# HiGHS Version 1.0.0

highsopt@gmail.com

June 9, 2021

## Contents

# 1 Introduction

This document defines what HiGHS can do, and how it can be run and controlled externally. It does not describe the theory underlying the optimzation methods that it uses, how they implemented, or how HiGHS is engineered internally. Mathematically, this document assumes no more than familiarity with the definition of a linear programming (LP) problem, and the fact that they can be solved with either the simplex algorithm or interior point methods. Although HiGHS can solve mixed-integer programming (MIP) and quadratic programming (QP) problems, users who have no interest in doing so can ingore references to these problem classes.
HiGHS was developed from the dual simplex solver of Huangfu [3] by adding the interior point method solver (for LP problems) of Schork [4], the MIP solver of Gottwald, and the convex QP solver of Feldmeier. Its presolve for LP and MIP was written by Galabova and Gottwald, and its primal simplex solver was written by Hall, based on work by Huangfu. Other features have been written by Hall.
HiGHS is mainly written in C++ with OpenMP directives, but also has some C. The primary interface to HiGHS is via C++, but it also has full interfaces for C, C#, FORTRAN 90 and Python. There are also third-party interfaces to HiGHS from Julia, Rust and Javascript. This document describes the C++ interface and corresponding C interface methods. The other interfaces are analogous to the C interface.
HiGHS uses CMake as a build system and requires version 3.15. It has been developed and tested on various Linux, MacOS and Windows installations using both the GNU (g++) and Intel (icc) C++ compilers. Note that HiGHS requires (at least) version 4.9 of the GNU compiler. If OpenMP is unavailable to HiGHS, then it will compile, but will run in serial. HiGHS has no third-party dependencies.
Although HiGHS is freely available under the MIT license, we would be pleased to learn about users' experience and give advice via email sent to highsopt@gmail.com. If you use HiGHS in an academic context, please acknowledge this and cite the article of Huangfu and Hall [3], unless you make specific use of the interior point method solver, in which case you should cite the work of Schork and Gondzio [4].

## 1.1 Overview

Sections 2- 7 set out what HiGHS can do, the methods by which this is achieved, and the parameters that control them. This is done by introducing the data structures of HiGHS that are used in the C++ interface, the methods in the C++ interface, and the corresponding methods in the C interface.
Models are either passed to HiGHS directly, or read from data files. The definition of the HiGHS model data structure is set out in Section 2, together with the methods for model input and output in the C++ and C interfaces. Section 3 describes how HiGHS is used to solve models, and how the resulting solution data may be obtained. Models may be modified in ways set out in Section 5, and other functionality available within HiGHS is covered in Section 6.
The parameters that control how HiGHS runs are referred to option values. Their definition, default values and the means of modifying them are defined in Section 7.

## 1.2 Creating a Highs instance

In C++, an instance of the Highs class named (for example) highs is created thus

```
Highs highs;
```

This requires the inclusion of the header file Highs.h.
In C, an instance of Highs is created and destroyed using the methods defined in Table 1. So an instance

```
void* Highs_create()
void Highs_destroy(void* highs)
```

Table 1: Creating and destroying a Highs instance in C using the methods Highs_create() and Highs_destroy(void* highs)

named (for example) highs is created and destroyed thus

```
void* highs = Highs_create();

Highs_destroy(highs);
```

For all other methods in the C interface, the first argument is the void pointer highs.

## 1.3 Long integers

By default, HiGHS compiles with standard (32-bit) integers. However, to allow larger models to be handled (with the exception of the interior point method solver) it is now possible to compile HiGHS so that it uses long (64-bit) integers in all cases. This is achieved by setting the CMake parameter -DHIGHSINT64=on. The type HighsInt is then set to be int64_t rather than int. Long integers are not currently supported by the FORTRAN interface. Since HiGHS can be compiled with either standard or long integers, integer values are referred to below anonymously as integer.

