \mathbb{R}$ is *convex* when the following is satisfied:

$$f(\alpha x_1 + \beta x_2) \leq \alpha f(x_1) + \beta f(x_2), \tag{11}$$

for all $x_1, x_2 \in \mathbb{R}^n$ and all $\alpha, \beta \in \mathbb{R}$ with $\alpha + \beta = 1,\ \alpha \geq 0,\ \beta \geq 0$. This means that a line joining two points x_1 and x_2 on f will lie on or above f for all choices of x_1 and x_2.



(a) A non-convex function. The line joining $(x_1, f(x_1))$ and $(x_2, f(x_2))$ lies below the graph of f, violating the convexity condition (11). This is indicated in red. The green line joining $(x_2, f(x_2))$ and $(x_3, f(x_3))$ lies above f, meaning f is convex in this interval.

(b) A convex function. No matter where we choose to place x_1, x_2 and x_3, the lines joining the points lie above f.

Figure 2: Convex and non-convex functions.

Linear functions are convex, because all points on a line joining two points on the function would coincide with the function itself. Functions with positive curvature in the entire function domain (positive definite Hessian) are convex.

Convex sets.

Convex hulls.

Convex relaxations.

1.4.3 The B-spline

1.4.4 Global optimization and the Branch-and-Bound algorithm

1.5 Availability

CENSO is not publically available as of today.

1.6 Prerequisites

CENSO relies on a few open-source libraries to function. The most important of these are listed below.

1.6.1 Ipopt

Ipopt is the default solver used in CENSO for solving optimization problems locally. Ipopt uses an interior point algorithm to find local solutions of (1). For more information on Ipopt and installation instructions, the reader is referred to the Ipopt home page. Details about the algorithm itself can be found in [6].

1.6.2 Eigen

Eigen is a C++ template library for linear algebra. It includes functionality for Matlab-like matrix and vector operations, solvers and other related algorithms such as matrix decompositions. For more information on Eigen and installation instructions, the reader is referred to the Eigen web site [4].

2 Installation

2.1 Getting system packages

2.2 Getting the code

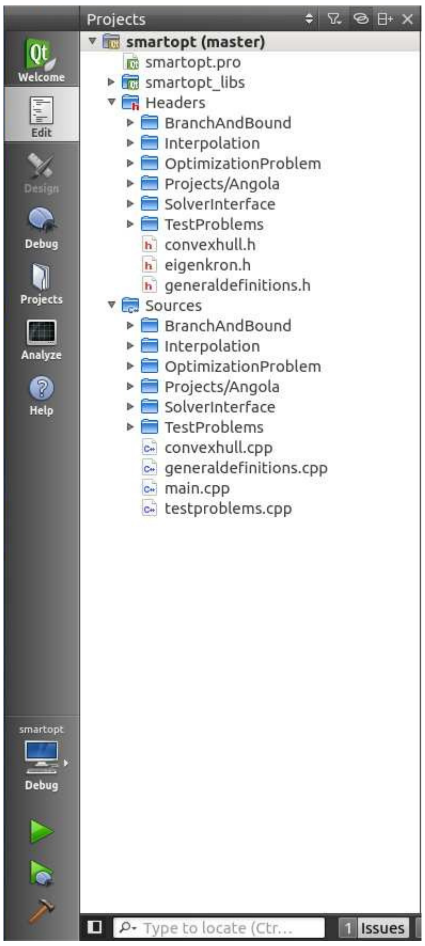
2.3 External code

2.4 Compiling

3 Using CENSO through C++/Qt

CENSO is written in C++ 11, using the Qt IDE (Qt Creator), which is a powerful and widely used IDE. Although the Qt framework offers a vast suite of functions, these are not used in CENSO for the sake of easy portability.

3.1 Code structure



```
smartopt (master)
    smartopt.pro
  ▸ smartopt_libs
  ▾ Headers
    ▸ BranchAndBound
    ▸ Interpolation
    ▸ OptimizationProblem
    ▸ Projects/Angola
    ▸ SolverInterface
    ▸ TestProblems
      convexhull.h
      eigenkron.h
      generaldefinitions.h
  ▾ Sources
    ▸ BranchAndBound
    ▸ Interpolation
    ▸ OptimizationProblem
    ▸ Projects/Angola
    ▸ SolverInterface
    ▸ TestProblems
      convexhull.cpp
      generaldefinitions.cpp
      main.cpp
      testproblems.cpp
```

Figure 3: Folder structure in the CENSO code, as shown in Qt Creator:
BranchAndBound: Contains the code and classes required for the Branch-and-Bound algorithm.
Interpolation: Contains interpolation functionality. This includes the `InterpolationTable` class, the B-spline functionality, and linear interpolation functionality.
OptimizationProblem: Contains the classes required to formulate the optimization problem, that is, the classes which describe constraints and objective functions.
SolverInterface: Contains the code which translates the optimization problem desribed by the constraint and objective function classes into the format required by the different solvers used (Ipopt, Bonmin, Nomad).
TestProblems: Contains a framework for testing and some testing problems (including the examples in chapter 6).

In Qt Creator, the .cpp and header files are automatically placed in separate folders project tree view, even though they are stored in the same actual folder. When working on a file in the editor, pressing the F4 key conveniently switches between the .cpp and header file.

In addition to the folders described above, some additional files deserve some explanation:

- `censo.pro` is the CENSO project file, and the project is opened in Qt Creator by opening this file. What it actually contains is the information required by qmake to build the application.
- The folder `censo_libs` contains the `censo_libs.pri` file which contains library include paths to the various libraries used by CENSO. It also specifies some qmake configuration and compile flags.
- `generaldefinitions.h` and `generaldefinitions.cpp` contain some general functionality like typedefs, enums, printing functions and transformations, which is reused throughout the code.
- The `main.cpp` file contains the `main()` function.

4 Formulation of an optimization problem

This section will introduce the C++ classes of the optimization framework. An optimization problem is made up of one objective object, one constraint object, three double vectors (variable lower bound, upper bound and solver starting point). Branch-and-Bound solvers will in addition to these need two int vectors, one for variable types (continuous, binary or integer) and one for the indices of variables that can be divided in the branching procedure.

4.1 Type definitions

The framework makes use of type definitions for the most commonly used data structures and types.

```
// Eigen vectors
typedef Eigen::VectorXd DenseVector;
typedef Eigen::SparseVector<double> SparseVector;

// Eigen matrices
typedef Eigen::MatrixXd DenseMatrix;
typedef Eigen::SparseMatrix<double> SparseMatrix; // declares a
    column-major sparse matrix type of double
```

4.2 Smart pointers

Smart pointers are used for all classes defined in the framework. The smart pointer implementation used is the stl `shared_pointer` via typedefs that append *Ptr* to the class name, i.e.

```
typedef std::shared_ptr< ClassName > ClassNamePtr;.
```

4.3 Variable bounds

Variable bounds are represented as stl vectors of type double. A constant `INF` is defined to represent positive infinity and is used in the case on an unbounded variable.

As an example take a problem with the three variables x_0, x_1, x_2 and the variable bounds

$$\begin{aligned} 0 &\leq x_0 \leq 1 \\ 0 &\leq x_1 \leq 3 \\ -\infty &\leq x_2 \leq \infty \end{aligned}$$

The declaration of these would be

```
std::vector< double > lb {0, 0, -INF};
std::vector< double > ub {1, 2,  INF};
```

4.4 Variable types

Variable types are represented as stl vectors of type `int`. The types are declared as an `enum`. The available types are `BINARY`, `INTEGER` and `CONTINUOUS`. The code snippet below shows how to create a vector of variable types.

```
std::vector< int > variable_types {BINARY, INTEGER, CONTINUOUS
    };
```

4.5 Starting point

Solvers require a starting point for the optimization. This is provided as a double vector.

```
std::vector< double > z0 {0, 0, 0};.
```

4.6 Branching variables

Branch and bound solvers must be supplied with a list of indices that indicate which variables that should be used in the branching procedure. This is to avoid unnecessary branching on variables that do not take part in nonconvex terms. The indexes are given as an `int` vector. As an example; to allow branching on x_0, x_1 and x_5 the code would be

```
std::vector< int > branching_variables {0, 1, 5};.
```

4.7 The objective function

The objective is given by the `Objective` class. `Objective` is abstract and defines the interface that solvers can use. The most important functions of the interface are explained below.

```
void eval(DenseVector& x,DenseVector& y)
```

evaluates the objective at `x` and stores the value in `y`.

```
void evalGradient(DenseVector& x,DenseVector& dx)
```

evaluates the objective gradient at `x` and stores the value in `dx`.

```
void evalHessian(DenseVector& x,DenseVector& ddx)
```

evaluates the objective Hessian at `x` and stores the value in `ddx`. The Hessian is stored in the format indicated by

```
void structureHessian(std::vector< int >& iRow, std::vector<
    int >& jCol)
```

where `iRow` and `jCol` indicates the positions in the Hessian matrix.

```
void augmentDomain(int dim)
```

will increase the dimension of the domain without altering the objective function itself (this is only to allow solver to introduce additional variables for instance when creating convex relaxations).

As an example; to create the linear objective function $f(x) = c^T x$ with $c^T = [0, 0, 1]$ we must first create the vector c^T. It is represented by a Eigen matrix object $v = c^T$. We can create this by writing

```
int numVars = 3;
DenseMatrix v (1,numVars);
v << 0, 0, 1;
```

An `Objective` pointer is then made by writing

```
ObjectivePtr obj (new ObjectiveLinear(v));
```

Currently only linear objectives are avaiable. The global branch and bound solver assumes a convex objective.

4.8 The constraints

Constraints are defined using the `Constraint` class. Most constraint class implementations represents a type of constraint function, i.e. linear equations, polynomials or B-splines. The exception being the composite constraint. The composite constraint holds a collection of other constraint objects and represents them as if they were one unified object.

The most important functions in the constraint interface are described below.

```
void eval(DenseVector& x, DenseVector& y)
```

evaluates the constraint at `x` and stores the value in `y`.

```
void evalJacobian(DenseVector& x, DenseVector& dx)
```

evaluates the constraint Jacobian at `x` and stores the value in `dx` using the structure given by

```
void structureJacobian(std::vector< int >& iRow, std::vector<
    int >& jCol).
```

```
void evalHessian(DenseVector& x, DenseVector& ddx)
```

evaluates the constraint Hessian at `x` and stores the value in `ddx` using the structure given by

```
void structureHessian(std::vector< int >& eqnr, std::vector<
    int >& iRow, std::vector< int >& jCol)
```

```
void setDomainBounds(std::vector< double > lb, std::vector<
    double > ub)
```

chages the domain bounds. The new bounds must be a subset of the current constraint domain.

The constraint composite is a special constraint object that contains a list of other constraint objects. It is based on the composite pattern. The composite has a add function that accepts a constraint object along with a vector of indexes that indicates which variables that are related to the constraint.

the following code snippet illustrates how to make a cubic B-spline constraint from a data table.

```
InterpolationTable data = ...
int splineDegree = 3;
int equality = true;
ConstraintPtr cbspline(new ConstraintBspline(data, 3, equality)
    );
```

If we wish to use more than one constraint they must be collected in a composite object. The composite is created by first passing the number of variables along with the variable bounds. The constraint objects are then added in turn.

```
int numvars = ...;
ConstraintCompositePtr constraints(new ConstraintComposite(
    numvars, lb, ub));

//add the bspline constraint using variables x1, x3 and x5
std::vector< int > idx{1, 3, 5};
constraints->add(cbspline, idx);

//add more constraints...
std::vector< int > idx2{i1, i2, ..., in};
ConstraintPrt c2 = new Constraint....
constraints->add(c2, idx2);
```

4.9 The Solver

The solvers are called by creating an `Optimizer` object, passing the objective, constraint, bounds, starting point and, if required, the variable types and branching variables to the `Optimizer` constructor. The problem is then solved by calling `optimize()` and the solution can be extracted using get functions. Available solvers are Ipopt, Bonmin and a homemade global branch and bound solver.

The optimization problem is solved by calling `optimize()`. The return value from optimize indicates whether the solver was successful or not. Possible return values are 1 (success) and 0 (unsuccessful).

```
//Local solution using Ipopt (ignoring integer restrictions)
OptimizerIpopt ip (objective, constraints, z0);
int returnStatus = ip.optimize();

//Local, integer feasible solution using Bonmin
OptimizerBonmin ip (objective, constraints,
    z0, variable_types, branching_variables);
int returnStatus = ip.optimize();

//Global solution using branch and bound
BranchAndBound bnb(objective, constraints,
    z0, variable_types, branching_variables);
int returnStatus = bnb.optimize();
```

5 Solvers

5.1 Ipopt

Ipopt solves convex NLPs and LPs. Refer to the Ipopt documentation.

5.2 CENSO

CENSO solves convex MINLPs and non-convex MINLPs with convex relaxations available.

5.2.1 Basic algorithm parameters

This section describes the parameters/settings of the CENSO Branch-and-Bound algorithm. The corresponding variables that must be adjusted for the various parameters are found in table 1. These variables are all members in the `BranchAndBound` class. If adjustments are to be made, they must be made in the instance of `BranchAndBound` used for optimization.

- **Convergence limit ε.** The value of ε is the termination criteria for the Branch-and-Bound algorithm. When the optimality gap (the difference between the upper and lower bounds) becomes less than or equal to ε, the Branch-and-Bound algorithm terminates.

- **Maximum number of iterations.** This parameter sets the maximum number of iterations before the algorithm terminates (regardless of whether the optimal point has been found or not).

- **Maximum number of infeasible iterations.** This parameter sets the maximum number of infeasible iterations before the algorithm terminates. A high number of infeasible iterations can indicate that the algorithm has ventured outside the domain in which the constraints are well defined, and is struggling to find feasible solutions.

- **Maximum node tree depth.** This parameter sets the maximum depth in the Branch-and-Bound node tree. It the node tree gets very deep, this can indicate that the convex relaxations do not get tight, resulting in indefinite branching.

Parameter	**Data type**	**Variable name**
Convergence limit ε	`double`	`epsilon`
Max. # iterations	`int`	`maxIterations`
Max. # infeasible iterations	`int`	`maxInfeasibleIterations`
Max. node tree depth	`int`	`maxDepth`

Table 1: Algorithm parameters and corresponding variable names

5.2.2 Branch-and-Bound specific parameters

This section describes a few more specialized parameters which can be used to adjust the behaviour of the Branch-and-Bound algorithm. A summary of these parameters and their corresponding variables is found in table 7. These variables are all members in the `BranchAndBound` class. If adjustments are to be made, they must be made in the instance of `BranchAndBound` used for optimization.

- **Node selection strategy.** This parameter decides which node in the node tree is selected as the next node for processing. The available options are shown in table 2.

Variable name: `nodeSelectionStrategy`	
`BEST_FIRST`	Selects the node with the best (greatest) lower bound.
`DEPTH_FIRST`	Selects the nodes based on a standard depth-first search (DFS).
`BREADTH_FIRST`	Selects the nodes based on a standard breadth-first search (BFS).
`DEPTH_FIRST_BREADTH_AFTER`	Selects nodes based on depth-first search until a feasible solution is found, then switches to breadth-first search.
`BREADTH_FIRST_DEPTH_AFTER`	Selects nodes based on breadth-first search until a feasible solution is found, then switches to depth-first search.
`RANDOM_NODE`	Selects a random node from the tree.

Table 2: Node selection strategy options

- **Node processing policy.** This parameter decides the policy for node processing. The available options are shown in table 3.

