

MOOCHO

Version of the Day

Generated by Doxygen 1.7.6.1

Wed Mar 18 2015 23:00:02

## Contents

<b>1</b>	<b>MOOCHO: Multi-functional Object-Oriented arCHitecture for Optimization</b>	<b>1</b>
1.1	Outline . . . . .	1
1.2	Introduction . . . . .	2
1.3	MOOCHO Mathematical Overview Document . . . . .	3
1.4	Hyper-linked HTML version of this Document . . . . .	3
1.5	MOOCHO Quickstart . . . . .	4
1.5.1	Setting up a driver program to call a MOOCHO solver . . . . .	4
1.5.2	Running MOOCHO to Solve Optimization Problems . . . . .	4
1.6	Representing Nonlinear Programs for MOOCHO to Solve . . . . .	14
1.6.1	Representing General Serial NLPs with Explicit Jacobian Entries . . . . .	14
1.6.2	Representing Simulation-Constrained Parallel NLPs through Thyra . . . . .	16
1.7	Other Trilinos Packages on which MOOCHO Directly Depends . . . . .	17
1.8	Individual MOOCHO Doxygen Collections . . . . .	18
1.9	Browse all of MOOCHO as a Single Doxygen Collection . . . . .	19
1.10	Links to Other Documentation Collections . . . . .	19
<b>2</b>	<b>Module Index</b>	<b>20</b>
2.1	Modules . . . . .	20
<b>3</b>	<b>Module Documentation</b>	<b>20</b>
3.1	Sample MOOCHO input and output. . . . .	21
3.2	Sample MOOCHO Options File . . . . .	23
3.3	Sample MOOCHO Console Output . . . . .	25
3.4	Sample MOOCHO Algorithm Configuration Output (MoochoAlgo.out) . . . . .	26
3.5	Sample MOOCHO Algorithm Summary Output (MoochoSummary.out) . . . . .	27
3.6	Sample MOOCHO Algorithm Journal Output (MoochoJournal.out) . . . . .	28
<b>4</b>	<b>Example Documentation</b>	<b>29</b>
4.1	ExampleNLPBandedMain.cpp . . . . .	29
4.2	NLPThyraEpetraAdvDiffReactOptMain.cpp . . . . .	29

<a href="#">4.3 NLPThyraEpetraModelEval4DOptMain.cpp</a>	29
<a href="#">4.4 NLPWBCounterExampleMain.cpp</a>	29

## **1 MOOCHO: Multi-functional Object-Oriented arCHitecture for Optimization**

### **1.1 Outline**

- [Introduction](#)
- [MOOCHO Mathematical Overview Document](#)
- [Hyper-linked HTML version of this Document](#)
- [MOOCHO Quickstart](#)
  - [Setting up a driver program to call a MOOCHO solver](#)
  - [Running MOOCHO to Solve Optimization Problems](#)
    - \* [Linear solver input parameters for Stratimikos \(Thyra models only\)](#)
    - \* [MOOCHO input options](#)
    - \* [MOOCHO algorithm output](#)
      - [Console output \(output\)](#)
      - [Algorithm Configuration Output \(MoochoAlgo.out\)](#)
      - [Algorithm Summary and Timing Output \(MoochoSummary.out\)](#)
      - [Algorithm Journal Output \(MoochoJournal.out\)](#)
    - \* [Algorithm Interruption](#)
- [Representing Nonlinear Programs for MOOCHO to Solve](#)
  - [Representing General Serial NLPs with Explicit Jacobian Entries](#)
  - [Representing Simulation-Constrained Parallel NLPs through Thyra](#)
- [Other Trilinos Packages on which MOOCHO Directly Depends](#)
- [Individual MOOCHO Doxygen Collections](#)
- [Browse all of MOOCHO as a Single Doxygen Collection](#)
- [Links to Other Documentation Collections](#)

## 1.2 Introduction

MOOCHO (Multifunctional Object-Oriented arCHitecture for Optimization) is a **Trilinos** package written in C++ designed to solve large-scale, equality and inequality nonlinearly constrained, non-convex optimization problems (i.e. nonlinear programs) using reduced-space successive quadratic programming (SQP) methods. The most general form of the optimization problem that can be solved is:

$$\begin{aligned} &\text{minimize} && f(x) \\ &\text{subject to} && c(x) = 0 \\ &&& x_L \leq x \leq x_U \end{aligned}$$

where  $x \in \mathbb{R}^n$  the vector of optimization variables,  $f(x) \in \mathbb{R}^n \rightarrow \mathbb{R}$  is the nonlinear scalar objective function,  $c(x) = 0$  (where  $c(x) \in \mathbb{R}^n \rightarrow \mathbb{R}^m$ ) are the nonlinear constraints, and  $x_L$  and  $x_U$  are the upper and lower bounds on the variables. The current algorithms in MOOCHO are well suited to solving optimization problems with massive numbers of unknown variables and equations but few so-called degrees of optimization freedom (i.e. the degrees of freedom = the number of variables minus the number of equality constraints =  $n - m$ ). Various line-search based globalization methods are available, including exact penalty functions and a form of the filter method. Many of the algorithms in MOOCHO are provably locally and globally convergent for a wide class of problems in theory but in practice the behavior and the performance of the algorithms varies greatly from problem to problem.