## 1.4 Return values

Most methods in the C++ and C interfaces return a status to indicate the success of the call. Those that do not are methods where model or solution information is returned (and success can be assumed). The type of the return status is enum class HighsStatus, with members listed in Table 2, where the integer value given is the cast used in the C interface.

| Name | Integer | Description |
|---|---|---|
| HighsStatus::kError | -1 | The method failed |
| HighsStatus::kOk | 0 | The method ran successfully |
| HighsStatus::kWarning | 1 | The method encountered an unexpected or uncommon scenario, but ran successfully |

Table 2: The enum class type names for HighsStatus

## 1.5 Terminology

The word "vector" is used to refer to the mathematical object in the definition of a model. Within code, the corresponding data structure is referred to as an array. The model within HiGHS that has been passed, read or built, and possibly modified, is referred to as the "incumbent model".

## 1.6 Style

For the C++ interface and, to an increasing extent internally, HiGHS now conforms to the Google C++ style guide [1].

# 2 Model definition

Models in HiGHS are defined as an instance of the HighsModel class. This consists of one instance of the HighsLp class, and one instance of the HighsHessian class. Communication of models to and from HiGHS is possible via instances of the HighsLp or HighsModel class. In the C and other interfaces, communication of models is via scalar values and addresses of arrays.

## 2.1 The `HighsLp` class

The `HighsLp` class allows LP and MIP problems of the form

$$\begin{array}{rl} \text{minimize/maximize} & f + \boldsymbol{c}^T\boldsymbol{x} \\ \text{subject to} & \boldsymbol{L} \le A\boldsymbol{x} \le \boldsymbol{U} \\ & \boldsymbol{l} \le \boldsymbol{x} \le \boldsymbol{u} \\ & \{x_i : i \in \mathcal{I}\} \in \mathbb{Z} \end{array}$$

to be defined, and its members are defined in Table 3. Since `HiGHS` is written in C++ and C, all indexing begins from zero.

| Type | Name | Description |
|---|---|---|
| `integer` | `num_col_` | Number of columns (variables) $\boldsymbol{x}$ in the model |
| `integer` | `num_row_` | Number of rows (constraints) in the model |
| `std::vector<integer>` | `a_start_` | Start of each vector of $A$ in the compressed sparse storage |
| `std::vector<integer>` | `a_index_` | Indices of each vector of $A$ in the compressed sparse storage |
| `std::vector<double>` | `a_value_` | Values of each vector of $A$ in the compressed sparse storage |
| `std::vector<double>` | `col_cost_` | Cost (gradient) vector $\boldsymbol{c}$ in the objective function |
| `std::vector<double>` | `col_lower_` | Lower bounds $\boldsymbol{l}$ on the columns |
| `std::vector<double>` | `col_upper_` | Upper bounds $\boldsymbol{u}$ on the columns |
| `std::vector<double>` | `row_lower_` | Lower bounds $\boldsymbol{L}$ on the rows |
| `std::vector<double>` | `row_upper_` | Upper bounds $\boldsymbol{U}$ on the rows |
| `MatrixOrientation` | `orientation_` | Orientation of the constraint matrix |
| `ObjSense` | `sense_` | Sense of the objective: minimize or maximize |
| `double` | `offset_` | Constant term $f$ in the objective function |
| `std::string` | `model_name_` | The name of the model |
| `std::vector<std::string>` | `col_name_` | The names of the columns |
| `std::vector<std::string>` | `row_name_` | The names of the rows |
| `std::vector<HighsVarType>` | `integrality_` | The set $\mathcal{I}$ of integrality restrictions on columns |

Table 3: `HighsLp` class members

Three members of the `HighsLp` class (`orientation_`, `sense_` and `integrality_`) are of types particular to `HiGHS`. They are all `enum class`, with members listed in Table 4, where the integer value given is the cast used in the C interface.