Variable name: `nodeProcessingPolicy`	
`EAGER`	When a node is selected, its child nodes are processed - nodes are selected after they have been processed. The benefit of this approach is that it causes the algorithm to select better nodes for processing, but it may process more nodes than necessary.
`LAZY`	When a node is selected, the node itself is processed - nodes are selected before they have been processed. Selecting nodes before they are processed may cause the algorithm to select poor nodes for processing, but it does not process more nodes than necessary.

Table 3: Node selection strategy options

- **Branching rules for integer variables.** This parameter decides the rules for branching on integer varianles. The available options are shown in table 4.

Variable name: `branchingRuleInteger`	
`MOST_FRACTIONAL`	Branches on the most fractional integer variable, that is, the variable furthest from an integer value, that is, the variable with fractional part closest to 0.5.
`PSEUDOCOST`	This rule keeps track of how successful the algorithm has been in branching on each variable which has been branched on so far, and selects the variable with the best record in increasing the lower bound. *This strategy has not yet been implemented.*
`STRONG_BRANCHING`	Strong branching tests which of the candidates will give the best improvement before branching. *This strategy has not yet been implemented.*

Table 4: Options for branching rules for integer variables

- **Branching rules for continuous variables.** This parameter decides the rules for branching on continuous varianles. The available options are shown in table 5.

Variable name: `branchingRuleContinuous`	
`MOST_PROMISING`	This rule keeps track of how successful the algorithm has been in branching on each variable which has been branched on so far, and selects the variable with the best record in increasing the lower bound.
`LONGEST_INTERVAL`	This rule selects the variable with the longest bound interval, that is, the difference between the upper and lower bound for the variable.
`RANDOM_VARIABLE`	This rule selects a random variable for branching.

Table 5: Options for branching rules for continuous variables

- **Branching priority.** This parameter decides how the different variable types are prioritized for branching. The available options are shown in table 6.

Variable name: branchingRuleContinuous	
BINARY_INTEGER_CONTINUOUS	Starts with branching on all binary variables. When finished with the binary variables, the algorithm branches on the integer variables. Finally, continuous variables are branched on.
RANDOM_MIX	Selects a random mix of binary, integer and continuous variables for branching.

Table 6: Options for branching priority

A summary of the parameters described above is found in the table below:

Parameter	**Variable name**	**Possible values**
Node sel. strategy	nodeSelectionStrategy	BEST_FIRST DEPTH_FIRST BREADTH_FIRST DEPTH_FIRST_BREADTH_AFTER BREADTH_FIRST_DEPTH_AFTER RANDON_NODE
Node proc. policy	nodeProcessingPolicy	EAGER LAZY
Branching rule for int. vars.	branchingRuleInteger	MOST_FRACTIONAL PSEUDOCOST STRONG_BRANCHING
Branching rule for cont. vars.	branchingRuleContinuous	MOST_PROMISING LONGEST_INTERVAL RANDOM_VARIABLE
Branching priorities	branchingPriority	BINARY_INTEGER_CONTINUOUS RANDOM_MIX

Table 7: Summary of Branch-and-Bound specific parameters, their corresponding variable names and possible values.

5.3 Bonmin

Bonmin solves convex MINLPs. Refer to the Bonmin documentation.

6 Example problems

6.1 Testing framework and the `TestProblem` class

The `TestProblem` class is a framework for creating test problems. The code that implements the test problem itself is in the `solveProblem()` function of each class. The test problems described in the following section are summarized in table 8. Each of these classes inherit from `TestProblem`.

Section	**Problem**	**Class name**
6.2.1	Farming LP case	`FarmingLP`
6.2.2	Farming QP case	`FarmingQP`
6.2.3	Farming QP case with integer variables	`FarmingInteger`
6.3.1	Network maximum flow case	`MaximumFlow`
6.3.2	Network maximum flow with routing	`FlowWithRouting`
6.4.3	Six-Hump Camelback function: Local optimization with the `ConstraintPolynomial` constraint class	`SixHumpCamelPolynomial`
6.4.4	Six-Hump Camelback function: Local optimization with a custom constraint class	`SixHumpCamelCustom`
6.4.5	Six-Hump Camelback function: Global optimization using an alpha-Branch-and-Bound approach with custom constraint and relaxed constraint classes	`SixHumpCamelABnB`
6.4.6	Six-Hump Camelback function: Global optimization using B-Spline approximation	`SixHumpCamelBSpline`

Table 8: Test problems and their class names

6.2 Farming problem

In this section we will look at a simple optimization problem taken from the lecture notes of the NTNU course TTK4135 Optimization and Control. We will look at three cases:

1. Linear programming case
2. Quadratic programming case
3. Introduction of integer variables

6.2.1 Linear programming case

We will start with a simple LP case. A farmer wants to grow apples and bananas. He has a field of size 100000 m^2. Growing 1 tonne of apples requires an area of 4000 m^2 and 60 kg of fertilizer. Growing 1 tonne of bananas requires an area of 3000 m^2 and 80 kg of fertilizer. The profit for apples is 7000 per tonne (including fertilizer cost), and the profit for bananas is 6000 per tonne (including fertilizer cost). The farmer can legally use up to 2000 kg of fertilizer. He wants to maximize his profits.

Optimization variables. We will introduce two optimization variables. x_0 is the number of tonnes of apples grown, and x_1 is the number of tonnes of bananas grown. We define a vector of optimization variables $x = [x_0, x_1]^\top$.

Objective function. The farmer wants to maximize his profits, that is, he wants to

$$\text{maximize} \quad 7000x_0 + 6000x_1.$$

We want a minimization problem, so we negate the objective function and write it on standard LP form:

$$\text{minimize} \quad \underbrace{\begin{bmatrix} -7000 & -6000 \end{bmatrix}}_{v^\top} \underbrace{\begin{bmatrix} x_0 \\ x_1 \end{bmatrix}}_{x}.$$

Constraints. In this problem, we have two constraints. The farmer cannot grow more than he has room for in the field:

$$\underbrace{4000x_0}_{\text{Area used for apples}} + \underbrace{3000x_1}_{\text{Area used for bananas}} \leq \underbrace{100000}_{\text{Available area}}.$$

He cannot exceed the legal amount of fertilizer used:

$$\underbrace{60x_0}_{\text{Fertilizer used for apples}} + \underbrace{80x_1}_{\text{Fertilizer used for bananas}} \leq \underbrace{2000}_{\text{Legal amount of fertilizer}}.$$

We want a constraint on the form $Ax \leq b$, so we write

$$\underbrace{\begin{bmatrix} 4000 & 3000 \\ 60 & 80 \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} x_0 \\ x_1 \end{bmatrix}}_{x} \leq \underbrace{\begin{bmatrix} 100000 \\ 2000 \end{bmatrix}}_{b}$$

Variable bounds. We also have nonnegativity constraints on the optimization variables (he cannot grow negative amounts). These are included in the variable bounds:

$$0 \leq x \leq \infty.$$

Optimization problem. We now have the optimization problem

$$\begin{aligned} &\text{minimize} \quad v^\top x && (12a) \\ &\text{subject to} \quad Ax \leq b && (12b) \\ &\qquad\qquad 0 \leq x \leq \infty && (12c) \end{aligned}$$

The problem is illustrated in fig. 4.

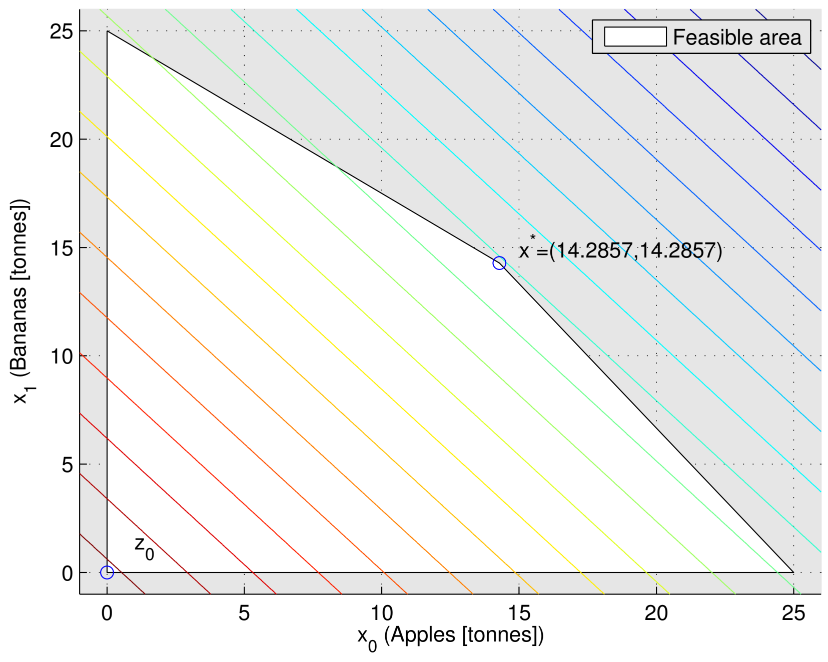


Figure 4: Graphical representation of the farming LP. The optimal point at $x^* = (14.2857, 14.2857)$ and starting point $z_0 = (0, 0)$ are marked with blue circles.

Setting up and solving the problem. We set up a new TestProblem class called FarmingLP. The code below is implemented in the solveProblem() function to set up and solve the farming LP.

```
1  void FarmingLP::solveProblem()
2  {
3      // Variable bounds
4      std::vector< double > lb {0, 0};
5      std::vector< double > ub {INF, INF};
6      // Starting point
7      std::vector< double > z0 {0, 0};
8      // Objective function
9      DenseMatrix v(1, 2);
10     v << -7000, -6000;
11     ObjectivePtr objective(new ObjectiveLinear(v));
12     // Constraints
13     DenseMatrix A(2, 2);
14     A << 4000,  3000,
15             60,    80;
16     DenseVector b(2);
17     b << 100000,
18             2000;
19     ConstraintPtr cLinear(new ConstraintLinear(A, b, false));
20     // Constraint composite
21     ConstraintCompositePtr constraints(new ConstraintComposite
           (2, lb, ub));
22     constraints->add(cLinear);
23     // Optimize
24     OptimizerIpopt optIpopt(objective, constraints, z0);
25     int status = optIpopt.optimize();
26     fopt_found = optIpopt.getObjectiveValue();
27     zopt_found = optIpopt.getOptimalSolution();
28     cout << "Optimal solution: f*=" << fopt_found << endl;
29     cout << "Optimal point: x*=(" << zopt_found.at(0) << "," <<
            zopt_found.at(1) << ")" << endl;
30 }
```

Lines 4-7 declare and fill STL vectors with the lower variable bounds, upper variable bounds and starting point. Lines 8-11 define the objective function. Lines 12-19 define the linear constraints. Lines 20-22 create a constraint composite and inserts the linear constraint. Lines 23-25 creates an Ipopt optimizer object and solves the problem. Lines 26-29 extracts and prints information about the solution. When used in the TestProblem framework, this gives us the printout

```
Running Farming LP problem...

******************************************************************************
This program contains Ipopt, a library for large-scale nonlinear optimization.
Ipopt is released as open source code under the Eclipse Public License (EPL).
For more information visit http://projects.coin-or.org/Ipopt
******************************************************************************
```

```
Optimal solution:  f*=-185714
Optimal point:  x*=(14.2857,14.2857)
Farming LP problem successfully solved in 0 (ms)
Press <RETURN> to close this window...
```

We see that the correct optimal solution is found, as expected. The maximum profit is 185714.

6.2.2 Quadratic programming case

Now consider a case where the price of apples and bananas depends on how much is produced. Now the profit for apples is $7000 - 200x_0$ per tonne (including fertilizer cost) and the profit for bananas is $4000 - 140x_1$ per tonne (including fertilizer cost). All the other parameters of the problem (field size, fertilizer limit etc.) are the same as for the LP case. We want a linear objective function, so we introduce a new variable x_2 to represent the profit:

$$\begin{aligned} x_2 &= (7000 - 200x_0)x_0 + (4000 - 140x_1)x_1 \\ &= \begin{bmatrix} x_0 & x_1 \end{bmatrix} \begin{bmatrix} -200 & 0 \\ 0 & -140 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} + \begin{bmatrix} 7000 & 4000 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \end{bmatrix}. \end{aligned}$$

Optimization variables. We redefine our vector of optimization variables to be $x = [x_0, x_1, x_2]^\top$. Here, x_0 and x_1 represent the same as before, and the new variable x_2 is the profit.

Objective function. Since x_2 represents profit and we want to maximize this, our new objective is

$$\text{minimize} \quad \underbrace{\begin{bmatrix} 0 & 0 & -1 \end{bmatrix}}_{v^\top} \underbrace{\begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix}}_{x}.$$

Constraints. We keep the linear constraint from the LP case:

$$A \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} \le b.$$

We also have to create a constraint to define the relationship between the three variables. We have

$$x_2 - \left(\begin{bmatrix} x_0 & x_1 \end{bmatrix} \begin{bmatrix} -200 & 0 \\ 0 & -140 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} + \begin{bmatrix} 7000 & 4000 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} \right) = 0,$$

which we can write as

$$0 \leq x^\top \underbrace{\begin{bmatrix} 200 & 0 & 0 \\ 0 & 140 & 0 \\ 0 & 0 & 0 \end{bmatrix}}_{P} x + \underbrace{\begin{bmatrix} -7000 & -4000 & 1 \end{bmatrix}}_{q^\top} x \leq 0,$$

that is, we can use the `ConstraintQuadratic` class to define this constraint.

Variable bounds. Both x_0 and x_1 have to be nonnegative as before, but x_2 can in principle also be negative (meaning the farmer is losing money). Therefore, the bounds are

$$\begin{aligned} 0 \leq x_0 \leq \infty \\ 0 \leq x_1 \leq \infty \\ -\infty \leq x_2 \leq \infty \end{aligned}$$

Optimization problem. We now have the optimization problem

$$\begin{aligned} \text{minimize} \quad & v^\top x && \text{(13a)} \\ \text{subject to} \quad & A \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} \leq b, && \text{(13b)} \\ & 0 \leq x^\top P x + q^\top x \leq 0, && \text{(13c)} \\ & 0 \leq \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} \leq \infty, && \text{(13d)} \\ & -\infty \leq x_2 \leq \infty. && \text{(13e)} \end{aligned}$$

The problem is illustrated in fig. 5.

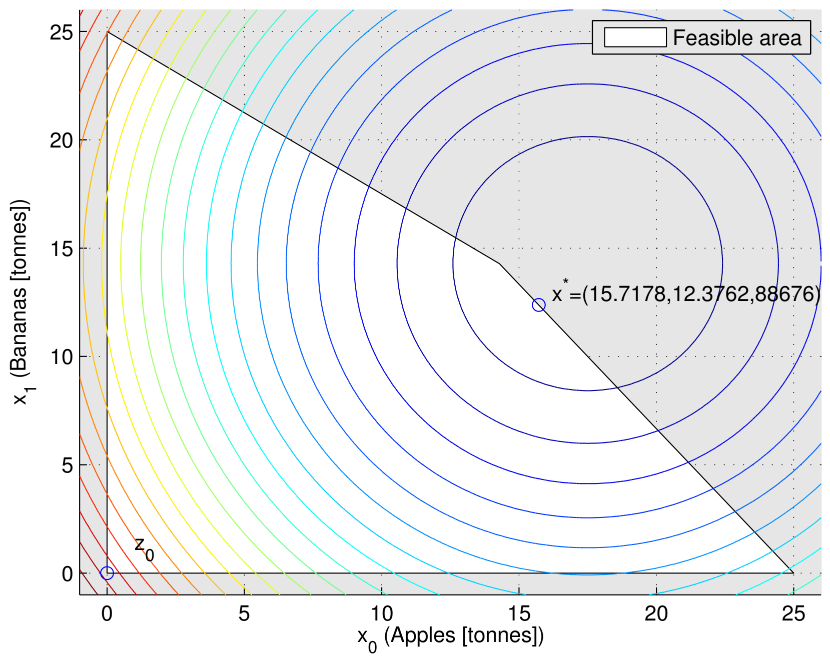


Figure 5: Graphical representation of the farming QP. The optimal point at $x^* = (15.7178, 12.3762, 88676)$ and starting point $z_0 = (0, 0, 0)$ are marked with blue circles.