MOOCHO was initially developed to solve general sparse optimization problems where there is no clear distinction between state variables and optimization parameters. For these types of problems a serial sparse direct solver must be used (i.e. MA28) to find a square basis that is needed for the variable reduction decompositions that are current supported.

More recently, MOOCHO has been interfaced through **Thyra** and the `Thyra::ModelEvaluator` interface to address very large-scale, massively parallel, simulation-constrained optimization problems that take the form:

$$\begin{aligned} &\text{minimize} && f(x_D, x_I) \\ &\text{subject to} && c(x_D, x_I) = 0 \\ &&& x_{D,L} \leq x_D \leq x_{D,U} \\ &&& x_{I,L} \leq x_I \leq x_{I,U} \end{aligned}$$

where  $x_D \in \mathbb{R}^m$  are the "dependent" state variables,  $x_I \in \mathbb{R}^{n-m}$  are the "independent" optimization parameters and  $c(x_D, x_I) = 0$  are the discrete nonlinear state simulation equations. Here the state Jacobian  $\frac{\partial c}{\partial x_D}$  must be square and nonsingular and the partitioning of  $x = \begin{bmatrix} x_D^T & x_I^T \end{bmatrix}^T$  into state variables  $x_D$  and optimization variables  $x_I$  must be known *a priori* and this partitioning can not change during a solve. Warning, the `Thyra::ModelEvaluator` interface uses a overlapping and inconsistent set of names for the variables and the problem functions than the names used by

MOOCHO. All of the functionality needed for MOOCHO to solve a simulation-constrained optimization problem can be specified through sub-classing the `Thyra::ModelEvaluator` interface, and related `Thyra` interfaces. Epetra-based applications can instead implement the `EpetraExt::ModelEvaluator` interface and never need to work with `Thyra` directly except in trivial and transparent ways.

For simulation-constrained optimization problems, MOOCHO can utilize the full power of the massively parallel iterative linear solvers and preconditioners available in Trilinos through `Thyra` through the `Stratimikos` package by just flipping a few switches in a parameter list. These include all of the direct solves in `Amesos`, the preconditioners in `Ipack` and `ML`, and the iterative Krylov solvers in `AztecOO` and `Belos` (`Belos` is not being released but is available in the development version of Trilinos). For small to moderate numbers of optimization parameters, the only bottleneck to parallel scalability is the linear solver used to solve linear systems involving the state Jacobian  $\frac{\partial c}{\partial x_D}$ . The reduced-space SQP algorithms in MOOCHO itself achieve extremely good parallel scalability. The parallel scalability of the linear solvers is controlled by the simulation application and the Trilinos linear solvers and preconditioners themselves. Typically, the parallel scalability of the linear solve is limited by the preconditioner as the problem is partitioned to more and more processes.

MOOCHO also includes a minimally invasive mode for reduced-space SQP where the simulator application only needs to compute the objective and constraint functions  $f(x_D, x_I) \in \mathbb{R}^n \rightarrow \mathbb{R}$  and  $c(x_D, x_I) \in \mathbb{R}^n \rightarrow \mathbb{R}^m$  and solve only forward linear systems involving  $\frac{\partial c}{\partial x_D}$ . All other derivatives can be approximated with directional finite differences but any exact derivatives that can be computed by the application are happily accepted and fully utilized by MOOCHO through the `Thyra::ModelEvaluator` interface.

### 1.3 MOOCHO Mathematical Overview Document

A more detailed mathematical overview of nonlinear programming and the algorithms that MOOCHO implements are described in the document [A Mathematical and High-Level Overview of MOOCHO](#). This document also defines the mapping of mathematical notation to C++ identifiers used by MOOCHO. User's should at least browse this document in order to understand the basics of what MOOCHO is doing.

### 1.4 Hyper-linked HTML version of this Document

The doxygen-generated hyper-linked version of this document can be found at the Trilinos website under the link to MOOCHO.

## 1.5 MOOCHO Quickstart

In order to get started using MOOCHO to solve your NLPs you must first build MOOCHO as part of Trilinos and install it.

### Quickstart Outline

- [Setting up a driver program to call a MOOCHO solver](#)
- [Running MOOCHO to Solve Optimization Problems](#)
  - [Linear solver input parameters for Stratimikos \(Thyra models only\)](#)
  - [MOOCHO input options](#)
  - [MOOCHO algorithm output](#)
    - \* [Console output \(output\)](#)
    - \* [Algorithm Configuration Output \(MoochoAlgo.out\)](#)
    - \* [Algorithm Summary and Timing Output \(MoochoSummary.out\)](#)
    - \* [Algorithm Journal Output \(MoochoJournal.out\)](#)
  - [Algorithm Interruption](#)

#### 1.5.1 Setting up a driver program to call a MOOCHO solver

Once an NLP is defined, a driver program must be constructed to setup a MOOCHO solver and configure it given options set by the user. When building a driver program to solve an NLP based on an

`NLPInterfacePack::NLPSerialPreprocessExplJac` subclass object, one should directly use the "Facade" solver class `MoochoPack::MoochoSolver` (see [NLPWBCounterExampleMain.cpp](#)). However, when using an NLP based on a `Thyra::ModelEvaluator` object, then the more specialized "Facade" solver class `MoochoPack::MoochoThyraSolver` should be used (see [NLPThyraEpetraModelEval4DOptMain.cpp](#)). The class `MoochoPack::MoochoThyraSolver` just uses `MoochoPack::MoochoSolver` internally for the main solve but provides a great deal of extra functionality to set initial guesses (also from an input file) and wrap the model evaluator object with various "Decorator" skins and to capture and return the final solution.