| Name | Integer | Description |
|---|---:|---|
| `MatrixOrientation::kNone` | 0 | Undefined matrix orientation |
| `MatrixOrientation::kColwise` | 1 | Column-wise matrix orientation |
| `MatrixOrientation::kRowwise` | 2 | Row-wise matrix orientation |
| `ObjSense::kMinimize` | 1 | Optimization sense is minimize |
| `ObjSense::kMaximize` | -1 | Optimization sense is maximize |
| `HighsVarType::kContinuous` | 0 | Variable can take continuous values |
| `HighsVarType::kInteger` | 1 | Variable must take integer values |

Table 4: The `enum class` type names for `HighsLp` class members

### 2.1.1 Constraint matrix

The constraint matrix $A$ is held using compressed sparse storage. Users unfamiliar with this format should consult Wikipedia [2]. Since solvers generally access the constraint matrix column-wise, this was originally the only orientation permitted by HiGHS. However, it is often preferable for modelling interfaces and users to specify the constraint matrix row-wise. The orientation_ member of the HighsLp class is used to specify whether the matrix is held row-wise or column-wise. When a HighsLp instance is created, the orientation is indicated as undefined by being set to MatrixOrientation::kNone. If the orientation is not set to either MatrixOrientation::kColwise or MatrixOrientation::kRowwise, HiGHS returns an error. In the C interface, an integer parameter indicates whether the matrix is row-wise or column-wise according to the integer values in Table 4.
If the constraint matrix $A$ is held column-wise (row-wise) then the size of a_start_ must be at least num_col_+1 (num_row_+1), the zero'th entry of a_start_ must be zero, and entry num_col_ (num_row_) is the number of entries in the arrays a_index_ and a_value_. Other requirements are discussed in Section 2.3.1.

### 2.1.2 Objective function

The sense of the objective (minimize or maximize) is defined by the value of the sense_ member of the HighsLp class. This is of type ObjSense, an enum class whose names are given in Table 4. When a HighsLp instance is created, sense_ is set to ObjSense::kMinimize so, unless changed, the objective is minimized. In the C interface, an integer parameter indicates whether the matrix is to be minimized or maximized according to the integer values in Table 4.

### 2.1.3 Names

When populating an instance of the HighsLp class, no name data needs to be supplied, and name data cannot be supplied via the C interface. The model_name_ member of the HighsLp class is the name of the model, and is used occasionally in logging output. By default it is an empty string. If the model is read from a data file, then model_name_ is the name of the file (less its extension). The col_name_ and row_name_ members of the HighsLp class are, by default, of zero size. If the model is read from an MPS data file, then col_name_ and row_name_ contain its column and row names. If names are available they are printed when HiGHS writes the model or solution.

### 2.1.4 Integrality

The integrality_ member of the HighsLp class indicates which variables (if any) must take integer values. If this member is of zero size, or if all its entries are HighsVarType::kContinuous, the set $\mathcal{I}$ is interpreted as being empty, so the instance is interpreted as being an LP. If entries of integrality_ are HighsVarType::kInteger then, by default, the HiGHS MIP solver will be used to find the optimal integer values of the corresponding variables. In the C interface, an integer indicates whether a particular variable is continuous or must take an integer value according to the integer values in Table 4.

## 2.2 The HighsHessian class

The HighsHessian class allows the HighsModel class to represent the Hessian matrix of the quadratic term in the objective function that distinguishes QP problems from LP problems. The representation of the Hessian matrix is column-wise, and the HighsHessian class members are defined in Table 5.
If a HighsHessian instance has a positive value for dim_, and the number of nonzeros in $Q$ is positive, then dim_ must be equal to num_col_. As with the column-wise constraint matrix, the size of q_start_ must be at least dim_+1, the zero'th entry of q_start_ must be zero, and entry dim_ is the number of entries in the arrays q_index_ and q_value_. Other requirements are discussed in Section 2.3.4.

| Type | Name | Description |
|---|---|---|
| integer | dim_ | Number of columns in the Hessian matrix |
| std::vector<integer> | q_start_ | Start of each column in the compressed sparse column storage |
| std::vector<integer> | q_index_ | Indices of each column in the compressed sparse column storage |
| std::vector<double> | q_value_ | Values of each column in the compressed sparse column storage |

Table 5: HighsHessian class members

## 2.3 Model validation

When HiGHS obtains a new model, it is validated for correctness and extreme data values. Control of the latter is determined by option values.