Setting up and solving the problem. We set up a new `TestProblem` class called `FarmingQP`. The code below is implemented in the `solveProblem()` function to set up and solve the farming QP.

```
void FarmingQP::solveProblem()
{
    // Variable bounds
    std::vector< double > lb {0, 0, -INF};
    std::vector< double > ub {INF, INF, INF};
    // Starting point
    std::vector< double > z0 {0, 0, 0};
    // Objective function
    DenseMatrix v(1, 3);
    v << 0, 0, -1;
    ObjectivePtr objective(new ObjectiveLinear(v));
    // Linear constraint
    DenseMatrix A(2, 2);
    A << 4000,  3000,
           60,    80;
    DenseVector b(2);
    b << 100000,
           2000;
    ConstraintPtr cLinear(new ConstraintLinear(A, b, false));
20      // Linear constraint variable mapping
21      std::vector< int > varMapLinear {0, 1};
22      // Quadratic constraint
23      DenseMatrix P(3, 3);
24      P << 200,   0, 0,
25             0, 140, 0,
26             0,   0, 0;
27      DenseMatrix q(3, 1);
28      q << -7000,
29           -4000,
30               1;
31      ConstraintPtr cQuadratic(new ConstraintQuadratic(P, q, 0,
            0, 0));
32      // Constraint composite
33      ConstraintCompositePtr constraints(new ConstraintComposite
            (3, lb, ub));
34      constraints->add(cLinear, varMapLinear);
35      constraints->add(cQuadratic);
36      // Optimize
37      OptimizerIpopt optIpopt(objective, constraints, z0);
38      int status = optIpopt.optimize();
39      fopt_found = optIpopt.getObjectiveValue();
40      zopt_found = optIpopt.getOptimalSolution();
41      cout << "Optimal solution: f*=" << fopt_found << endl;
42      cout << "Optimal point: x*=(" << zopt_found.at(0) << "," <<
             zopt_found.at(1) << "," << zopt_found.at(2) << ")" <<
            endl;
43  }
```

Lines 3-7 declare and fill STL vectors with the lower variable bounds, upper variable bounds and starting point. Lines 8-11 define the objective function. Lines 12-19 define the linear constraint. Line 21 defines an STL vector of integers which defines the mapping between the variables in the linear constraint (which has two variables) and the variables in the constraint composite (which has three variables). Lines 22-31 define the quadratic constraint. Lines 32-25 create a constraint composite and inserts the linear and quadratic constraints. Lines 36-38 creates an Ipopt optimizer object and solves the problem. Lines 39-42 extracts and prints information about the solution. When used in the `TestProblem` framework, this gives us the printout

```
Running Farming QP problem...

******************************************************************************
This program contains Ipopt, a library for large-scale nonlinear optimization.
Ipopt is released as open source code under the Eclipse Public License (EPL).
For more information visit http://projects.coin-or.org/Ipopt
******************************************************************************

Optimal solution:  f*=-88675.7
Optimal point:  x*=(15.7178,12.3762,88675.7)
Farming QP problem successfully solved in 10 (ms)
Press <RETURN> to close this window...
```

We see that the correct optimal solution is found, as expected. The maximum profit is 88675.7.

6.2.3 Integer variables

We now say that the farmer is only allowed to produce integer amounts of apples and bananas. That is, we impose the additional constraint that x_0 and x_1 are integers. Since the objective function is linear, and the original constraints are convex, this becomes a convex MINLP. This is easily formulated and solved in a Branch-and-Bound framework.

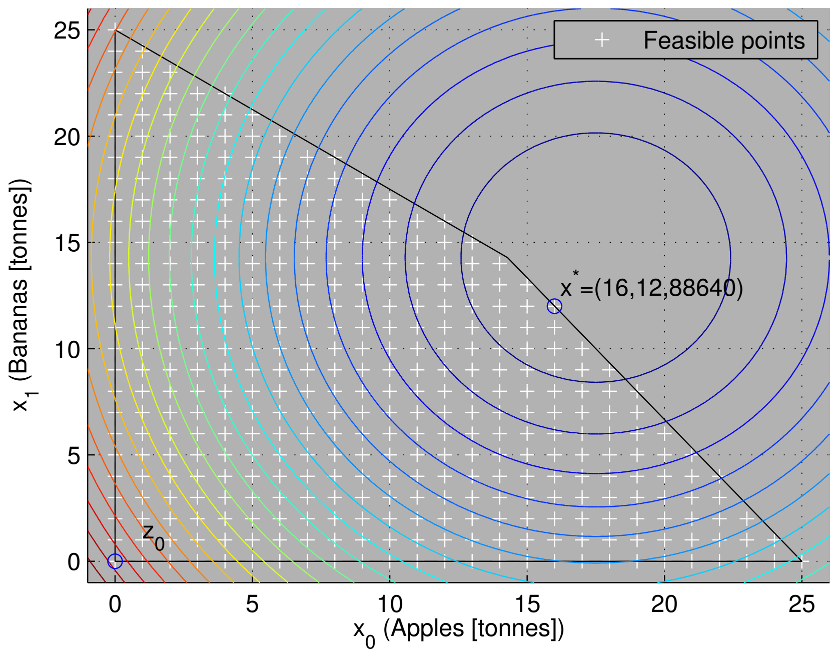


Figure 6: Graphical representation of the farming problem with integer variables. The optimal point at $x^* = (16, 12, 88640)$ and starting point $z_0 = (0, 0, 0)$ are marked with blue circles.

Setting up and solving the problem. We set up a new TestProblem class called FarmingInteger. The code below is implemented in the solveProblem() function to set up and solve the farming problem with integer variables.

```
void FarmingInteger::solveProblem()
{
    // Variable bounds
    std::vector< double > lb {0, 0, -INF};
    std::vector< double > ub {30, 30, INF};
    // Starting point
    std::vector< double > z0 {0, 0, 0};
    // Objective function
    DenseMatrix v(1, 3);
    v << 0, 0, -1;
    ObjectivePtr objective(new ObjectiveLinear(v));
    // Linear constraint
    DenseMatrix A(2, 2);
    A << 4000,  3000,
           60,    80;
    DenseVector b(2);
    b << 100000,
           2000;
    ConstraintPtr cLinear(new ConstraintLinear(A, b, false));
    // Linear constraint variable mapping
    std::vector< int > varMapLinear;
    varMapLinear.push_back(0);
    varMapLinear.push_back(1);
    // Quadratic constraint
    DenseMatrix P(3, 3);
    P << 200,   0, 0,
           0, 140, 0,
           0,   0, 0;
    DenseMatrix q(3, 1);
    q << -7000,
         -4000,
             1;
    ConstraintPtr cQuadratic(new ConstraintQuadratic(P, q, 0,
        0, 0));
    // Constraint composite
    ConstraintCompositePtr constraints(new ConstraintComposite
        (3, lb, ub));
    constraints->add(cLinear, varMapLinear);
    constraints->add(cQuadratic);
    // Variable types
    std::vector< int > variable_types;
    variable_types.push_back(INTEGER); // x0
    variable_types.push_back(INTEGER); // x1
    variable_types.push_back(CONTINUOUS); // x2
    // Branching variables
    std::vector< int > branching_variables;
    branching_variables.push_back(0); // x0
    branching_variables.push_back(1); // x1
    // Optimize
    BranchAndBound bnb(objective, constraints, z0,
        variable_types, branching_variables);
    int status = bnb.optimize();
    fopt_found = bnb.getObjectiveValue();
    zopt_found = bnb.getOptimalSolution();
}
```

This is very similar to the QP case. The differences are:

- In line 5, the upper bounds of x_0 and x_1 are set to 30 instead of ∞. This is because we are branching on these variables, so we must define an upper bound so that the Branch-and-Bound algorithm is able to calculate where to split the variable when branching. 30 is a reasonable choice as an upper bound, because there is not room to grow 30 tons of either fruit.

- Lines 38-42 define the variable types. Here, we have defined that x_0 and x_1 are integer variables, and x_2 (the profit function) is continuous.

- Lines 43-46 define which variables are to be branched on. Typically, we choose our degrees of freedom as branching variables; in our case, the amount of apples x_0 and the amount of bananas x_1.

- In lines 47-49, we use the `BranchAndBound` class instead of the `OptimizerIpopt` class. This is because we have integer variables and need to use branching to solve the problem.

When used in the `TestProblem` framework, this gives us the following output:

```
Branch and Bound tree search finished after 19 iterations, using 0 sec.
Global solution upper bound = -88640
Global solution lower bound = -88640
Optimality gap = 0 <= 0.001 (epsilon)
Optimal point x* = ( 16, 12, 8.864e+04)

Farming QP problem successfully solved in 180 (ms)
Press <RETURN> to close this window...
```

We see that the correct optimal solution is found (quickly). The maximum profit is 88640.

6.3 Network flow problem

Now we will look at some cases of network flow. Throughout this section we will look at a simple flow network with one source vertex, one sink vertex, four internal vertices and ten edges. The flow network is illustrated in fig. 7. We denote the vertices v_i, $i = \{0 \dots 5\}$ and the edges e_j, $j = \{0 \dots 9\}$. For context, we could say the flow network represents a routing network where the edges represent pipelines, and the vertices represent connection points between the pipelines.

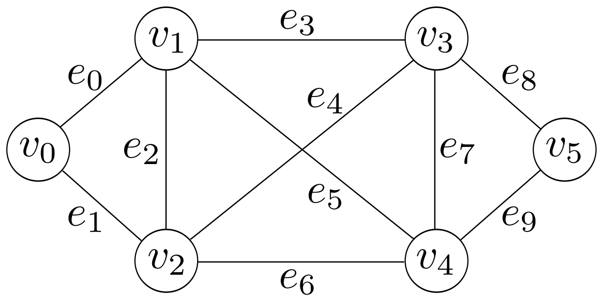


Figure 7: Flow network. v_0 is the source node and v_5 is the sink node.

6.3.1 Simple maximum flow case

Consider a case where our objective is to maximize the flow though the network, and each edge has a constant maximum capacity (see fig. 8).

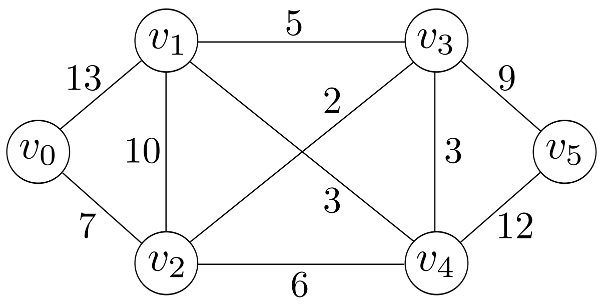


Figure 8: Flow network with constant capacities.

The notation we will use is the following:

- The flow through edge e_j is x_j. Since the flows are our degrees of freedom, these are our optimization variables. We define our vector of optimization variables $x = [x_0, \dots, x_9]^\top$.
- The maximum flow, or capacity, for edge e_j is $\overline{x}_j$. This gives us a vector of capacities $\overline{x} = [13, 7, 10, 5, 2, 3, 6, 3, 9, 12]^\top$. We also define $\underline{x} = -\overline{x}$, meaning the minimum flow is the negated maximum flow.

Incidence matrix. To formulate the optimization problem, we will make use of an *incidence matrix* $A = \{a_{i,j}\}$. The rows of the incidence matrix represent vertices, and the columns represent edges. If edge j leaves vertex i, then $a_{i,j} = -1$. If edge j enters vertex i, then $a_{i,j} = 1$. The incidence matrix for our flow network is shown below.

$$\begin{array}{c} \\ v_0 \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{array} \begin{array}{c} \begin{array}{cccccccccc} e_0 & e_1 & e_2 & e_3 & e_4 & e_5 & e_6 & e_7 & e_8 & e_9 \end{array} \\ \begin{bmatrix} -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & -1 & -1 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & -1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & -1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix} \end{array} = A$$

Note that this incidence matrix could be slightly different since our flow network is undirected (liquid can flow both ways through a pipeline). However, here we have assumed that the default flow direction is right $\rightarrow$ left and up $\rightarrow$ down.

Optimization variables. Our vector of optimization variables is $x = [x_0, \ldots, x_9]^\top$, where x_j is the flow through edge e_j.

Objective function. We want to maximize the flow through the network. Another way of saying this is that we want to maximize the flow into the sink vertex v_5. This flow is the sum of the flows in edges e_8 and e_9, in other words, we want to maximize $x_8 + x_9$. This gives the linear objective

$$\text{minimize} \quad \underbrace{\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & -1 \end{bmatrix}}_{v^\top} \underbrace{\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \end{bmatrix}}_{x}.$$

Note that we could just as well have maximized the flow out of the source vertex, or the flow across any cut in the network.

Constraints. We have two constraints; the mass balance constraint and the flow capacity constraint. We will implement the flow capacity constraint later when we define the variable bounds. The mass balance constraint says that for each vertex (except the source and the sink), the sum of inflows must equal the sum of outflows:

$$\begin{aligned} \text{For vertex } v_1: &\quad x_0 - x_2 - x_3 - x_4 = 0, \\ \text{For vertex } v_2: &\quad x_1 + x_2 - x_5 - x_6 = 0, \\ \text{For vertex } v_3: &\quad x_3 + x_5 - x_7 - x_8 = 0, \\ \text{For vertex } v_4: &\quad x_4 + x_6 + x_7 - x_9 = 0. \end{aligned}$$

To write this compactly, we define the matrix A_{int}, which is the incidence matrix A with the top and bottom rows removed (so that it represents only the internal nodes):

$$A_{\text{int}} = \begin{bmatrix} 1 & 0 & -1 & -1 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & -1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & -1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & -1 \end{bmatrix}.$$

Now we can state the mass balance constraint as a linear system of equations $A_{\text{int}}x = 0$. The source and sink vertices are assumed to have infinite capacity, so these are not represented in any constraints.