#### 1.5.2 Running MOOCHO to Solve Optimization Problems

Once an NLP is defined and a driver program is in place (see the above driver programs), then MOOCHO can be run to try to solve the optimization problem. Most of the options that affect MOOCHO (and the Trilinos linear solvers accessed through Stratimikos) can be read in from various input files or specified entirely on the

command line. The driver programs shown above show examples of how to setup a `Techos::CommandLineProcessor` object to accept a number of different command-line arguments that can be used to read in MOOCHO and Trilinos linear solver options. For example, consider the simple driver program `NLPThyraEpetraModelEval4DOptMain.cpp`. This example shows the use of both MOOCHO options and Stratimikos linear solver options.

Here are the command-line arguments that the program `NLPThyraEpetraModelEval4DOptMain.cpp` accepts:

Below, the various types of input and output are described. Input parameters/options are separated into linear solver parameters for Stratimikos and algorithm options for MOOCHO.

#### 1.5.2.1 Linear solver input parameters for Stratimikos (Thyra models only)

When using a `Thyra::ModelEvaluator`-based NLP, the linear solver options for inverting the basis of the equality constraints are read in through a `Techos::ParameterList` object which is accepted through the Stratimikos class `Stratimikos::DefaultLinearSolverBuilder`. When a `MoochoPack::MoochoThyraSolver` object is used to build a driver program, it can add options to the `Techos::CommandLineProcessor` object through the function `Stratimikos::DefaultLinearSolverBuilder::setupCLP()` (see `NLPThyraEpetraModelEval4DOptMain.cpp`). This adds the command-line arguments `--linear-solver-params-file` and `--extra-linear-solver-params` which are used to read in parameters for the Stratimikos-wrapped linear solvers in XML format.

The linear solver parameters file is specified in XML and the list of all of the valid options can be found in the documentation for the class `Stratimikos::DefaultLinearSolverBuilder` itself.

An example of a linear solver options input file that specifies the use of the Amesos solver `Amesos_Klu` is shown below:

The XML input for the linear solver parameters can be read from a file using the `--linear-solver-params-file` argument and/or from the command-line itself using the `--extra-linear-solver-params` argument. Note that any parameters specified by the `--extra-linear-solver-params` argument will append and overwrite those read in from the file specified by the `--linear-solver-params-file` argument. If the argument `--linear-solver-params-file` is missing, then a set of internal options is looked for instead.

## 1.5.2.2 MOOCHO input options

The input for the MOOCHO options currently uses a completely different system than for linear solver parameters used by Stratimikos. The class

`OptionsFromStreamPack::OptionsFromStream` is used to read in MOOCHO options from a text string (or a file) and is used to represent an options data base that is used by MOOCHO. The function

`MoochoPack::MoochoSolver::setup_commandline_processor()` can be used to set the command-line arguments `--moocho-options-file` and `--moocho-extra-options` to read in MOOCHO options. The format of the options file and a listing, with documentation, of all of the valid MOOCHO options is shown here.

An example of an options file showing some of the common options that a user might want to set is shown below:

```
begin_options
```

```
options_group NLPsSolverClientInterface {
    max_iter = 20;
    max_run_time = 2.0; *** In minutes
    opt_tol = 1e-2;
    feas_tol = 1e-7;
    * journal_output_level = PRINT_NOTHING; * No output to journal from algorithm
    * journal_output_level = PRINT_BASIC_ALGORITHM_INFO; * O(1) information usually
    journal_output_level = PRINT_ALGORITHM_STEPS; * O(iter) output to journal [default]
    * journal_output_level = PRINT_ACTIVE_SET; * O(iter*nact) output to journal
    * journal_output_level = PRINT_VECTORS; * O(iter*n) output to journal (lots!)
    * journal_output_level = PRINT_ITERATION_QUANTITIES; * O(iter*n*m) output to journal (big lots!)
    * null_space_journal_output_level = DEFAULT; * Set to journal_output_level [default]
    * null_space_journal_output_level = PRINT_ACTIVE_SET; * O(iter*nact) output to journal
    * null_space_journal_output_level = PRINT_VECTORS; * O(iter*(n-m)) output to journal
    null_space_journal_output_level = PRINT_ITERATION_QUANTITIES; * O(iter*(n-m)^2) output to journal
    journal_print_digits = 10;
    calc_conditioning = true;
    calc_matrix_norms = true; *** (costly?)
    calc_matrix_info_null_space_only = true; *** (costly?)
}

options_group DecompositionSystemStateStepBuilderStd {
    * null_space_matrix = AUTO; *** Let the solver decide [default]
    null_space_matrix = EXPLICIT; *** Compute and store D = -inv(C)*N explicitly
    * null_space_matrix = IMPLICIT; *** Perform operations implicitly with C, N (requires adjoint)
    * range_space_matrix = AUTO; *** Let the algorithm decide dynamically [default]
    * range_space_matrix = COORDINATE; *** Y = [ I; 0 ] (Cheaper computationally)
    range_space_matrix = ORTHOGONAL; *** Y = [ I; -N'*inv(C') ] (more stable)
}

options_group NLPAlgoConfigMamaJama {
    * quasi_newton = AUTO; *** Let solver decide dynamically [default]
    quasi_newton = BFGS; *** Dense BFGS
    quasi_newton = LBFGS; *** Limited memory BFGS
    * line_search_method = AUTO; *** Let the solver decide dynamically [default]
    * line_search_method = NONE; *** Take full steps at every iteration
}
```



```

*   line_search_method = DIRECT;          *** Use standard Armijo backtracking
    line_search_method = FILTER;          *** [default] Use the Filter line search method
}

end_options