### 2.3.1 Constraint matrix validation

There is considerable scope for errors when defining a matrix in compressed vector form. Thus, HiGHS validates any such data supplied to it. The zero'th entry of the a_start_ array must be zero, and entries must increase monotonically. This will not be strict if the matrix contains zero vectors. The a_index_ entries corresponding to a particular vector must be non-negative, must not be greater than or equal to the (full) vector dimension, and there must be no duplicate entries. If the data fails to satisfy any of these conditions, HiGHS will return an error.
There are limits on the minimum and maximum absolute values in the constraint matrix that HiGHS will use. These limits are HighsOptions values small_matrix_value and large_matrix_value. Any entries below small_matrix_value in absolute value are ignored (with a warning) but entries found to be greater than large_matrix_value in absolute value will cause HiGHS to return with an error.

### 2.3.2 Cost validation

In C++, if the size of col_cost_ is less than num_col_ then HiGHS will return an error. Any entries found to be greater than infinite_cost in absolute value will cause HiGHS to return with a warning or an error.

### 2.3.3 Bound validation

In C++, if the size of col_lower_ or col_upper_ is less than num_col_, or if the size of row_lower_ or row_upper_ is less than num_row_, then HiGHS will return an error. Any lower bound found to be greater than or equal to infinite_bound, or any upper bound found to be less than or equal to -infinite_bound, will cause HiGHS to return with an error.
Bounds that are inconsistent due to the lower bound exceeding the upper bound are permitted in HiGHS but, when found, a warning message will be issued. If HiGHS is asked to solve a model with inconsistent bounds, it will identify it as being infeasible.
Any lower bounds found to be less than or equal to -infinite_bound will be ignored (treated as if they were negative infinity). Any upper bounds found to be greater than or equal to infinite_bound will be ignored (treated as if they were positive infinity). HiGHS will report the number of bounds ignored in this way.

### 2.3.4 Hessian matrix validation

An instance of the HighsHessian class is validated for illegal indexing in the same way as the constraint matrix. To discuss the interpretation of the values in the instance, let $G$ be the corresponding matrix. HiGHS determines the matrix $Q = \frac{1}{2}(G + G^T)$ and assesses its values the same way as the constraint matrix, using the same option values for small and large entries. If $Q$ is identically zero, then the HighsHessian instance is ignored, so the model will be solved as an LP. Otherwise, $Q$ will be used as the quadratic term in the objective function.

As well as guaranteeing that the QP solver operates with a symmetric Hessian matrix, this allows flexibility in the definition of the Hessian matrix. If the original `HighsHessian` instance represents an entire symmetric matrix then, clearly, $Q = G$. However, it may be convenient to for $G$ to be only the strictly upper (equivalently lower) triangular portion of the Hessian, in which case its strictly off-diagonal entries must be double the value in the entire symmetric matrix.
The `HiGHS` QP solver is for strictly convex problems: those with positive definite Hessian. It will return with an error if it identifies non-convexity. Since an *a priori* test of strict convexity is a significant overhead, the only convexity test that is performed is a check that all the diagonal entries of $Q$ are at least `small_matrix_value`.