Variable bounds. Since the optimization variables are the flows through the edges, we can represent the flow capacity constraints as bounds on the optimization variables:

$$\underline{x} \leq x \leq \overline{x}.$$

Optimization problem. We now have the optimization problem

$$\begin{aligned} \text{minimize} &\quad v^\top x && (14a) \\ \text{subject to} &\quad A_{\text{int}}x = 0, \ \text{(Mass balance)} && (14b) \\ &\quad \underline{x} \leq x \leq \overline{x}. \ \text{(Flow capacity)} && (14c) \end{aligned}$$

Setting up and solving the problem. We set up a new `TestProblem` class called `MaximumFlow`. The code below is implemented in the `solveProblem()` function to set up and solve the maximum flow problem.

```
void MaximumFlow::solveProblem()
{
    // Starting point: zero flow
    std::vector< double > z0 {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
    // Flow capacities (c)
    std::vector< double > lb
            {-13,-7,-10,-5,-3,-2,-6,-3,-9,-12};
    std::vector< double > ub
            { 13, 7, 10, 5, 3, 2, 6, 3, 9, 12};
    // Incidence matrix
    DenseMatrix A(6, 10);
12      A << -1, -1,  0,  0,  0,  0,  0,  0,  0,  0,
13            1,  0, -1, -1, -1,  0,  0,  0,  0,  0,
14            0,  1,  1,  0,  0, -1, -1,  0,  0,  0,
15            0,  0,  0,  1,  0,  1,  0, -1, -1,  0,
16            0,  0,  0,  0,  1,  0,  1,  1,  0, -1,
17            0,  0,  0,  0,  0,  0,  0,  0,  1,  1;
18      // Incidence matrix for internal nodes
19      DenseMatrix A_int = A.block(1,0,4,10);
20      // Objective function
21      DenseMatrix v(1, 10);
22      v << 0, 0, 0, 0, 0, 0, 0, 0, -1, -1;
23      ObjectivePtr objective(new ObjectiveLinear(v));
24      // Mass balance constraint
25      DenseVector zeros; zeros.setZero(4,1);
26      ConstraintPtr cMassBalance(new ConstraintLinear(A_int,
            zeros, true));
27      // Constraint composite (with flow capacity as bounds)
28      ConstraintCompositePtr constraints(new ConstraintComposite
            (10, lb, ub));
29      constraints->add(cMassBalance);
30      // Optimize
31      OptimizerIpopt optIpopt(objective, constraints, z0);
32      int status = optIpopt.optimize();
33      fopt_found = optIpopt.getObjectiveValue();
34      zopt_found = optIpopt.getOptimalSolution();
35      cout << "Optimal solution: f*=" << fopt_found << endl;
36      cout << "Optimal flows:" << endl;
37      for (int i = 0; i < 10; i++) cout << "Edge " << i << ": "
            << zopt_found.at(i) << endl;
38  }
```

Line 4 defines the starting point, which we have chosen to be zero flow. Lines 6-9 define the variable bounds, which are also the edge capacities. Lines 10-19 define the incidence matrix and the internal node incidence matrix. Lines 20-23 define the objective function. Lines 24-26 define the mass balance constraint, and lines 27-29 define the constraint composite and adds the mass balance constraint. Lines 30-37 solves the problem and prints information about the solution. When used in the TestProblem framework, this gives us the following output:

```
Running Maximum Flow problem...

******************************************************************************
This program contains Ipopt, a library for large-scale nonlinear optimization.
Ipopt is released as open source code under the Eclipse Public License (EPL).
For more information visit http://projects.coin-or.org/Ipopt
******************************************************************************

Optimal solution:  f*=-16
Optimal flows:
Edge 0:  10.363
Edge 1:  5.63704
Edge 2:  2.36296
Edge 3:  5
Edge 4:  3
Edge 5:  2
Edge 6:  6
Edge 7:  0.196329
Edge 8:  6.80367
Edge 9:  9.19633
Maximum Flow problem successfully solved in 0 (ms)
Press <RETURN> to close this window...
```

We see that the four edges across the middle of the network are at full capacity, and it is not possible to increase the flow across this cut further, meaning this solution must be optimal (however, it is not the only optimal solution).

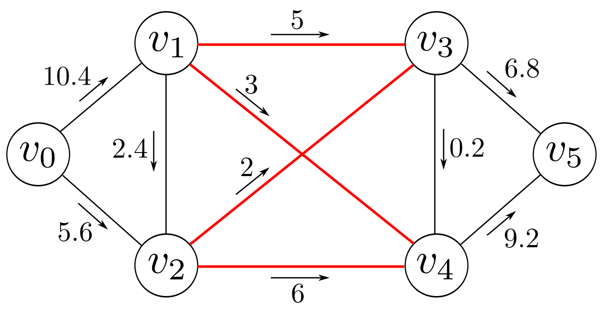


Figure 9: Optimal solution of maximum flow problem. Red edges indicate that the edge is at its maximum capacity.

6.3.2 Maximum flow with routing

Now consider a case where we have to choose exactly two of the four edges across the network (e_3 - e_6). To do this, we introduce four new *binary* optimization variables $b = [b_3, b_4, b_5, b_6]$ (we also augment v with four zeros). When $b_j = 1$, this means we allow flow through edge e_j, and when $b_j = 0$, the flow through edge e_j is locked to $x_j = 0$. That is,

$$x_j \in \begin{cases} [\underline{x}_j, \overline{x}_j], & b_j = 1, \\ \{0\}, & b_j = 0. \end{cases}$$

We can accomplish this by *collapsing* the variable bounds based on the value of the binary variables. Consider the following:

$$b_j \underline{x}_j \leq x_j \leq b_j \overline{x}_j.$$

When $b_j = 1$, we have $\underline{x}_j \leq x_j \leq \overline{x}_j$ as usual. However, when $b_j = 0$, we have $0 \leq x_j \leq 0$, meaning we must have $x_j = 0$. This is called *collapsing the* *bounds*. We can split this into two equations and write it on matrix form:

$$\begin{array}{l} b_j\underline{x}_j - x_j \leq 0 \\ x_j - b_j\overline{x}_j \leq 0 \end{array} \quad \Rightarrow \quad \begin{bmatrix} -1 & \underline{x}_j \\ 1 & -\overline{x}_j \end{bmatrix} \begin{bmatrix} x_j \\ b_j \end{bmatrix} \leq 0. \tag{15}$$

We want only two pipelines to be switched on, so we add the constraint

$$b_3 + b_4 + b_5 + b_6 = 2. \tag{16}$$

Applying (15) to all four routing options and combining this with (16), we get the linear inequality constraint

$$\underbrace{\begin{bmatrix} -1 & 0 & 0 & 0 & \underline{x}_3 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & \underline{x}_4 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 & \underline{x}_5 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & 0 & \underline{x}_6 \\ 1 & 0 & 0 & 0 & -\overline{x}_3 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & -\overline{x}_4 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & -\overline{x}_5 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & -\overline{x}_6 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & -1 & -1 & -1 & -1 \end{bmatrix}}_{A_R} \underbrace{\begin{bmatrix} x_3 \\ x_4 \\ x_5 \\ x_6 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \end{bmatrix}}_{\tilde{x}} \leq \underbrace{\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 2 \\ -2 \end{bmatrix}}_{b_R}$$

Optimization problem. Our new optimization problem becomes

$$\text{minimize} \quad v^\top \begin{bmatrix} x \\ b \end{bmatrix} \tag{17a}$$

$$\text{subject to} \quad A_{\text{int}}x = 0, \ \text{(Mass balance)} \tag{17b}$$

$$A_R\tilde{x} \leq b_R, \ \text{(Routing)} \tag{17c}$$

$$\underline{x} \leq x \leq \overline{x}. \ \text{(Flow capacity)} \tag{17d}$$

Setting up and solving the problem. We set up a new `TestProblem` class called `FlowWithRouting`. The code below is implemented in the `solveProblem()` function to set up and solve the flow problem with routing constraints.

```
void FlowWithRouting::solveProblem()
{
    // Starting point: zero flow
    std::vector< double > z0 {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0};
    // Flow capacities (c)
    std::vector< double > lb {-13, -7, -10, -5, -3, -2, -6, -3,
        -9, -12, 0, 0, 0, 0};
    std::vector< double > ub { 13,  7,  10,  5,  3,  2,  6,  3,
         9,  12, 1, 1, 1, 1};
    // Incidence matrix
    DenseMatrix A(6, 10);
    A << -1, -1,  0,  0,  0,  0,  0,  0,  0,  0,
          1,  0, -1, -1, -1,  0,  0,  0,  0,  0,
          0,  1,  1,  0,  0, -1, -1,  0,  0,  0,
          0,  0,  0,  1,  0,  1,  0, -1, -1,  0,
          0,  0,  0,  0,  1,  0,  1,  1,  0, -1,
          0,  0,  0,  0,  0,  0,  0,  0,  1,  1;
    // Incidence matrix for internal nodes
    DenseMatrix A_int = A.block(1,0,4,10);
    // Objective function
    DenseMatrix v(1, 14);
    v << 0, 0, 0, 0, 0, 0, 0, 0, -1, -1, 0, 0, 0, 0;
    ObjectivePtr objective(new ObjectiveLinear(v));
    // Mass balance constraints
    std::vector< int > massBalanceVars {0,1,2,3,4,5,6,7,8,9};
    VecD zeros4; zeros4.setZero(4, 1);
    ConstraintPtr cMassBalance(new ConstraintLinear(A_int,
        zeros4, true));
    // Routing constraints
    DenseMatrix AR(10, 8);
    AR << -1, 0, 0, 0, lb.at(3),        0,        0,        0,
           0,-1, 0, 0,        0, lb.at(4),        0,        0,
           0, 0,-1, 0,        0,        0, lb.at(5),        0,
           0, 0, 0,-1,        0,        0,        0, lb.at(6),
           1, 0, 0, 0,-ub.at(3),        0,        0,        0,
           0, 1, 0, 0,        0,-ub.at(4),        0,        0,
           0, 0, 1, 0,        0,        0,-ub.at(5),        0,
           0, 0, 0, 1,        0,        0,        0,-ub.at(6),
           0, 0, 0, 0,        1,        1,        1,        1,
           0, 0, 0, 0,       -1,       -1,       -1,       -1;
    VecD bR; bR.setZero(10, 1);
    bR(8) = 2;
    bR(9) = -2;
    // Variables x3-x6 and b3-b6
    std::vector< int > routingVars {3,4,5,6,10,11,12,13};
    ConstraintPtr cRouting(new ConstraintLinear(AR,bR,false));
    // Constraint composite (with flow capacity as bounds)
    ConstraintCompositePtr constraints(new ConstraintComposite
        (14, lb, ub));
    constraints->add(cMassBalance, massBalanceVars);
    constraints->add(cRouting, routingVars);
    // Variable types
    std::vector< int > variable_types;
    for (int i = 0; i < 10; i ++) variable_types.push_back(
        CONTINUOUS);
    for (int i = 0; i < 4; i ++) variable_types.push_back(
        BINARY);
    // Branching variables (routing options)
    std::vector< int > branching_variables {10, 11, 12, 13};
    // Optimize
    BranchAndBound bnb(objective, constraints, z0,
        variable_types, branching_variables);
    int status = bnb.optimize();
    fopt_found = bnb.getObjectiveValue();
```

```
58      zopt_found = bnb.getOptimalSolution();
59      cout << "Optimal solution: f*=" << fopt_found << endl;
60      cout << "Optimal flows:" << endl;
61      for (int i = 0; i < 10; i++) cout << "Edge " << i << ": "
            << zopt_found.at(i) << endl;
62      cout << "Routing decision:" << endl;
63      for (int i = 10; i < 14; i++) cout << "Edge " << i-7 << ":
            " << zopt_found.at(i) << endl;
64  }
```

The differences between this and the previous example is the following:

- Lines 3-7: Starting point and bound vectors are augmented with four elements to facilitate the new routing variables.
- Line 23: We have defined variable mapping vector for the mass balance constraints, since this constraint no longer includes all the variables.
- Lines 27-42 is the implementation of equation (17c); that is, the routing constraints.
- Lines 45-46 add both constraint pointers to the constraint composite (in the previous example only one constraint pointer was added).
- Lines 47-50 define the variable types; variables 0 to 9 (x_0-x_9) are continuous, while variables 10 to 13 (b_3-b_6) are binary.
- Line 52 define the branching variables; these are the binary variables b_3-b_6.
- Lines 54-55: We use the `BranchAndBound` solver in stead of `OptimizerIpopt`, since we have a MILP with integer variables which cannot be solved directly in Ipopt.

This code gives the output

```
Optimal solution: f*=-11
Optimal flows:
Edge 0: 7.526
Edge 1: 3.474
Edge 2: 2.526
Edge 3: 5
Edge 4: 0
Edge 5: 1.957e-08
Edge 6: 6
Edge 7: 0.2101
Edge 8: 4.79
Edge 9: 6.21
Routing decision:
Edge 3: 1
Edge 4:  0
Edge 5:  0
Edge 6:  1
Maximum Flow with Routing problem successfully solved in 370 (ms)
Press <RETURN> to close this window...
```

This solution is shown graphically in fig. 10. As expected, the two edges with the largest capacity are selected and the maximum flow is 11.

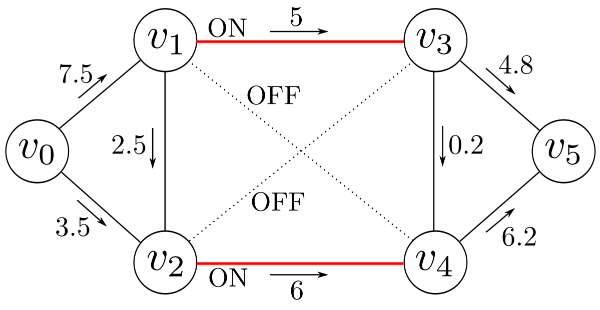


Figure 10: Optimal solution of maximum flow problem with routing constraints. Red edges indicate that the edge is at its maximum capacity and dotted lines indicate that they have been swithced off.

Integer variables. If we want a solution with integer flows, this is easily achieved by specifying x_0-x_9 as integer variables and adding them as branching variables in lines 48-53:

```
// Variable types
    std::vector< int > variable_types;
    for (int i = 0; i < 10; i ++) variable_types.push_back(
        INTEGER);
    for (int i = 0; i < 4; i ++) variable_types.push_back(
        BINARY);
    // Branching variables (routing options)
    std::vector< int > branching_variables
        {0,1,2,3,4,5,6,7,8,9,10,11,12,13};
```

This gives the solution shown in fig. 11.

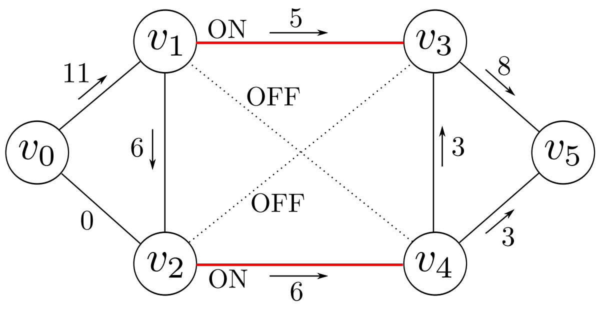


Figure 11: Optimal solution of maximum flow problem with routing constraints and integer variables.

6.3.3 Pressure driven flow case - Linear flow model (INCOMPLETE)

Now we will augment the network model to include pressure/potential. We assume a constant pressure in the source vertex, and a constant pressure in the sink vertex. Given a set of equations which describe the pressure drop across each edge/pipeline, we can formulate a maximum flow problem which takes into account the pressure loss in the pipelines. For this example, we assume the flow x_j through edge e_j is given by

$$x_j = f(p_A, p_B),$$

where p_A is the pressure in the start vertex and p_B is the pressure in the end vertex (see fig. 12)

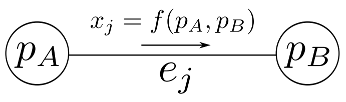


Figure 12: Flow model.