```

For more detailed documentation on what each of these options mean and more, see the full listing of the options [here](#).

The command-line argument `--moocho-options-file` is used to read in a set of MOOCHO options from a file using the format shown above. If the argument `--moocho-options-file` is missing, then a file with the name "Moocho.opt" is looked for in the current directory. If this file is not found, then a warning is printed and a default set of options are used. The user is warned to check that their opinions file was actually read and that it will be ignored if it is not found!

The argument `--moocho-extra-options` can be used to specify MOOCHO options directly on the command line in a slightly more terse format than the format of a MOOCHO options file. For example, the command-line equivalent to a subset of the options set in the above example MOOCHO options file is:

```

--moocho-extra-options="\
  NLP SolverClientInterface{max_iter=20,max_run_time=2.0,opt_tol=1e-2,feas_tol=1e-7\
    ,journal_output_level= PRINT_ALGORITHM_STEPS\
    ,null_space_journal_output_level=PRINT_ITERATION_QUANTITIES\
    ,journal_print_digits=10,calc_conditioning=true,calc_matrix_norms=true\
    ,calc_matrix_info_null_space_only=true}\
:DecompositionSystemStateStepBuilderStd{\
  null_space_matrix=EXPLICIT,range_space_matrix=ORTHOGONAL}\
:NLPAlgoConfigMamaJama{quasi_newton=BFGS,line_search_method=FILTER}"

```

The options specified in the `--moocho-extra-options` argument will append and override those read in from a MOOCHO input file specified by the `--moocho-options-file` argument.

### 1.5.2.3 MOOCHO algorithm output

When a MOOCHO optimization algorithm is run, by default, several different types of output are generated. By default, output is sent to the console (i.e. standard out) and to three different files: `MoochoSummary.out`, `MoochoAlgo.out`, and `MoochoJournal.out`. These four output streams provide different types of information about the MOOCHO algorithm.

To demonstrate the output files, here we show example output generated by the example program `NLPThyraEpetraModelEval4DOptMain.cpp`. This example is used since it is fairly simple but can be used to generate more interesting output files. The output from running MOOCHO on a `Thyra::ModelEvaluator`-based NLP looks very similar to running on one based on the more general NLP interface.

The example program `NLPThyraEpetraModelEval4DOpt.exe` when run with the command-line arguments:

with the above sample `Moocho.opt` options file, creates the output:

- [Console output \(output\)](#)
- [Algorithm Configuration Output \(MoochoAlgo.out\)](#)
- [Algorithm Summary and Timing Output \(MoochoSummary.out\)](#)
- [Algorithm Journal Output \(MoochoJournal.out\)](#)

Each of these different types of output are described below and the major types of output that are included in each output stream are discussed. The purpose of this treatment is to familiarize the user with the contents of these outputs and to give hints of where to look for a certain types of information.

Before going into the details of each individual type of output, first a few general comments are in order. First, at the top of every output file (except for the console output) a header is included that briefly describes the general purpose of the output file. This header is followed by an echo of the options that were read into the `OptionsFromStreamPack::OptionsFromStream` object. These options include those set in the input file `Moocho.opt` or by some other means (e.g. in the executable or on the command line) as described above. The purpose of echoing the options in each file is to help record what settings were used to produce the output in the file. Of course the output is also influenced by other factors (e.g. other command-line options, properties of the specific NLP being solved etc.) and therefore these options do not determine the complete behavior of the software.

**Console Output (output)** Console outputting is generated by a default `IterationPack::AlgorithmTracker` subclass object of type `MoochoPack::MoochoTrackerConsoleStd`. This output is designed to approximately fit in an 80 character wide console. Here is the output that is generated for this example program:

Above, one of the first things printed is the size of the NLP where `n` is the total number of variables, `m` is the total number of equality constraints and `nz` is the number of nonzeros in the Jacobian  $\nabla c(GC)$ . Note that for a simulation-constrained optimization problem that `nz` will not give any useful information since this is not available through the Thyra interfaces.