## 2.4 Methods

In C++, the model is passed to `HiGHS` by creating an instance of the `HighsModel` class (or using a `HighsLp` instance if the model of interest is only an LP or MIP), populating its data, and passing it to `HiGHS` using `passModel` as defined in Table 6. The forms of the argument to `passModel` are defined so that maximum

```
  HighsStatus Highs::passModel(HighsModel model)
  HighsStatus Highs::passModel(HighsLp lp)
```

Table 6: Passing a `HighsModel` or `HighsLp` to `Highs` in C++

memory efficiency can be achieved by passing the `HighsModel` or `HighsLp` instance via `std::move`. For example, suppose that the user's `HighsModel` instance is called `model`. Calling `passModel(std::move(model))` ensures that the array content of the user's `HighsModel` instance is moved to the `HighsModel` instance of the incumbent model in `Highs` without creating an additional copy at any stage. A consequence of this approach is that, on return from `passModel`, the user's `HighsModel` instance has no array content.
In the C interface, the components of a `HighsModel` or `HighsLp` instance are passed as indivdual scalars or pointers. The methods for an LP, MIP or general model are defined in Table 7, where the parameter names are arbitrary. However, for comparison, the names of the corresponding class members are used.

# 3 Model solution

When the incumbent model is solved, the solution data that is available from `HiGHS` comes from four sources: the model status, a set of scalar information values, the solution of the model and the status of each variable and constraint.

## 3.1 The model status

When the incumbent model is solved, `HiGHS` will identify a model status. In C++ this value is taken from an `enum class` `HighsModelStatus`, whose entries are listed in Table 8. The corresponding integer values used in the C interface are also listed.

## 3.2 Scalar information

Scalar information about the solution of the model is held by `HiGHS` in an instance of the `HighsInfo` class. The members of this class are listed in Table 9. The truth of `valid` indicates whether the values of the remaining members are valid. In the C++ interface, a `const` reference to this instance of the `HighsInfo` class is available. Values of individual members can also be obtained, and this is how they are in the C interface. The values of `primal_solution_status` and `dual_solution_status` indicate the nature of the solution values that are available. Valid values for the primal or dual infeasiblility data are naturally non-negative. Thus a negative value indicates that the data are not available.

```
integer Highs_passLp(  void* highs,
                       const integer num_col_, const integer num_row_,
                       const integer num_nz_,
                       const integer orientation_, const integer sense_,
                       const double offset_,
                       const double* col_cost_, const double* col_lower_,
                       const double* col_upper_, const double* row_lower_,
                       const double* row_upper_, const integer* a_start_,
                       const integer* a_index_, const double* a_value_, );
integer Highs_passMip(  void* highs,
                        const integer num_col_, const integer num_row_,
                        const integer num_nz_,
                        const integer orientation_, const integer sense_,
                        const double offset_,
                        const double* col_cost_, const double* col_lower_,
                        const double* col_upper_, const double* row_lower_,
                        const double* row_upper_, const integer* a_start_,
                        const integer* a_index_, const double* a_value_,
                        const integer* integrality_ );
integer Highs_passModel(  void* highs,
                          const integer num_col_, const integer num_row_,
                          const integer num_nz_, const integer hessian_num_nz_,
                          const integer orientation_, const integer sense_,
                          const double offset_,
                          const double* col_cost_, const double* col_lower_,
                          const double* col_upper_, const double* row_lower_,
                          const double* row_upper_, const integer* a_start_,
                          const integer* a_index_, const double* a_value_,
                          const integer* q_start_, const integer* q_index_,
                          const double* q_value_,
                          const integer* integrality_ );
```

Table 7: Passing data for a HighsModel or HighsLp to Highs in C

Methods in ihe C++ interface allows values of individual members to be obtained, and it is via correponding methods in the C interface that these information values are obtained.

## 3.3 Solution values

Solution values for the model are held by HiGHS in an instance of the HighsSolution structure. Its components are listed in Table 10 In the C++ interface, a const reference to this instance of the HighsSolution structure is available, and a copy of the same data can be obtained via the C interface. When the objective sense is switched, the dual optimality condition changes sign, and this is reflected in the sign of the dual values.