In this section, we will assume a simple model where the flow is a linear function of the pressure drop, that is,

$$f(p_A, p_B) = k(p_A - p_B).$$

Optimization variables. We have to add the pressures in each vertex to the vector of optimization variables. Our new vector of optimization variables is $[x^\top, p^\top]^\top = [x_0, \ldots, x_9, p_0, \ldots, p_5, b_3, \ldots b_6]^\top$, where p_i is the pressure in vertex v_i.

Objective function. The objective function is the same as in the previous example, but we must augment v with six zeros to facilitate the new optimization variables: $v = [0, 0, 0, 0, 0, 0, 0, 0, -1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]^\top$.

Constraints. We keep the mass balance $A_{\text{int}}x = 0$, and add the flow equation constraints

$$\begin{aligned}
x_0 &= k_0(p_0 - p_1)\\
x_1 &= k_1(p_0 - p_2)\\
x_2 &= k_2(p_1 - p_2)\\
x_3 &= k_3(p_1 - p_3)\\
x_4 &= k_4(p_1 - p_4)\\
x_5 &= k_5(p_2 - p_3)\\
x_6 &= k_6(p_2 - p_4)\\
x_7 &= k_7(p_3 - p_4)\\
x_8 &= k_8(p_3 - p_5)\\
x_9 &= k_9(p_4 - p_5).
\end{aligned}$$

We can make use of our incidence matrix and collect the k_j's in a diagonal matrix $K = \text{diag}\,\{k_0, \ldots, k_9\}$, and write this as $-KA^\top p = x$ (we leave it to the reader to verify this), which gives us the constraint

$$\begin{bmatrix} -I & -KA^\top \end{bmatrix} \begin{bmatrix} x \\ p \end{bmatrix} = 0.$$

Variable bounds. We keep the flow capacity constraints as bounds on x and let the pressures in the internal vertices be unbounded. However, we want constant pressures in the source vertex and the sink vertex. We let the pressure in the source be 10, and demand the pressure in the sink be greater than 0. This is accomplished by using the upper and lower bounds for these two pressures.

Optimization problem. We now have the optimization problem

$$\text{minimize} \quad v^\top \begin{bmatrix} x \\ p \\ b \end{bmatrix} \tag{18a}$$

$$\text{subject to} \quad A_{\text{int}}x = 0, \text{ (Mass balance)} \tag{18b}$$

$$A_R\tilde{x} \le b_R, \text{ (Routing)} \tag{18c}$$

$$\begin{bmatrix} -I & -KA^\top \end{bmatrix} \begin{bmatrix} x \\ p \end{bmatrix} = 0, \text{ (Flow equations)} \tag{18d}$$

$$-c \le x \le c, \text{ (Flow capacity)} \tag{18e}$$

$$10 \le p_0 \le 10, \text{ (Source pressure)} \tag{18f}$$

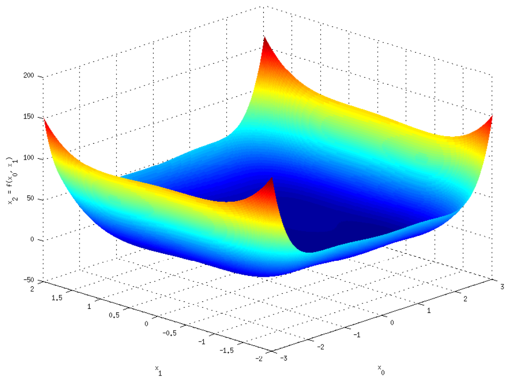
$$0 \le p_5 \le \infty. \text{ (Sink pressure)} \tag{18g}$$

6.4 The Six-Hump Camelback function

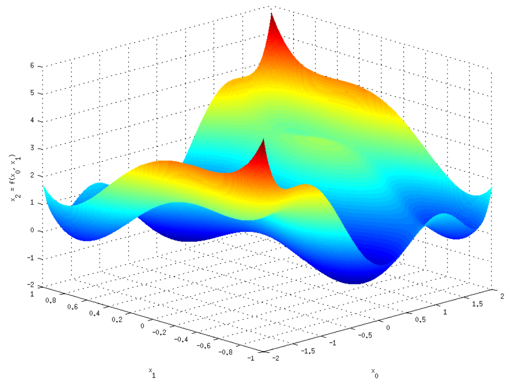
In this section we will minimize the Six-Hump Camelback function locally and globally, using various methods. The Six-Hump Camelback function is a classical test function for optimization algorithms, and is found many places in the literature. The function has two variables, and is given by

$$f_{\mathrm{SH}}(x_0, x_1) = \left(4 - 2.1x_0^2 + \frac{x_0^4}{3}\right) x_0^2 + x_0 x_1 + (-4 + 4x_1^2)x_1^2. \tag{19}$$

The interesting area of this function is in the domain given by $-3 \le x_0 \le 3$ and $-2 \le x_1 \le 2$. Within this domain, there are six local minima, two of which are global minima. The global minimum is $f_{\mathrm{SH}}^* = f_{\mathrm{SH}}(x_0^*, x_1^*) = -1.0316$ at $(x_0^*, x_1^*) = (-0.0898, 0.7126)$ and $(0.0898, -0.7126)$.



(a) $-3 \le x_0 \le 3$ and $-2 \le x_1 \le 2$



(b) A closer look: $-2 \le x_0 \le 2$ and $-1 \le x_1 \le 1$

Figure 13: The Six-Hump Camelback function.

6.4.1 Formulating the optimization problem

We want to find the global minimum of the Six-Hump Camelback function, that is, we want to solve the unconstrained optimization problem

$$\text{minimize} \quad x_2 = \left(4 - 2.1x_0^2 + \frac{x_0^4}{3}\right) x_0^2 + x_0 x_1 + (-4 + 4x_1^2)x_1^2 \tag{20}$$

Note that we have introduced the new variable $x_2 = f_{\mathrm{SH}}(x_0, x_1)$. The reason for this is that we want to state (20) in its epigraph form (see section 1.4.1). We define a vector of optimization variables $x = [x_0, x_1, x_2]^\top$. Since we want to minimize x_2, this gives the linear objective function

$$f(x) = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = v^\top x. \tag{21}$$

We want to restrict x_2 to the epigraph of f_{SH}, so we add the constraint $x_2 \geq f_{\mathrm{SH}}(x_0, x_1)$, which is equivalent to $f_{\mathrm{SH}}(x_0, x_1) - x_2 \leq 0$. Since we want both an upper and lower bound on our constraint function, we add the lower bound $-\infty$ and end up with the reformulated optimization problem

$$\text{minimize} \quad f(x) = v^\top x \tag{22a}$$

$$\text{subject to} \quad -\infty \leq c_1(x_0, x_1, x_2) = f_{\mathrm{SH}}(x_0, x_1) - x_2 \leq 0. \tag{22b}$$

Note that setting the lower bound equal to zero would give the same solution; this would restrict x_2 to lie *on* the graph of f_{SH} as opposed to *on or above* the graph of f_{SH}.

6.4.2 Domain bounds, starting point, objective function and constraint set.

Defining the variable domain bounds, starting point, objective function and constraint set will be the same for all the different approaches below, so we will start with these.

Define domain bounds. First, we define our domain bounds. We have restricted x_0 to ± 3 and x_1 to ± 2, while x_2 is our free variable and should be unrestricted ($\pm\infty$). The domain bounds are defined using vectors of doubles which are passed to the constraint composite (see section **??**).

```
std::vector<double> lb; // Lower bound
std::vector<double> ub; // Upper bound

lb.push_back(-3);        // Lower bound for x0
lb.push_back(-2);        // Lower bound for x1
lb.push_back(-IPOPT_UNBOUNDED); // Lower bound for x2
```

```
ub.push_back(3);        // Upper bound for x0
ub.push_back(2);        // Upper bound for x1
ub.push_back(IPOPT_UNBOUNDED);  // Upper bound for x2
```

Define the starting point. The starting point for the optimization is also defined using a vector of doubles. Since we are solving the problem locally, the starting point will affect which solution we end up with. When following this example, the reader should experiment with several different starting points to see how the solver behaves.

```
std::vector<double> z0; // Starting point

z0.push_back(0);    // Starting point, x0 coordinate
z0.push_back(0);    // Starting point, x1 coordinate
z0.push_back(0);    // Starting point, x2 coordinate
```

Define the objective function. The objective function is defined using the `ObjectiveLinear` class (since we have a linear objective function). The constructor for this class takes an Eigen matrix as an input parameter, which is the vector v from (21):

```
int numVars = 3;    // Number of variables (x0, x1 and x2)
DenseMatrix v(1,numVars); // 1 x 3 matrix (row vector)
v << 0, 0, 1;       // Fill v with values
// Create the objective object
ObjectivePtr objective (new ObjectiveLinear(v));
```

Create a `ConstraintComposite` object. To create the `ConstraintComposite` object, we need the number of variables, and the upper and lower bounds for these variables. These were defined above, so we can proceed with creating a constraint composite to which we can add constraints later.

```
ConstraintCompositePtr constraints(new ConstraintComposite(
    numVars, lb, ub));
```

6.4.3 Local optimization using the `ConstraintPolynomial` class

Note that f_{SH} can be written as a sum of monomial terms:

$$f_{\text{SH}}(x_0, x_1) = \frac{x_0^6}{3} - 2.1x_0^4 + 4x_0^2 + x_0x_1 + 4x_1^4 - 4x_1^2.$$

This means we can write (22b) on the form (8) and implement the Six-Hump Camelback function as a polynomial constraint. First, we need to find the vector c and the matrix E used in (8). We have that

$$\begin{aligned}&\frac{x_0^6}{3} - 2.1x_0^4 + 4x_0^2 + x_0x_1 + 4x_1^4 - 4x_1^2 - x_2\\ &= \frac{1}{3}x_0^6x_1^0x_2^0 - 2.1x_0^4x_1^0x_2^0 + 4x_0^2x_1^0x_2^0\\ &+ x_0^1x_1^1x_2^0 + 4x_0^0x_1^4x_2^0 - 4x_0^0x_1^2x_2^0 - x_0^0x_1^0x_2^1,\end{aligned}$$

so we get

$$c = \begin{bmatrix} \frac{1}{3} \\ -2.1 \\ 4 \\ 1 \\ 4 \\ -4 \\ -1 \end{bmatrix} \quad \text{and} \quad E = \begin{bmatrix} 6 & 0 & 0 \\ 4 & 0 & 0 \\ 2 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 4 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{23}$$

We are now ready to solve the optimization problem using CENSO. To do this, we complete the following steps:

- Add the constraint header file
- Define the Six-Hump Camelback constraint
- Add the constraint to the constraint composite
- Solve the optimization problem with a local solver

Add the constraint header file. The first thing we must do is to include the `ConstraintPolynomial` header file, since we will be using this class to define our constraint.

```
#include "OptimizationProblem/constraintpolynomial.h"
```

Define the Six-Hump Camelback constraint. As mentioned above, we will use the `ConstraintPolynomial` class to define the Six-Hump Camelback function using the vector c and the matrix E from (23). c must be defined as an Eigen vector and E must be defined as an Eigen matrix:

```
DenseVector c(7);        // 7 X 1 Eigen vector
DenseMatrix E(7, 3);     // 7 X 3 Eigen matrix

// Fill c and E with values using the << operator
c << (1/3.0), -2.1, 4, 1, 4, -4, -1;
E <<    6, 0, 0,
        4, 0, 0,
        2, 0, 0,
        1, 1, 0,
        0, 4, 0,
        0, 2, 0,
        0, 0, 1;
```

Now we can create the ConstraintPolynomial object. The third and fourth parameters passed to the constructor are the constraint's lower and upper bound from (22b), which are $-\infty$ and 0, respectively.

```
ConstraintPtr cSixHump(new ConstraintPolynomial(c, E, -
    IPOPT_UNBOUNDED, 0));
```

Add the constraint to the ConstraintComposite object. Now that we have defined a constraint object, we need to add this to the ConstraintComposite object that we created earlier. To do this, we have to define a variable mapping to tell the ConstraintComposite object which variables are present in the constraint being added. In this case, all the variables (x_0, x_1 and x_2) are present, so we create a vector of integers which contains the indices of each variable (0, 1 and 2):

```
std::vector<int> variableMapping;
variableMapping.push_back(0);
variableMapping.push_back(1);
variableMapping.push_back(2);
```

When this vector of integers is passed to the constraint composite object, we are telling it that variable 0 in the constraint composite corresponds to variable 0 in the polynomial constraint that is being added, and the same goes for variables 1 and 2. Now that the variable mapping vector has been created, we are ready to add our constraint to the constraint composite:

```
constraints->add(cSixHump, variableMapping);
```

Solve the optimization problem with a local solver. Since we do not yet have a convex relaxation of our constraint available, we are only able to solve the optimization problem locally. Now that we have defined our objective object, constraint composite object and starting point, CENSO provides a sleek interface to Ipopt for solving optimization problems locally. All that is required is to create an object of type OptimizerIpopt with our objective, constraints and starting point, and call its `optimize()` function:

```
OptimizerIpopt optIpopt(objective, constraints, z0);
int status = optIpopt.optimize();
```

We can also output some information about the solution obtained:

```
cout << "Solved (locally) using Ipopt!" << endl;
cout << "Status: " << status << endl;
cout << "f*: " << optIpopt.getObjectiveValue() << endl;
cout << "x*: ";
printVec(optIpopt.getOptimalSolution());
```

With the starting point $z_0 = (0, 0, 0)$, we should get the following output:

```
Solved (locally) using Ipopt!
Status:  1
```

```
f*:  -9.99e-09
x*:  ( 0, 0, 0)
```

This is not the global optimum; Ipopt is stuck in the saddle point located at the origin. However, if we try different starting points, we will obtain better solutions. For instance, if we give Ipopt the starting point $z_0 = (1, 0, 0)$, it finds the global minimum at $x^* = (0.08984, -0.7127, -1.032)$, and the starting point $z_0 = (-1, 0, 0)$ gives the other globally optimal point at $x^* = (-0.08984, 0.7127, -1.032)$.

6.4.4 Local optimization using a custom constraint class

What if our constraint does not fit into the `ConstraintPolynomial` class or any of the other contstraint classes provided? In this case, we must create a custom constraint class which inherits from the `Constraint` class. If we take a look in the header file of this class, we see that quite a few of the public functions are declared `virtual`. These functions must be implemented in any class which inherits from the `Constraint` class. We will create a new class which inherits from `Constraint` and call it `ConstraintSixHumpCamel`. We start off with the header file `constraintsixhumpcamel.h`:

```
#ifndef CONSTRAINTSIXHUMPCAMEL_H
#define CONSTRAINTSIXHUMPCAMEL_H
#include "constraint.h"

class ConstraintSixHumpCamel : public Constraint
{
public:
    ConstraintSixHumpCamel();
};
#endif // CONSTRAINTSIXHUMPCAMEL_H
```

We will now complete the following steps to implement this constraint class and solve the optimization problem:

- Implement the constructor
- Implement the clone function
- Implement the function evaluation
- Implement the Jacobian evaluation
- Implement the Hessian evaluation
- Define the structure of the Jacobian
- Define the structure of the Hessian
- Create an instance of the constraint
- Add the constraint to the constraint composite

- Run the optimization problem

Implementing the constructor. In the constructor, we must set some constraint properties that are used for various checks. The most basic properties we must set are

- The number of variables (the dimension of the constraint's domain).
- The number of outputs. If there are several equations in one constraint class this number may be different than 1, but in this case there is only one equation.
- The domain bounds of the variables.
- The domain bounds of the output(s).
- Basic constraint properties:
  - Is the Jacobian calculated?
  - Is the Hessian calculated?
  - Is the constraint linear?
  - Is the constraint convex?
  - Is there a convex relaxation available?
- The number of nonzero elements in the Jacobian.
- The number of nonzero elements in the Hessian.