Following the global dimensions of the problem, a table containing summary information for each rSQP iteration is printed in real time. Each column in this table has the following meaning:

- **k** : The SQP iteration counter. This count starts from zero so the total number of SQP iterations is one plus the final  $k$ .
- **f** : The value of the objective function  $f(x)$  (possibly scaled) at current estimate of the solution  $x_k$
- **||c||s** : The scaled residual of the norm of the equality constraints  $c(x)$  at current estimate of the solution  $x_k$ . The scaling is determined by the convergence check (see the step "CheckConvergence" in [MoochoAlgo.out](#) and [MoochoJournal.out](#)) and this value is actually equal to the iteration quantity `feas_kkt_err` (see the file [MoochoAlgo.out](#)). This is the error that is compared to the tolerance `feas_tol` in the convergence check (which is equal to the option `NLPSolverClientInterface{feas_tol}`). The unscaled constraint norm can be viewed in the more detailed iteration summary table printed in the file [MoochoSummary.out](#).
- **||rGL||s** : The scaled norm of the reduced gradient of the Lagrangian  $Z^T \nabla_x L$  at current estimate of the solution  $x_k$ . The scaling is determined by the convergence check (see the step "CheckConvergence" in [MoochoAlgo.out](#) and [MoochoJournal.out](#)) and this value is actually equal to the iteration quantity `opt_kkt_err` (see the file [MoochoAlgo.out](#)). This is the error that is compared to the tolerance `opt_tol` in the convergence check (which is equal to the option `NLPSolverClientInterface{opt_tol}`). The unscaled norm can be viewed in the more detailed summary table printed in the file [MoochoSummary.out](#).
- **QN** : This field indicates whether a quasi-Newton update of the reduced Hessian was performed or not. The following are the possible values:
  - **IN** : Reinitialized (usually to identity  $I$ )
  - **DU** : A dampened update was performed
  - **UP** : An undamped update was performed
  - **SK** : The update was skipped on purpose
  - **IS** : The update was skipped because it was indefinite
- **#act** : Number of active constraints in the QP subproblem. This field only has meaning for an active-set algorithms. For interior-point algorithms, this will just equal the number of bounded variables and does not provide any useful information. For problems without any bounds or inequality constraints, this column is not shown.
- **||Ypy||2** : The  $||\cdot||_2$  norm of the quasi-normal contribution  $(Yp_y)_k$ . This norm gives a sense of how large the feasibility steps are.
- **||Zpz||2** : The  $||\cdot||_2$  norm of the tangential contribution  $(Zp_z)_k$ . This norm gives a sense of how large the optimality steps are.

- **`||d||inf`** : The  $||\cdot||_\infty$  norm of the total step  $d_k = (Yp_y)_k + (Zp_z)_k$ . This norm gives a sense of how large the full SQP steps are in  $x$ .
- **`alpha`** : The step length taken along  $x_{k+1} = x_k + \alpha d_k$ . A step length of  $\alpha = 0$  represents a major event in the algorithm such as a line search failure followed by the selection of a new basis or a QP failure followed by a reinitialization of the reduced Hessian. A small number for  $\alpha$  indicates that many backtracking line search iterations were required and is an indication that the computed search direction  $d_k$  is of poor quality. A value of `alpha=1.0` usually indicates that the algorithm is taking full spaces and may be performing well.
- **`time(s)`** : The total wall-clock time consumed by the algorithm to that point. By differencing the times between iterations, one can compute the amount of time taken for each iteration. See the more detailed timing output in the file [MoochoSummary.out](#).

After the iteration summary is printed, the total wall-clock time is given in `Total time`. This is the wall-clock time that is consumed from the time that the `MoochoPack::MoochoTrackerConsoleStd` object is first initialized up until the time that the final state of the algorithm is reported. Therefore, this wall-clock time may contain more than just the execution time of the algorithm proper. For more detailed built-in timings, see the table at the end of the file [MoochoSummary.out](#).

Following the total runtime, the total number of function and gradient evaluations is given for the objective and the constraints. Note that if finite difference testing is turned on, then many extra evaluations will be performed and this will inflate these counters.

**Algorithm Configuration Output ([MoochoAlgo.out](#))** In addition to output the console, MOOCHO will also write a file called [MoochoAlgo.out](#) by default that gives information about what MOOCHO algorithm is configured and what logic went into its configuration. This file is too long to be shown here. This file provides the road map for determining what iteration quantities are being used by the algorithm, what the algorithmic steps are, and what the logic of the algorithm is using a shorthand, Matlab-like, notation. This file is the first place to go when trying to figure out what a MOOCHO algorithm is doing and is critical to understand the [MoochoJournal.out](#) file.