The solution data that are available from HiGHS depends on the value of the model status, when it was identified, and how. To identify what solution data are available, and whether the solution values are feasible or not, consult the values of primal_solution_status and dual_solution_status.

| Integer | HighsModelStatus | Description |
|---|---|---|
| 0 | `kNotset` | Not set |
| 1 | `kLoadError` | Error loading the model |
| 2 | `kModelError` | Error in the model definition |
| 3 | `kPresolveError` | Error presolving the model |
| 4 | `kSolveError` | Error solving the model |
| 5 | `kPostsolveError` | Error postsolving the model |
| 6 | `kModelEmpty` | Model is empty |
| 7 | `kOptimal` | Model solution is optimal |
| 8 | `kInfeasible` | Model is (primal) infeasible |
| 9 | `kUnboundedOrInfeasible` | Model is (primal) unbounded or infeasible |
| 10 | `kUnbounded` | Model is (primal) unbounded |
| 11 | `kObjectiveBound` | Any optimal objective is at least a given bound |
| 12 | `kObjectiveTarget` | There is a feasible solution with at least a given target value |
| 13 | `kTimeLimit` | The solution time limit has been reached |
| 14 | `kIterationLimit` | The solution iteration limit has been reached |
| 15 | `kUnknown` | The model status is unknown |

Table 8: Model status values

## 3.4 The variable and constraint status

The status of each variable and constraint is held by `HiGHS` in an instance of the `HighsBasis` structure. In the case of LP problems, this is the *basis*. Its main components are listed in Table 11 The `HighsBasisStatus` type is an `enum class`, with members listed in Table 12, where the integer value given is the cast used in the C interface. In the C++ interface, a `const` reference to this instance of the `HighsBasis` structure is available, and a copy of the same data can be obtained via the C interface.

The variable and constraint status data that are available from `HiGHS` depends on the value of the model status, when it was identified, and how. To identify what variable and constraint status data are available, consult the value of `basis_status`.

## 3.5 Methods

The incumbent model is solved by calling the C++ method

- `HighsStatus Highs::run()`

The corresponding method in the C interface is

- `integer highs_run(void* highs)`

By default, `Highs` will use what it considers to be the best solver of the appropriate type for the incumbent model, and the best run-time option settings. To force `Highs` to solve with a particular solver and run-time options, `HighsOptions` values in Section 7 may be modified.

| Type | Name | Description |
|---|---|---|
| bool | valid | Validity of the data in the class instance |
| int64_t | mip_node_count | Number of MIP nodes created |
| integer | simplex_iteration_count | Cumulative number of simplex iterations performed |
| integer | ipm_iteration_count | Cumulative number of interior point iterations performed |
| integer | crossover_iteration_count | Cumulative number of crossover iterations performed |
| integer | qp_iteration_count | Cumulative number of QP solver iterations performed |
| integer | primal_solution_status | Status of the primal solution |
| integer | dual_solution_status | Status of the dual solution |
| integer | basis_status | Status of the variable and constraint status information |
| double | objective_function_value | Objective function value |
| double | mip_dual_bound | Dual bound from the MIP solover |
| double | mip_gap | Relative primal-dual gap from the MIP solover |
| integer | num_primal_infeasibilities | Number of primal infeasibilities exceeding the tolerance |
| double | max_primal_infeasibility | Maximum primal infeasibility |
| double | sum_primal_infeasibilities | Sum of primal infeasibilities |
| integer | num_dual_infeasibilities | Number of dual infeasibilities exceeding the tolerance |
| double | max_dual_infeasibility | Maximum dual infeasibility |
| double | sum_dual_infeasibilities | Sum of dual infeasibilities |

Table 9: Members of the `HighsInfo` class

| Type | Name | Description |
|---|---|---|
| std::vector<double> | col_value | (Primal) solution values for columns (variables) |
| std::vector<double> | row_value | (Primal) solution values for rows (constraints) |
| std::vector<double> | col_dual | Dual values for columns |
| std::vector<double> | row_dual | Dual values for rows |