We implement the following constructor:

```
ConstraintSixHumpCamel::ConstraintSixHumpCamel()
{
    // Dimension of domain and codomain
    dimensionDomainF = 3;
    dimensionCodomainF = 1;
    // Set the variable bounds to be unbounded
    setDomainBoundsUnbounded();
    // Set the output bounds between -Inf and 0
    lowerBoundF.push_back(-IPOPT_UNBOUNDED);
    upperBoundF.push_back(0);
    // Gradient: true, Hessian: true, Linear: false, Convex:
       false, Convex relaxation: false
    setConstraintProperties(true,true,false,false,false);
    // Number of nonzero elements in the Jacobian and Hessian
    nnzGradient = 3;
    nnzHessian  = 3;
    // Check settings for obvious mistakes
    checkConstraintSanity();
}
```

Implementing the clone function. When a constraint is added to a constraint composite, what actually happens is that a copy of the constraint is created by calling the desired constraint class `clone()` function. Therefore, we must implement the `clone()` function here. This function simply returns a pointer to a clone of the constraint which is created by the copy constructor, which is a built-in function that creates a new object and copies all the values of each data member from the original to the copy. We implement it directly in the header file:

```
virtual ConstraintSixHumpCamel* clone() const {return new
    ConstraintSixHumpCamel(*this);}
```

Implementing the function evaulation. In the function evaluation function, we simply implement the function given by (22b). We add the declaration as a public function in the header file:

```
public:
    virtual void evalF(DenseVector & x,DenseVector & y);
```

The parameters of the function are

- `x`, which is an Eigen vector of doubles with the values of each variable.
- `y`, which is also an Eigen vector of doubles where we write the function value evaluated at `x`.

Recall that we wish to evaluate the function

$$c_1(x_0, x_1, x_2) = \left(4 - 2.1x_0^2 + \frac{x_0^4}{3}\right) x_0^2 + x_0x_1 + (-4 + 4x_1^2)x_1^2 - x_2.$$

We implement the function in the .cpp file:

```
void ConstraintSixHumpCamel::evalF(DenseVector & x,DenseVector
    & y)
{
    double t1 = (4.0-2.1*pow(x(0),2)+pow(x(0),4)/3.0);
    double t2 = -4+4*pow(x(1),2);

    y(0) = t1*pow(x(0),2)+x(0)*x(1)+t2*pow(x(1),2)-x(2);
}
```

Here, the doubles `t1` and `t2` are only used as temporary storage to make the expression for `y(0)` a little bit tidier.

Implementing the Jacobian evaulation. To implement the Jacobian evaulation, we must first calculate the Jacobian. The Jacobian $\nabla c_1(x)$ is

$$\begin{aligned}\nabla c_1(x) &= \begin{bmatrix} \frac{\partial c_1}{\partial x_0}, & \frac{\partial c_1}{\partial x_1}, & \frac{\partial c_1}{\partial x_2} \end{bmatrix} \\ &= \begin{bmatrix} 8x_0 - 8.4x_0^3 + 2x_0^5 + x_1, & x_0 - 8x_1 + 16x_1^3, & -1 \end{bmatrix}.\end{aligned}$$

When implementing the Jacobian evaluation function, the value of `nnzGradient` defines the number of elements we must calculate. This value was set to three because there are three nonzero elements in the Jacobian. We add the declaration as a public function in the header file:

```
public:
    virtual void evalGradF(DenseVector & x,DenseVector & dx);
```

The parameters of the function are

- `x`, which is an Eigen vector of doubles with the values of each variable.
- `dx`, which is also an Eigen vector of doubles where we write the value of the Jacobian elements evaluated at `x`.

We implement the function in the .cpp file:

```
void ConstraintSixHumpCamel::evalGradF(DenseVector & x,
    DenseVector & dx)
{
    dx(0) = 8*x(0)-8.4*pow(x(0),3)+2*pow(x(0),5)+x(1);
    dx(1) = x(0)-8*x(1)+16*pow(x(1),3);
    dx(2) = -1;
}
```

An important remark here is that `dx(0)` does not necessarily mean the first element of the Jacobian; we will later implement the function `structureGradF`, which maps each element of `dx` to the correct element in the Jacobian.

Implementing the Hessian evaluation. We calculate the Hessian $\nabla^2 c_1(x)$:

$$\nabla^2 c_1(x) = \begin{bmatrix} 8 - 25.2x_0^2 + 10x_0^4 & 1 & 0 \\ 1 & 48x_1^2 - 8 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{24}$$

This Hessian has three nonzero elements, which is the reason we set `nnzHessian` to 3 in the constructor. Although there are technically four nonzero elements, we do not count the symmetric elements of the Hessian, since these are redundant. We add the declaration as a public function in the header file:

```
public:
    virtual void evalHessianF(DenseVector& x,DenseVector & ddx)
        ;
```

The parameters of the function are

- `x`, which is an Eigen vector of doubles with the values of each variable.
- `ddx`, which is also an Eigen vector of doubles where we write the value of the Hessian elements evaluated at `x`.

We implement the function in the .cpp file:

```
void ConstraintSixHumpCamel::evalHessianF(DenseVector& x,
    DenseVector & ddx)
{
     ddx(0) = 8-25.2*pow(x(0),2)+10*pow(x(0),4);
     ddx(1) = 1;
     ddx(2) = 48*pow(x(1),2)-8;
}
```

As with the Jacobian, the elements of `ddx` contain the values of the nonzero elements, and we will define the position of these elements later in the function `structureHessianF`.

Defining the structure of the Jacobian. Next, we will implement the function `structureGradF`, which defines the position of the nonzero elements in the Jacobian. We add the declaration as a public function in the header file:

```
public:
     virtual void structureGradF(std::vector< int >& iRow, std::
         vector< int >& jCol);
```

The parameters of the function are

- `iRow`, which is an STL vector of integers with the row indices of the elements of `dx`.
- `jCol`, which is an STL vector of integers with the column indices of the elements of `dx`.

In our case, the calculated values assume the following positions in the Jacobian:

	Col 0	Col 1	Col 2
Row 0	dx(0)	dx(1)	dx(2)

That is, the value calculated for `dx(0)` should be placed in the (0,0) element of the Jacobian, `dx(1)` should be placed in the (0,1) element and `dx(2)` should be placed in the (0,2) element. This gives the following implementation in the .cpp file:

```
void ConstraintSixHumpCamel::structureGradF(std::vector< int >
    &iRow, std::vector< int > &jCol)
{
     // Position of dx(0) is (0,0)
     iRow.push_back(0); jCol.push_back(0);
     // Position of dx(1) is (0,1)
     iRow.push_back(0); jCol.push_back(1);
     // Position of dx(2) is (0,2)
     iRow.push_back(0); jCol.push_back(2);
}
```

Defining the structure of the Hessian. The function `structureHessianF` defines the position of the nonzero elements in the Hessian. We add the declaration as a public function in the header file:

```
public:
    virtual void structureHessianF(std::vector< int >& eqnr,std
        ::vector< int >& iRow, std::vector< int >& jCol);
```

The parameters of this function are

- `eqnr`, which is an STL vector of integers which defines the equation number associated with the position defined in `iRow`/`jCol`. This equation number is associated with the row number in the Jacobian from which the Hessian is calculated; is the codomain of the constraint function has dimension 1, the equation number is always zero.
- `iRow`, which is an STL vector of integers with the row indices of the elements of `ddx`.
- `jCol`, which is an STL vector of integers with the column indices of the elements of `ddx`.

In our case, the calculated values assume the following positions in the Hessian:

	Col 0	Col 1	Col 2
Row 0	ddx(0)	ddx(1)	–
Row 1	ddx(1)	ddx(2)	–
Row 2	–	–	–

The implementation becomes

```
void ConstraintSixHumpCamel::structureHessianF(std::vector< int
     > &eqnr, std::vector< int > &iRow, std::vector< int > &
    jCol)
{
    // Position of ddx(0) is (0,0)
    eqnr.push_back(0); iRow.push_back(0); jCol.push_back(0);
    // Position of ddx(1) is (0,1) (and (1,0))
    eqnr.push_back(0); iRow.push_back(0); jCol.push_back(1);
    // Position of ddx(2) is (1,1)
    eqnr.push_back(0); iRow.push_back(1); jCol.push_back(1);
}
```

Create an instance of the constraint. We use the following code to create an instance of the constraint class we just created:

```
ConstraintPtr cSixHump(new ConstraintSixHumpCamel());
```

Add the constraint to the `ConstraintComposite` object. Now that we have defined a constraint object, we need to add this to the `ConstraintComposite` object that we created earlier. This is done the same way as for the polynomial constraint:

```
constraints->add(cSixHump, variableMapping);
```

Running the optimization problem. Now we are ready to solve the optimization problem again, with the new constraint class:

```
OptimizerIpopt optIpopt(objective, constraints, z0);
int status = optIpopt.optimize();
```

Of course we can display solution information here the same way as we could when we used the polynomial constraint class.

6.4.5 Global optimization using Alpha-Branch-and-Bound

As explained in section 1.4.4, it is possible to find the global minimum of a function if we apply a convex relaxation of the function in a Branch-and-Bound framework, as long as the relaxation improves as the domain bounds shrink. In this section, we will implement a (pretty bad) convex relaxation of the Six-Hump Camelback function using a technique presented in [1] called Alpha-Branch-and-Bound (αBB). A detailed explanation of this approach is beyond the scope of this manual, but in short, it finds a convex underestimator of the (in our case) Six-Hump Camelback function which depends on the domain bounds. The term *convex underestimator* refers to a function that is convex, and that is less than or equal to the Six-Hump Camelback function for all x within the domain bounds, that is

$$\underline{f}_{\text{SH}}(x) \leq f_{\text{SH}}(x),\ \forall\, x^L \leq x \leq x^U,$$

where $\underline{f}_{\text{SH}}(x)$ is the convex underestimator of $f_{\text{SH}}(x)$. In this example we will use a simple variant, where the convex underestimator is on the form

$$\underline{f}_{\text{SH}}(x) = f_{\text{SH}}(x) + \alpha \sum_i (x_i^L - x_i)(x_i^U - x_i). \tag{25}$$

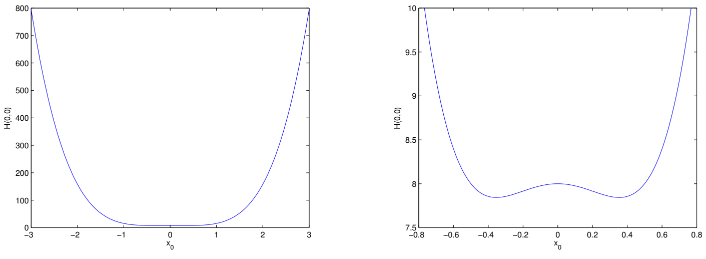
It can be shown that this function is convex if and only if

$$\alpha \geq \max\left\{0, -\frac{1}{2} \min_{i, x^L \leq x \leq x^U} \lambda_i(x)\right\}, \tag{26}$$

where the $\lambda_i(x)$'s are the eigenvalues of $H_f(x)$, the Hessian of $f(x)$. To find the required value of α for our convex relaxation, we need to know the minimum eigenvalue of the Hessian of $f(x)$. This is not trivial to solve. However, Thm. 3.2 in [1] gives us a neat trick that we can use to find a lower bound on the minimum eigenvalue of a matrix $A = \{a_{ij}\}$. It is given by

$$\lambda_{\min} \geq \min_i \left[\underline{a}_{ii} - \sum_{j \neq i} \max(|\underline{a}_{ij}|, |\overline{a}_{ij}|) \right] \tag{27}$$

In other words, if we can find upper and lower bounds on each element of the Hessian, we can also get a lower bound on the minimum eigenvalue. The Hessian of the Six-Hump Camelback function was given in (24). Note that all the elements are constant except for the (0,0) element, which is a function of x_0 only, and the (1,1) element, which is a function of x_1 only. We can exploit this to find upper and lower bounds. In fig. 14 we see the (0,0) element of the Hessian as a function of x_0. We know that both the minimum and maximum of this function must be located either in the end points (x_0^L and x_0^U) or in one of the stationary points, which are located at $(0, 8)$ and $(\pm 1.225, -7.876)$.



(a) $-3 \leq x_0 \leq 3$ (b) $-0.8 \leq x_0 \leq 0.8$

Figure 14: (0,0) element of Hessian defined in (24).

The same goes for the (1,1) element, except that this is even easier since it is only a second degree polynomial, and it is convex (see fig. 15). Here, the minimum and and maximum must be located either in x_1^L, x_1^U, or the stationary point at $(0, -8)$.

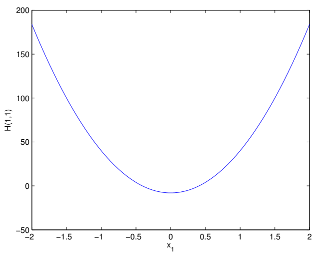


Figure 15: (1,1) element of Hessian defined in (24) $(-2 \leq x_1 \leq 2)$.

We create a new constraint class which we name `ConstraintSixHumpCamelWithRelaxation`, which starts out the same as the `ConstraintSixHumpCamel` class. That is, our starting point is the following header file:

```
#ifndef CONSTRAINTSIXHUMPCAMELWITHRELAXATION_H
#define CONSTRAINTSIXHUMPCAMELWITHRELAXATION_H
#include "constraint.h"

class ConstraintSixHumpCamelWithRelaxation : public Constraint
{
public:
    ConstraintSixHumpCamelWithRelaxation();

    // Clone function - uses copy constructor
    virtual ConstraintSixHumpCamelWithRelaxation* clone() const
         {return new ConstraintSixHumpCamelWithRelaxation(*this
        );}

    virtual void evalF(DenseVector & x,DenseVector & y);

    virtual void evalGradF(DenseVector & x,DenseVector & dx);

    virtual void evalHessianF(DenseVector& x,DenseVector & ddx)
        ;

    virtual void structureGradF(std::vector< int >& iRow, std::
        vector< int >& jCol);

    virtual void structureHessianF(std::vector< int >& eqnr,std
        ::vector< int >& iRow, std::vector< int >& jCol);
};

#endif // CONSTRAINTSIXHUMPCAMELWITHRELAXATION_H
```

We now go through the steps required to modify this class to allow global optimization. These steps are:

- Add a pointer to a relaxed constraint as a private data member
- Modify the constructor
- Implement the copy constructor
- Implement the `setDomainBounds()` function
- Implement the `computeConvexRelaxation()` and `getConvexRelaxation()` functions
- Create a new constraint class to represent the convex relaxation
- Solve the optimization problem in a Branch-and-Bound framework

Add a pointer to a relaxed constraint as a private data member. We add the following in the header file:

```
private:
    ConstraintPtr relaxedConstraint;
```

This is a pointer to the relaxation of the constraint. In the Branch-and-Bound framework, many instances of the constraint will be created, and each instance maintains its own relaxed constraint.

Modify the constructor. We need to make some changes to the constructor:

- We have to set the constraint property `convexRelaxationAvailable` to `true`, to tell the Branch-and-Bound framework that this constraint has a convex relaxation available. The easiest way to accomplish this is to change the corresponding parameter of the `setConstraintProperties` function.