Many of the options specified in the options file are shown in the printed algorithm. The user can therefore study the algorithm printout to see what effect some of the options have. For example, the option `NLPSolverClientInterface{opt_tol}` is used in the Step "CheckConvergence" under the name `opt_tol` in the files [MoochoAlgo.out](#) and [MoochoJournal.out](#). Some of the options only determine the algorithm configuration, which affects what steps are included, how steps are set up and in what order they are included. These option names are not specifically shown in the algorithm printout per-se. For example, the option

```
NLPAlgo_ConfigMamaJama{max_dof_quasi_newton_dense}
```

determines when the algorithm configuration will switch from using dense BFGS to using limited-memory BFGS but this identifier name `max_dof_quasi_newton_dense` is not shown anywhere in the listing. However,

the configuration object can print out a short log (to the `MoochoAlgo.out` file) to show the user how these options impact the configuration of the algorithm.

**Algorithm Summary and Timing Output ([MoochoSummary.out](#))** The file [MoochoSummary.out](#) contains a more detailed summary table than what is sent to the console, a table of the timings for each algorithm step for each iteration, and some limited profiling-type output (produced by `Teuchos::TimeMonitor`).

**Algorithm Journal Output ([MoochoJournal.out](#))** The file [MoochoJournal.out](#) contains more detailed, iteration by iteration, step by step information on what the algorithm is doing. The steps shown in this output are the same that are shown in the pseudo algorithm description shown in the file [MoochoAlgo.out](#) described above. The amount of output produced in this file is mainly controlled by the option `NLPSolverClientInterface{journal_output_level}` and the value of `PRINT_ALGORITHM_STEPS` is usually the most appropriated in most cases and prints only  $O(k)$  output, where  $k$  is the SQP iteration counter. The value of `ITERATION_QUANTITIES` will produce obscene amounts of debugging output and will dump nearly every vector and every matrix used in the algorithm. There are many options in the `Moocho.opt` options file that control exactly what type of output is generated to meet different needs. Note that the option `NLPSolverClientInterface{null_space_journal_output_level}` will override `NLPSolverClientInterface{journal_output_level}` for quantities that lie in the null space. This is helpful for seeing the progress of the algorithm where there are few degrees of optimization freedom.

#### 1.5.2.4 Algorithm Interruption

All MOOCHO algorithms can be interrupted at any time while the algorithm is running and result in a graceful termination, even for parallel runs with MPI. When running in interactive mode (i.e. the user has access to standard in and standard out at the console) then typing `Ctrl-C` will cause the algorithm to pause at the end of the current algorithm step and menu like the following will appear:

```
IterationPack::Algorithm::interrupt(): Received signal SIGINT. Wait for
the end of the current step and respond to an interactive query, kill
the process by sending another signal (i.e. SIGKILL).

IterationPack::Algorithm: Received signal SIGINT.
Just completed current step curr_step_name = "EvalNewPoint", curr_step_poss = 1
of steps [1...9].
Do you want to:
(a) Abort the program immediately?
(c) Continue with the algorithm?
(s) Gracefully terminate the algorithm at the end of this step?
(i) Gracefully terminate the algorithm at the end of this iteration?
Answer a, c, s or i ?
```

To terminate the algorithm gracefully at the end of the current step, type 's', which brings up the next question:

Terminate the algorithm with true (t) or false (f) ?

Answering false ('f'), which is interpreted as failure, results in the algorithm exiting immediately with the partial solution being returned to the NLP object and everything being cleaned up correctly on exit. The full output from this type of interrupt looks something like:

```
*****
*** Start of rSQP Iterations ***
n = 1331, m = 1111, nz = 1478741

  k      f          ||c||s      ||rGL||s  QN ||Ypy||2 ||Zpz||2 ||d||inf alpha  time(s)
  ---
    0      2.1          0.11      0.095 IN  1e+001      7          5          1      1.152
    1      4.3      0.00025      0.27 UP      0.1          2      0.1          1      2.294
    2      4.1      8.5e-006      0.25 DU      0.007          3      0.3          1      3.405
```

```
IterationPack::Algorithm::interrupt(): Received signal SIGINT. Wait for the end of
the current step and respond to an interactive query, kill the process by sending
another signal (i.e. SIGKILL).
```

```
IterationPack::Algorithm: Received signal SIGINT.
Just completed current step curr_step_name = "EvalNewPoint", curr_step_poss = 1 of
steps [1...9].
```

Do you want to:

- (a) Abort the program immediately?
- (c) Continue with the algorithm?
- (s) Gracefully terminate the algorithm at the end of this step?
- (i) Gracefully terminate the algorithm at the end of this iteration?

Answer a, c, s or i ? s

Terminate the algorithm with true (t) or false (f) ? f

```
-----
  3      3.4      -      -      -      -      -      -      -      7.762
```

Total time = 7.762 sec

Oops! Not the solution. The user terminated the algorithm and said to return non-optimal!