Table 10: Members of the `HighsSolution` structure

## 4 Model extraction

## 5 Model modification

## 6 Other features

## 7 Options

- `double infinite_cost`
- `double infinite_bound`
- `double small_matrix_value`
- `double large_matrix_value`
- `double primal_feasibility_tolerance`
- `double dual_feasibility_tolerance`
- `double ipm_optimality_tolerance`
- `double objective_bound`

| Type | Name | Description |
|---|---|---|
| `std::vector<HighsBasisStatus>` | `col_status` | Status of the columns (variables) |
| `std::vector<HighsBasisStatus>` | `row_status` | Status of the rows (constraints) |

Table 11: Members of the `HighsBasis` structure

| Name | Integer | Description |
|---|---|---|
| `HighsBasisStatus::kLower` | 0 | At lower bound or fixed value |
| `HighsBasisStatus::kBasic` | 1 | Solved for (basic) |
| `HighsBasisStatus::kUpper` | 2 | At upper bound |
| `HighsBasisStatus::kZero` | 3 | Free and set to zero |

Table 12: The `enum class` type names for `HighsBasisStatus`

- `double objective_target`
- `integer highs_random_seed`
- `integer highs_debug_level`
- `integer highs_analysis_level`
- `integer simplex_strategy`
- `integer simplex_scale_strategy`
- `integer simplex_crash_strategy`
- `integer simplex_dual_edge_weight_strategy`
- `integer simplex_primal_edge_weight_strategy`
- `integer simplex_iteration_limit`
- `integer simplex_update_limit`
- `integer ipm_iteration_limit`
- `integer highs_min_threads`
- `integer highs_max_threads`
- `std::string solution_file`
- `std::string log_file`
- `bool write_solution_to_file`
- `bool write_solution_pretty`
- `bool output_flag`
- `bool log_to_console`
- `integer log_dev_level`
- `bool run_crossover`

- bool allow_unbounded_or_infeasible
- bool use_implied_bounds_from_presolve
- bool mps_parser_type_free
- integer keep_n_rows
- integer allowed_simplex_matrix_scale_factor
- integer allowed_simplex_cost_scale_factor
- integer simplex_dualise_strategy
- integer simplex_permute_strategy
- bool dual_simplex_cleanup
- integer simplex_price_strategy
- integer presolve_substitution_maxfillin
- bool simplex_initial_condition_check
- double simplex_initial_condition_tolerance
- double dual_steepest_edge_weight_log_error_threshold
- double dual_simplex_cost_perturbation_multiplier
- double primal_simplex_bound_perturbation_multiplier
- double presolve_pivot_threshold
- double factor_pivot_threshold
- double factor_pivot_tolerance
- double start_crossover_tolerance
- bool less_infeasible_DSE_check
- bool less_infeasible_DSE_choose_row
- bool use_original_HFactor_logic
- integer mip_max_nodes
- integer mip_max_stall_nodes
- integer mip_max_leaves
- integer mip_lp_age_limit
- integer mip_pool_age_limit
- integer mip_pool_soft_limit
- integer mip_pscost_minreliable
- integer mip_report_level

- double `mip_feasibility_tolerance`
- double `mip_epsilon`
- double `mip_heuristic_effort`

## References

[1] Google C++ Style Guide. `https://google.github.io/`, 2021. Accessed: 17/05/2021.

[2] Sparse matrix. `https://en.wikipedia.org/wiki/Sparse_matrix`, 2021. Accessed: 17/05/2021.

[3] Q. Huangfu and J. A. J. Hall. Parallelizing the dual revised simplex method. *Mathematical Programming Computation*, 10(1):119–142, 2018.

[4] L. Schork and J. Gondzio. Implementation of an interior point method with basis preconditioning. *Mathematical Programming Computation*, 12(4):603–635, 2020.