- We have to create a convex relaxation of the constraint along with the constraint itself. To do this, we need to know the domain bounds. Therefore, we add two parameters to the constructor `lb` and `ub`, which are STL vectors of doubles. In the header file, the declaration becomes

  ```
  public:
      ConstraintSixHumpCamelWithRelaxation(std::vector<
         double > lb, std::vector< double > ub);
  ```

  We use these parameters to update the `domainLowerBound` and `domainUpperBound` members, respectively. Now we can create some function `computeConvexRelaxation()` that uses these domain bounds to calculate the convex relaxation, and updates the `relaxedConstraint` member. We will define the function later, for now we just call it in the constructor.

The implementation in the .cpp file becomes:

```
ConstraintSixHumpCamelWithRelaxation::
   ConstraintSixHumpCamelWithRelaxation(std::vector< double >
   lb, std::vector< double > ub):
    relaxedConstraint(nullptr)
{
    // Dimension of domain and codomain
    dimensionDomainF = 3;
    dimensionCodomainF = 1;
    // Check consistency of domain bounds
    assert(lb.size() == ub.size());
    assert(lb.size() == dimensionDomainF);
    // Set domain bounds
    domainLowerBound = lb;
    domainUpperBound = ub;
    // Set the output bounds between -Inf and 0
```

```
    lowerBoundF.push_back(-IPOPT_UNBOUNDED);
    upperBoundF.push_back(0);
    // Gradient: true, Hessian: true, Linear: false, Convex:
        false, Convex relaxation: true
    setConstraintProperties(true, true, false, false, true);
    // Number of nonzero elements in the Jacobian and Hessian
    nnzGradient = 3;
    nnzHessian  = 3;
    // Compute convex relaxation
    computeConvexRelaxation();
    // Check settings for obvious mistakes
    checkConstraintSanity();
}
```

Implement the copy constructor. The default copy constructor only copies the members of the object, not any objects they might be pointing to. In this case, the smart pointer `relaxedConstraint` will be copied, but the `ConstraintSixHumpCamelConvexRelaxation` object it points to will not. This is illustrated in fig. 16. This behaviour will cause problems when one of the constraint objects makes changes to its relaxation, since these changes then will also apply to the copy of the constraint (which has made no such changes to its relaxation).

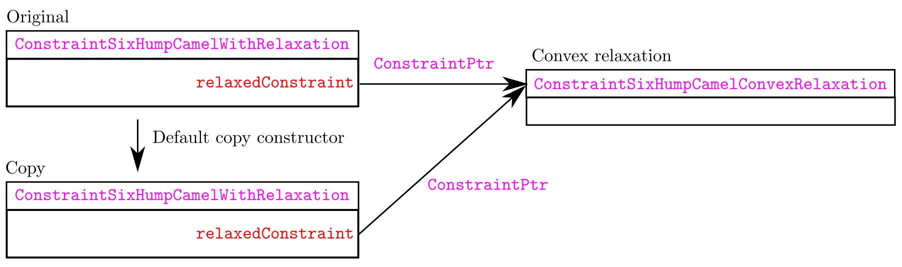


Figure 16: Default copy constructor behaviour.

The solution is to implement a copy constructor that clones the relaxed constraint as well, so that the copy of the constraint gets its very own instance of the relaxed constraint. This is illustrated in fig. 17.

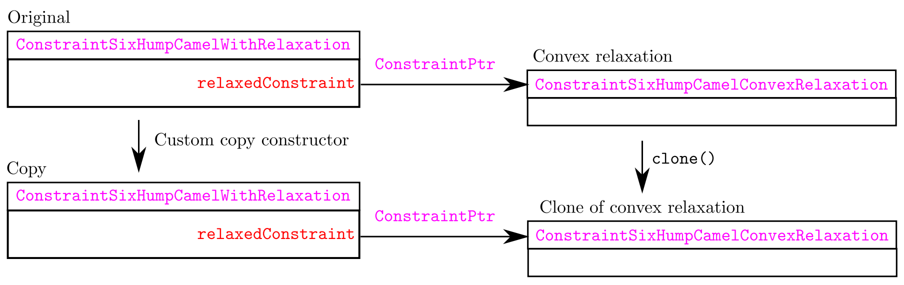

Original
ConstraintSixHumpCamelWithRelaxation
relaxedConstraint
ConstraintPtr
Convex relaxation
ConstraintSixHumpCamelConvexRelaxation
Custom copy constructor
clone()
Copy
ConstraintSixHumpCamelWithRelaxation
relaxedConstraint
ConstraintPtr
Clone of convex relaxation
ConstraintSixHumpCamelConvexRelaxation

Figure 17: Desired copy constructor behaviour.

To accomplish this, we add the declaration in the header file:

```
public:
    ConstraintSixHumpCamelWithRelaxation(
    ConstraintSixHumpCamelWithRelaxation const& copy);
```

and the implementation in the .cpp file:

```
ConstraintSixHumpCamelWithRelaxation::
    ConstraintSixHumpCamelWithRelaxation(
    ConstraintSixHumpCamelWithRelaxation const& copy)
     : Constraint(copy)
{
     if (copy.relaxedConstraint == nullptr)
     {
          relaxedConstraint = nullptr;
     }
     else
     {
          relaxedConstraint = ConstraintPtr(copy.
              relaxedConstraint->clone());
     }
}
```

Note that we have to check whether the relaxation of the constraint we are trying to copy exists or not, before we call the `clone()` function to create a copy of the relaxation. Also, note that we also call the default copy constructor of the `Constraint` class (`: Constraint(copy)`) to copy all the basic constraint properties.

Implement the `setDomainBounds()` function. When the domain bounds are changed, we must also calculate a new convex relaxation, since the relaxation changes when the bounds change. To do this, we overload the `setDomainBounds()` function of the `Constraint` class to include the calculation of a new relaxation:

```
void ConstraintSixHumpCamelWithRelaxation::setDomainBounds(std
    ::vector< double > lb, std::vector< double > ub)
{
     // Check parameters
     assert(lb.size() == dimensionDomainF);
     assert(ub.size() == dimensionDomainF);
     // Avoid unnecessary updates
     if (compareVec(lb,domainLowerBound) && compareVec(ub,
         domainUpperBound)) return;
     // Set domain bounds
     Constraint::setDomainBounds(lb, ub);
     // Recalculate the convex relaxation
     computeConvexRelaxation();
}
```

Implement the `computeConvexRelaxation()` and `getConvexRelaxation()` functions. To implement these functions, we need equations (26) and (27). The function `computeConvexRelaxation()` is declared a private function:

```
private:
    void computeConvexRelaxation();
```

Its implementation looks like this:

```
void ConstraintSixHumpCamelWithRelaxation::
    computeConvexRelaxation()
{
    // Matrices to represent the interval Hessian
    DenseMatrix H_lo, H_hi;
    // Compute interval Hessian (depends on current domain
        bounds)
    intervalHessian(H_lo, H_hi);
    // Calculate lower bound on minimum eigenvalue
    double minEigenVal = minIntervalEigenValue(H_lo, H_hi);
    // Calculate required value for alpha
    double alpha = fmax(0, -0.5*minEigenVal);

    // Create new relaxed constraint based on alpha
    relaxedConstraint = ConstraintPtr(new
        ConstraintSixHumpCamelConvexRelaxation(domainLowerBound
        , domainUpperBound, alpha));
}
```

The first line:

```
// Matrices to represent the interval Hessian
DenseMatrix H_lo, H_hi;
```

declares two matrices `H_lo` and `H_hi`, which represent the lower and upper bounds on the elements of the Hessian (24). These bounds are calculated in the second line:

```
// Compute interval Hessian (depends on current domain bounds)
intervalHessian(H_lo, H_hi);
```

The function `intervalHessian` has the declaration

```
private:
    void intervalHessian(DenseMatrix& H_lo, DenseMatrix& H_hi);
```

and the implementation (refer to the comments to see exactly what the function does):

```
void ConstraintSixHumpCamelWithRelaxation::intervalHessian(
    DenseMatrix &H_lo, DenseMatrix &H_hi)
{
    // The Hessian of the six hump camelback function has the
        following form:
    //
    //     | 8 - 25.2x0^2 + 10x0^4       1         0 |
    // H = |         1              48x1^2 - 8     0 |
    //     |         0                    0        0 |
//
// When the bounds on x0 and x1 are given, it is easy to
// find bounds on the elements of the Hessian matrix.
// The only non-constant elements are the (0,0) element
// and the (1,1) element. The (0,0) element has three
// stationary points at x0 = 0, x0 = +/- 1.2249722
// where the function value is 8 and -7.876, respectively.
// The (1,1) element has one stationary point at x1 = 0,
// where the function value is -8.

// x0 and x1 must be bounded for us to be able to find an
// interval Hessian.
// Upper bound on x0
assert(domainUpperBound.at(0) < IPOPT_UNBOUNDED);
// Upper bound on x1
assert(domainUpperBound.at(1) < IPOPT_UNBOUNDED);
// Lower bound on x0
assert(domainLowerBound.at(0) > -IPOPT_UNBOUNDED);
// Lower bound on x1
assert(domainLowerBound.at(1) > -IPOPT_UNBOUNDED);

// Max/min candidates
std::vector< double > maxMinCandidates;

// Find bounds on (0,0) element
double statPoint1 = -1.2249722;  // Stationary points
double statPoint2 = 0;
double statPoint3 = 1.12249722;

// Extreme points (at lower/upper bound)
double y0L, y0U, x0L, x0U;
x0L = domainLowerBound.at(0);
x0U = domainUpperBound.at(0);
y0L = 8 - 25.2 * pow(x0L, 2) + 10 * pow(x0L, 4);
y0U = 8 - 25.2 * pow(x0U, 2) + 10 * pow(x0U, 4);
maxMinCandidates.push_back(y0L);
maxMinCandidates.push_back(y0U);

// Stationary points are added to candidate points
// if they are within the domain bounds
if (valueWithinBounds(statPoint1, 0)) maxMinCandidates.
    push_back(-7.876);  // Value at x0 = -1.2249722
if (valueWithinBounds(statPoint2, 0)) maxMinCandidates.
    push_back(0);       // Value at x0 = 0
if (valueWithinBounds(statPoint3, 0)) maxMinCandidates.
    push_back(-7.876);  // Value at x0 = 1.2249722

// Find the largest and smalles elements in the vector of
// max/min candidates
double maxElem00 = *std::max_element(maxMinCandidates.begin
    (), maxMinCandidates.end());
double minElem00 = *std::min_element(maxMinCandidates.begin
    (), maxMinCandidates.end());
    maxMinCandidates.clear();

    // Find bounds on (1,1) element
    double y1L, y1U, x1L, x1U;
    x1L = domainLowerBound.at(1);
    x1U = domainUpperBound.at(1);
    y1L = 48 * pow(x1L, 2) - 8;
    y1U = 48 * pow(x1U, 2) - 8;
    maxMinCandidates.push_back(y1L);
    maxMinCandidates.push_back(y1U);
    if (valueWithinBounds(0, 1)) maxMinCandidates.push_back(-8)
        ;  // Value at x1 = 0

    // Find the largest and smalles elements in the vector of
    // max/min candidates
    double maxElem11 = *std::max_element(maxMinCandidates.begin
        (), maxMinCandidates.end());
    double minElem11 = *std::min_element(maxMinCandidates.begin
        (), maxMinCandidates.end());

    // Fill out upper/lower bounds
    H_lo.setZero(3, 3);
    H_lo(0,0) = minElem00;
    H_lo(0,1) = 1;
    H_lo(1,0) = 1;
    H_lo(1,1) = minElem11;
    H_hi.setZero(3, 3);
    H_hi(0,0) = maxElem00;
    H_hi(0,1) = 1;
    H_hi(1,0) = 1;
    H_hi(1,1) = maxElem11;
}
```

This function uses the function `valueWithinBounds(double value, int varNo)`, which only checks whether `value` lies within the domain bounds of variable number `varNo`. The declaration is

```
private:
    bool valueWithinBounds(double value, int varNo);
```

and the implementation is:

```
bool ConstraintSixHumpCamelWithRelaxation::valueWithinBounds(
    double value, int varNo)
{
    // Check that varNo is a valid variable index
    assert(varNo < dimensionDomainF);
    // Check if value is within bounds, return true/false
    if (domainLowerBound.at(varNo) <= value && domainUpperBound
        .at(varNo) >= value)
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

Now that we have calculated bounds on each element in the Hessian, we can calculate a lower bound on the minimum eigenvalue of the Hessian. This is done in the third line:

```
// Calculate lower bound on minimum eigenvalue
double minEigenVal = minIntervalEigenValue(H_lo, H_hi);
```

The function `minIntervalEigenValue` is nothing else than an application of (27). The declaration is

```
private:
    double minIntervalEigenValue(DenseMatrix& H_lo, DenseMatrix
       & H_hi);
```

and the implementation is

```
double ConstraintSixHumpCamelWithRelaxation::
   minIntervalEigenValue(DenseMatrix &H_lo, DenseMatrix &H_hi)
{
    // Uses Thm. 3.2 in Adjiman et. al (1998) to find a lower
    // bound on the minimum eigenvalue of the interval matrix
    // given by H_lo and H_hi.