Number of function evaluations:

```
-----
f(x)   : 10
c(x)   : 10
Gf(x)  : 5
Gc(x)  : 5
Some algorithmic error occurred!
```

A MOOCHO algorithm can also be interrupted without access to standard in or standard out (i.e. when running in batch mode) by setting up an interrupt file. When the interrupt file is found, the algorithm is terminated. MOOCHO must be told to look for an interrupt file by setting the option `IterationPack_Algorithm{interrupt_file_name="interrupt.in"}` where any file name can be substituted for the

name "interrupt.in". At the end of each algorithm step, MOOCHO will look for the file "interrupt.in", usually in its current working directory (or an absolute path can be specified as well). If it finds the file it will read it for termination instructions. For example, a interruption file that contains

```
i f
```

will result in the algorithm terminating at the end of the current iteration with the condition 'false', which means failure. The output generated from this type of interrupt looks something like:

```
*****
*** Start of rSQP Iterations ***
n = 1331, m = 1111, nz = 1478741

  k      f      ||c||s      ||rGL||s  QN ||Ypy||2 ||Zpz||2 ||d||inf alpha      time(s)
  ---
    0      2.1      0.11      0.095 IN  1e+001      7      5      1      1.161
    1      4.3      0.00025      0.27 UP      0.1      2      0.1      1      2.293
    2      4.1      8.5e-006      0.25 DU      0.007      3      0.3      1      3.455

IterationPack::Algorithm: Found the interrupt file "interrupt.in"!
Just completed current step curr_step_name = "EvalNewPoint", curr_step_poss = 1 of
steps [1...9].
Read a value of abort_mode = 'i': Will abort the program gracefully at the end of
this iteration!
Read a value of terminate_bool = 'f': Will return a failure flag!

    3      3.4      1.6e-005      0.23 DU      0.006      7      2      1      4.616
  ---
    3      3.4      1.6e-005      0.23 DU      0.006      7      2      1      4.626

Total time = 4.626 sec

Oops! Not the solution. The user terminated the algorithm and said to return
non-optimal!

Number of function evaluations:
-----
f(x) : 11
c(x) : 11
Gf(x) : 5
Gc(x) : 5
Some algorithmic error occurred!
```

Currently when an algorithm is interrupted and terminated, only the current status of the solution variables are returned to the NLP (i.e. through the `Thyra::ModelEvaluator::reportFinalPoint()` callback function) and no internal check-pointing is performed. Therefore, a user should not expect to be able to restart an interrupted algorithm and have it behave the same as if it was never interrupted. MOOCHO currently does not support general check-pointing and

restarting but this is a feature that is on the wish list for MOOCHO for an upcoming release.

This brings the MOOCHO quickstart to a conclusion. The remaining sections provide more detailed information on topics mentioned in the above quickstart.

## 1.6 Representing Nonlinear Programs for MOOCHO to Solve

In order to utilize the most powerful rSQP algorithms in MOOCHO the NLP subclass must support the `NLPInterfacePack::NLP`, `NLPInterfacePack::NLPFirstOrder`, and `NLPInterfacePack::NLPVarReductPerm` interfaces and must supply an object that supports the `AbstractLinAlgPack::BasisSystem` and `AbstractLinAlgPack::BasisSystemPerm` interfaces. The details of these interfaces are really not the concern of a general user who just wants to solve an NLP. Therefore, here we will only discuss some of the basic issues associated with these interfaces and what adapter-support subclasses are available to help implement the needed functionality.

As described above in the quickstart, there are two well supported tracts to developing concrete NLP subclasses to be used with MOOCHO. Each of these tracts provides support software that allow the user to provide only the most basic types of information needed to define the NLP. The first type of NLPs that are supported are general NLPs with explicit derivative components and these NLPs can only be solved in serial. This first type requires a direct linear solver that can be used to select a basis matrix. The second type are simulation-constrained NLPs that can be solved on massively parallel computers by utilizing preconditioned iterative linear solvers. This type of NLP is supported through the `Thyra::ModelEvaluator` interface and can utilize much of the linear solver capability in Trilinos. The key difference in this second type of NLP is that the application must know *a priori* what the selection of state (or dependent) variables is in order to obtain a square and well conditioned basis matrix.

These two approaches to defining NLPs are described in the next two sections [Representing General Serial NLPs with Explicit Jacobian Entries](#) and [Representing Simulation-Constrained Parallel NLPs through Thyra](#).

### 1.6.1 Representing General Serial NLPs with Explicit Jacobian Entries

One type of NLP that MOOCHO can solve are general NLPs where explicit gradient and Jacobian entries are available. This means that the gradient of the objective function  $\nabla f$  must be available in vector coefficient form and the gradient of the constraints matrix  $\nabla c$  (i.e. the rectangular Jacobian  $\frac{\partial c}{\partial x} = \nabla c^T$ ) must be available in sparse matrix form. In this type of problem, a basis matrix for the constraints need not be known *a priori* but this requires the availability of a linear direct solver that can be used to find a square nonsingular basis from a rectangular matrix. There are a few



direct solvers available that could in principle find a square basis given a rectangular input matrix but MOOCHO only currently contains wrappers for LAPACK (i.e. dense factorization using `DEGETRF ( . . . )`) and the Harwell Subroutine Library (HSL) routine MA28. The MA28 routine is the only viable option currently supported for handling large sparse linear systems. In the past, other direct solvers have been experimented with and an ambitious user can provide support for any direct solver they would like (with the ability to find a square basis) by providing an implementation of the `AbstractLinAlgPack::DirectSparseSolver` interface. If your NLP can also provide explicit objective function gradients, then your concrete subclass should derive from the `NLPInterfacePack::NLPSerialPreprocessExplJac` subclass. More details are given below.