    // Check that both matrices are square and that they are
    // the same size
    assert(H_lo.rows() == H_lo.cols());
    assert(H_hi.rows() == H_hi.cols());
    assert(H_lo.rows() == H_hi.rows());
    assert(H_lo.cols() == H_hi.cols());

    int dim = H_lo.rows();

    double minEigenVal = IPOPT_UNBOUNDED;
    for (int i = 0; i < dim; i++)
    {
        double minCandidate = H_lo(i,i);
        for (int j = 0; j < dim; j++)
        {
            if (i != j)
            {
                minCandidate -= fmax( fabs(H_lo(i,j)) , fabs(
                   H_hi(i,j)));
            }
        }
        if (minCandidate < minEigenVal)
        {
            minEigenVal = minCandidate;
        }
    }
    return minEigenVal;
}
```

We use the obtained bound to calculate the required value of α with (26) in line four:

```
// Calculate required value for alpha
double alpha = fmax(0, -0.5*minEigenVal);
```

and pass this value to the constructor of the relaxed constraint class in the fifth and final line:

```
// Create new relaxed constraint based on alpha
    relaxedConstraint = ConstraintPtr(new
        ConstraintSixHumpCamelConvexRelaxation(domainLowerBound
        , domainUpperBound, alpha));
```

The function `getConvexRelaxation()` simply clones the relaxed constraint and returns a pointer to the clone. It has the declaration

```
public:
    virtual Constraint* getConvexRelaxation() const;
```

and the implementation

```
Constraint* ConstraintSixHumpCamelWithRelaxation::
    getConvexRelaxation() const
{
    assert(relaxedConstraint != nullptr);
    return relaxedConstraint->clone();
}
```

Create a new constraint class to represent the convex relaxation. We need to create a new class to represent the convex relaxations. We will call it `ConstraintSixHumpCamelConvexRelaxation`. The implementation of this class will be very similar to the implementation of the `ConstraintSixHumpCamel` class from section 6.4.4, with a few differences:

- The function, Jacobian and Hessian evaluations are obviously slightly different.
- The constructor takes the domain bounds as parameters as these are used in the function evaluations.
- It contains the private data member `alpha`, which is also used in the function evaluations.
- The constraint properties `constraintConvex` and `convexRelaxationAvailable` are set to `true`.

Let the convex relaxation of $c_1(x)$ be denoted $\underline{c}_1(x)$. Then we have

$$\begin{aligned}\underline{c}_1(x) &= f_{\mathrm{SH}}(x_0, x_1) + \alpha(x_0^L - x_0)(x_0^U - x_0) + \alpha(x_1^L - x_1)(x_1^U - x_1) - x_2 \\ &= \left(4 - 2.1x_0^2 + \frac{x_0^4}{3}\right)x_0^2 + x_0x_1 + (-4 + 4x_1^2)x_1^2 \\ &\quad + \alpha(x_0^L - x_0)(x_0^U - x_0) + \alpha(x_1^L - x_1)(x_1^U - x_1) - x_2\end{aligned}$$

The gradient $\nabla \underline{c}_1(x)$ is

$$\nabla \underline{c}_1(x) = \begin{bmatrix} 8x_0 - 8.4x_0^3 + 2x_0^5 + x_1 + 2\alpha x_0 - \alpha(x_0^L + x_0^U) \\ x_0 - 8x_1 + 16x_1^3 + 2\alpha x_1 - \alpha(x_1^L + x_1^U) \\ -1 \end{bmatrix}^\top .$$

The Hessian $\nabla^2 \underline{c}_1(x)$ is

$$\nabla^2 \underline{c}_1(x) = \begin{bmatrix} 8 - 25.2x_0^2 + 10x_0^4 + 2\alpha & 1 & 0 \\ 1 & 48x_1^2 - 8 + 2\alpha & 0 \\ 0 & 0 & 0 \end{bmatrix} .$$

This gives us the header file

```
#ifndef CONSTRAINTSIXHUMPCAMELCONVEXRELAXATION_H
#define CONSTRAINTSIXHUMPCAMELCONVEXRELAXATION_H

#include "constraint.h"

class ConstraintSixHumpCamelConvexRelaxation : public
   Constraint
{
public:
    ConstraintSixHumpCamelConvexRelaxation(std::vector< double
        > lb, std::vector< double > ub, double alpha);

    // Clone function - uses copy constructor
    virtual ConstraintSixHumpCamelConvexRelaxation* clone()
        const {return new
        ConstraintSixHumpCamelConvexRelaxation(*this);}

    virtual void evalF(DenseVector & x,DenseVector & y);

    virtual void evalGradF(DenseVector & x,DenseVector & dx);

    virtual void evalHessianF(DenseVector& x,DenseVector & ddx)
        ;

    virtual void structureGradF(std::vector< int >& iRow, std::
        vector< int >& jCol);

    virtual void structureHessianF(std::vector< int >& eqnr,std
        ::vector< int >& iRow, std::vector< int >& jCol);

private:
    double alpha;
};

#endif // CONSTRAINTSIXHUMPCAMELCONVEXRELAXATION_H
```

and the .cpp file

```
#include "constraintsixhumpcamelconvexrelaxation.h"
```

```
ConstraintSixHumpCamelConvexRelaxation::
    ConstraintSixHumpCamelConvexRelaxation(std::vector< double
    > lb, std::vector< double > ub, double alpha)
{
     dimensionDomainF = 3;
     dimensionCodomainF = 1;

     gradientCalculatedF = true;
     hessianCalculatedF = true;

     assert(lb.size() == ub.size());
     assert(lb.size() == dimensionDomainF);

     domainLowerBound = lb;
     domainUpperBound = ub;

     lowerBoundF.push_back(-IPOPT_UNBOUNDED);
     upperBoundF.push_back(0);

     this->alpha = alpha;

     // Gradient: true, Hessian: true, Linear: false, Convex:
         true, Convex relaxation: true
     setConstraintProperties(true, true, false, true, true);

     nnzGradient = 3;
     nnzHessian  = 3;

     checkConstraintSanity();
}

void ConstraintSixHumpCamelConvexRelaxation::evalF(DenseVector
    & x,DenseVector & y)
{
     double x0L = domainLowerBound.at(0);
     double x0U = domainUpperBound.at(0);
     double x1L = domainLowerBound.at(1);
     double x1U = domainUpperBound.at(1);

     double t1 = (4.0-2.1*pow(x(0),2)+pow(x(0),4)/3.0)*pow(x(0)
         ,2);
     double t2 = x(0)*x(1);
     double t3 = (-4+4*pow(x(1),2))*pow(x(1),2);
     double t4 = alpha*(x0L*x0U-x(0)*(x0L+x0U)+pow(x(0),2));
     double t5 = alpha*(x1L*x1U-x(1)*(x1L+x1U)+pow(x(1),2));

     y(0) = t1+t2+t3+t4+t5-x(2);
}

void ConstraintSixHumpCamelConvexRelaxation::evalGradF(
    DenseVector & x,DenseVector & dx)
{
     double x0L = domainLowerBound.at(0);
    double x0U = domainUpperBound.at(0);
    double x1L = domainLowerBound.at(1);
    double x1U = domainUpperBound.at(1);
    dx(0) = 8*x(0)-8.4*pow(x(0),3)+2*pow(x(0),5)+x(1)+2*alpha*x
        (0)-alpha*(x0L+x0U);
    dx(1) = x(0)-8*x(1)+16*pow(x(1),3)+2*alpha*x(1)-alpha*(x1L+
        x1U);
    dx(2) = -1;
}

void ConstraintSixHumpCamelConvexRelaxation::evalHessianF(
    DenseVector& x,DenseVector & ddx)
{
    ddx(0) = 8-25.2*pow(x(0),2)+10*pow(x(0),4)+2*alpha;
    ddx(1) = 1;
    ddx(2) = -8+48*pow(x(1),2)+2*alpha;
}

void ConstraintSixHumpCamelConvexRelaxation::structureGradF(std
    ::vector< int > &iRow, std::vector< int > &jCol)
{
    iRow.push_back(0); jCol.push_back(0);
    iRow.push_back(0); jCol.push_back(1);
    iRow.push_back(0); jCol.push_back(2);
}

void ConstraintSixHumpCamelConvexRelaxation::structureHessianF(
    std::vector< int > &eqnr, std::vector< int > &iRow, std::
    vector< int > &jCol)
{
    eqnr.push_back(0);  iRow.push_back(0);  jCol.push_back(0);
    eqnr.push_back(0);  iRow.push_back(0);  jCol.push_back(1);
    eqnr.push_back(0);  iRow.push_back(1);  jCol.push_back(1);
}
```

Solve the optimization problem in a Branch-and-Bound framework. We will now use the `ConstraintSixHumpCamelWithRelaxation` class we created in the paragraphs above to solve the Six-Hump Camelback problem globally. First, we create an instance of the constraint and add it to the constraint composite as we have done earlier:

```
ConstraintPtr cSixHump(new ConstraintSixHumpCamelWithRelaxation
    (lb, ub));
constraints->add(cSixHump, variableMapping);
```

In addition to the constraints and objective function, the Branch-and-Bound framework needs to know the variable types (continuous/integer) and which variables should be branced on. In our case, all variables are continuous and x_0 and x_1 are the branching variables. We create two STL integer vectors and insert the correct variable indices:

```
std::vector< int > variable_types;
```

```
variable_types.push_back(CONTINUOUS); // x0
variable_types.push_back(CONTINUOUS); // x1
variable_types.push_back(CONTINUOUS); // x2

std::vector< int > branching_variables;
branching_variables.push_back(0);
branching_variables.push_back(1);
```

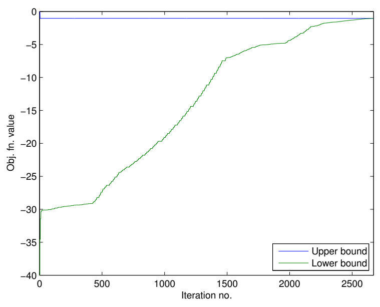
Now we can create a BranchAndBound object, which takes the objective, constraints, starting point, variable types and branching variables as parameters, and call its `optimize()` function to run the Branch-and-Bound algorithm:

```
BranchAndBound bnb(objective, constraints, z0, variable_types,
    branching_variables);
bnb.optimize();
```

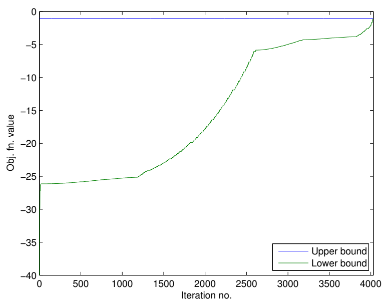
The optimal point and objective function value are available through the `getOptimalSolution()` and `getObjectiveValue()` functions, respectively:

```
zopt_found = bnb.getOptimalSolution();
fopt_found = bnb.getObjectiveValue();
```

The Branch-and-Bound algorithm finds the global optimum, but achieving epsilon-convergence takes a long time due to the poor quality of the convex relaxations. For instance, with the starting point $z_0 = (0,0,0)$, it takes 2671 iterations to achieve epsilon-convergence (see fig. 18(a)), and with the starting point $z_0 = (1,0,0)$ it takes 4031 iterations (fig. 18(b)). However, the global optimal point is found within a few iterations; it is the shrinking of the optimality gap (our certificate of a global solution) that takes a long time.



(a) $z_0 = (0,0,0)$



(b) $z_0 = (1,0,0)$

Figure 18: Branch-and-Bound algorithm progress with the αBB approach.

6.4.6 Global optimization using a B-spline approximation

The methods desribed in the paragraphs above are time consuming and error-prone. An easier approach is to use a B-spline approximation of the Six-Hump Camelback function to solve the problem globally. As mentioned in section 1.4.3, an attractive property of the B-spline is that the control points used to describe the B-spline can be used to construct a convex relaxation of the B-spline. This property has been exploited in the `ConstraintBspline` class, meaning that we can solve problems defined with B-splines globally. To solve the Six-Hump Camelback problem with this approach, we have to complete the following steps:

- Create an `InterpolationTable` object and fill it with samples
- Create a B-spline constraint using the interpolation table
- Add the constraint to the constraint composite
- Solve the optimization problem in a Branch-and-Bound framework

Create an `InterpolationTable` object and fill it with samples. The B-spline constraint constructor takes three parameters; an interpolation table filled with samples, the desired degree of the polynomial pieces from which the B-spline is constructed, and a boolean value to indicate equality or inequality. First, we create a function to evaluate the Six-Hump Camelback function itself:

```
DenseVector sixHumpCamelFunction(DenseVector x)
{
    assert(x.rows() == 2);
    DenseVector y; y.setZero(1);
    y(0) = (4 - 2.1*x(0)*x(0) + (1/3.)*x(0)*x(0)*x(0)*x(0))*x
        (0)*x(0) + x(0)*x(1) + (-4 + 4*x(1)*x(1))*x(1)*x(1);
    return y;
}
```

Now, we use this function to evaluate the Six-Hump Camelback function in a grid (see section **??** for details on the `InterpolationTable` class).

```
InterpolationTable* data = new InterpolationTable(2, 1, false);

double dx = 0.05;
for (double x1 = lb.at(0); x1 <= ub.at(0); x1+=dx)
{
    for (double x2 = lb.at(1); x2 <= ub.at(1); x2+=dx)
    {
        std::vector< double > x;
        x.push_back(x1);
        x.push_back(x2);

        DenseVector xd; xd.setZero(2);
        xd(0) = x1;
        xd(1) = x2;
        DenseVector yd = sixHumpCamelFunction(xd);

        data->addSample(x,yd(0));
    }
}
```

Create a B-spline constraint using the interpolation table. We select a polynomial degree of 3, since this gives us a B-spline which is twice continuously differentiable. We set the third parameter to `true` to indicate that we want an equality constraint.

```
ConstraintPtr cbspline(new ConstraintBspline(*data, 3, true));
```

Add the constraint to the constraint composite. As before,

```
constraints->add(cbspline, variableMapping);
```

Solve the optimization problem in a Branch-and-Bound framework. This is done in the same way as described for the αBB approach; we create two STL vectors with variable types and brancing variables, and pass these to a `BranchAndBound` object along with the objective, constraints and starting point:

```
std::vector< int > variable_types;
variable_types.push_back(CONTINUOUS); // x0
variable_types.push_back(CONTINUOUS); // x1
variable_types.push_back(CONTINUOUS); // x2
std::vector< int > branching_variables;
branching_variables.push_back(0);
branching_variables.push_back(1);

BranchAndBound bnb(objective, constraints, z0, variable_types,
    branching_variables);
bnb.optimize();

zopt_found = bnb.getOptimalSolution();
fopt_found = bnb.getObjectiveValue();
```

This time, the problem is solved in only 49 iterations. The large improvement in performance is due to the improved quality of the convex relaxations. The progress of the algorithm is shown in fig. 19. Note that the initial lower bound is very close to the upper bound (< 0.01 as opposed to > 25-30 for the αBB approach).

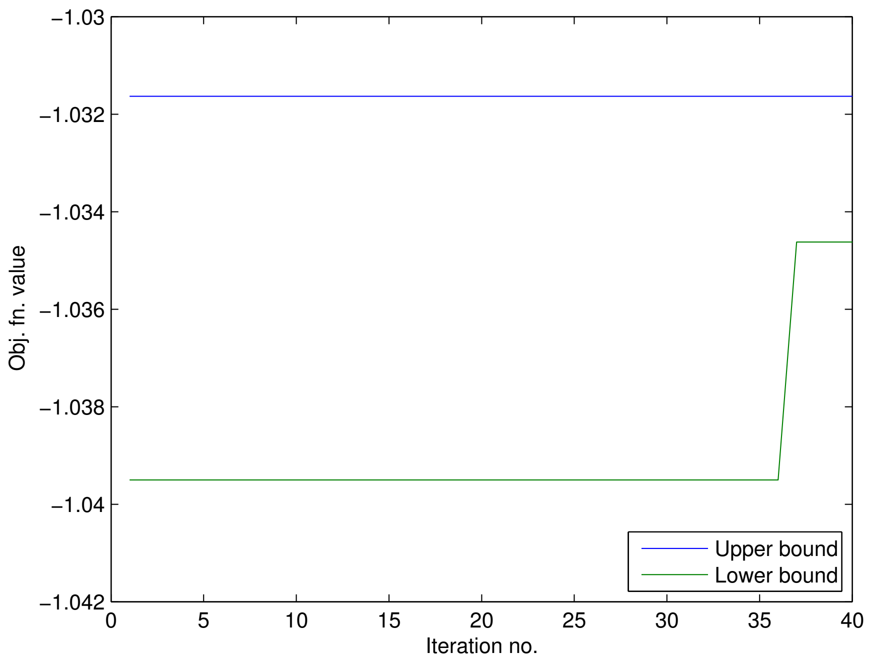


Figure 19: Branch-and-Bound algorithm progress with the B-spline approach.

Since we are using an approximation, the optimal solution will have an error. However, this error becomes smaller when we increase the number of samples.

References

[1] C.S. Adjiman, S. Dallwig, C.A. Floudas, and A. Neumaier. A global optimization method, Îśbb, for general twice-differentiable constrained {NLPs} âĂŤ i. theoretical advances. *Computers & Chemical Engineering*, 22(9):1137 – 1158, 1998.

[2] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, New York, NY, USA, 2004.

[3] Jens Clausen. Branch and bound algorithms-principles and examples. *Department of Computer Science, University of Copenhagen*, pages 1–30, 1999.

[4] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[5] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer, 1999.

[6] Andreas WÃďchter and Lorenz T. Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical Programming*, 106(1):25–57, 2006.