The first utility base subclass for general serial (i.e. runs in a single process or perhaps on an SMP) NLPs is `NLPInterfacePack::NLPSerialPreprocess`. This utility class derives from the `NLPInterfacePack::NLP`, `NLPInterfacePack::NLPObjGrad`, and `NLPInterfacePack::NLPVarReductPerm` interfaces and takes care of a lot of details like preprocessing out fixed variables, converting general inequality constraints to equalities by the addition of slack variables and maintaining the current basis permutations. All of this is done to transform the "original" NLP into standard form. The "original" NLP can include general inequality constraints in addition to general equality constraints. The "original" NLP, however, can also include fixed variables (i.e.  $(x_L)_{(i)} = (x_U)_{(i)}$ ). There are several different intermediate forms of the NLP that a `NLPSerialPreprocess` object maintains in the transformation from the "original" NLP to the final form. The first type of transformation is the addition of slack variables to convert the general inequality constraints into an extra set of equality constraints. This is called the "full" form of the NLP. The second type transformation is the removal of fixed variables which are preprocessed out of the problem but leaving the general inequalities intact which some parts of a MOOCHO algorithm may access (e.g. globalization steps) through the `NLP` interface. The last type of transformation is the permutation of the variables and the constraints according to the current basis selection. All of this functionality is very useful and this makes the `NLPSerialPreprocess` subclass the place to start when going to implement any type of serial NLP to be used with an rSQP algorithm. Note that this subclass does not address the structure or handling of the Jacobian or Hessian matrices in any way. The handling of these matrices is deferred to subclasses to define.

While it may seem that the details of the transformations performed by `NLPInterfacePack::NLPSerialPreprocess` are of no concern to end users, this is not always the case. For example, a user must understand how their original NLP is transformed in order to understand the output printed in the [MoochoJournal.out](#) file when the journal output level `NLPsSolverClientInterface{journal_print_level}` is set to a value equal to or higher than `PRINT_VECTORS`.

Subclasses that wish to use a generic sparse data structure for the Jacobian matrix  $\nabla_c^T$  and a generic sparse direct linear solver to select, factor and solve linear systems

with the basis matrix  $C$  should derive from the `NLPInterfacePack::NLPSerialPreprocessExplJac` subclass (which itself derives from `NLPSerialPreprocess`). This subclass performs all of the same types of transformations as its `NLPSerialPreprocess` base class (i.e. removal of entries for fixed variables, addition of slack variables and basis permutations) with the explicit Jacobian entries that are supplied by the concrete NLP subclass. The concrete implementations of both the Jacobian matrix subclass for `Gc` and the `BasisSystem` subclass can be overridden by the client but yet have good default implementations. The default implementation for the matrix class for `Gc` is `AbstractLinAlgPack::MatrixSparseCOORSerial` (which uses a coordinate sparse matrix format). The implementation of the `AbstractLinAlgPack::BasisSystem` object is handled through a subclass of `AbstractLinAlgPack::BasisSystemFactory` called `AbstractLinAlgPack::BasisSystemFactoryStd`.

The `AbstractLinAlgPack::BasisSystemFactoryStd` subclass can create `AbstractLinAlgPack::BasisSystem` objects implemented through several different direct linear solvers. Currently, only the solvers `LAPACK` (for small, dense Jacobians) and `MA28` (for large, sparse systems) are currently supported (see the options group `BasisSystemFactoryStd` to select what solver to use manually). Note that MOOCHO must be configured with `MOOCHO_ENABLE_MA28` to support the MA28 solver.

**Warning!** This NLP adapter-support software is going to most likely change in a major way before the next major release of Trilinos. Therefore, it is recommended that, if possible, users derive their NLPs from the Thyra-based simulation-constrained interfaces described in the next section [Representing Simulation-Constrained Parallel NLPs through Thyra](#). However, this set of software is the only currently supported way to solve certain types of general NLPs and therefore remains for the time being.

See examples above in the section `moocho_explicit_nlps_examples_sec`.

### 1.6.2 Representing Simulation-Constrained Parallel NLPs through Thyra

Another type of NLP that can be solved using MOOCHO are simulation-constrained NLPs where the basis section is known up front. For these types of NLPs, it is recommended that the NLP be specified through the `Thyra::ModelEvaluator` interface and this provides access to a significant linear solver capability through Trilinos. These types of NLPs can also be solved in single program multiple data (SPMD) mode in parallel on a massively parallel computer.

The `Thyra::ModelEvaluator` interface uses a different notation than the standard MOOCHO NLP notation. The model evaluator notation is:

$$\begin{aligned}
&\text{minimize} && g(x, p) \\
&\text{subject to} && f(x, p) = 0 \\
&&& x_L \leq x \leq x_U \\
&&& p_L \leq p \leq p_U
\end{aligned}$$

where  $x \in \mathfrak{R}^{n_x}$  are the state variables,  $p \in \mathfrak{R}^{n_p}$  are the optimization parameters,  $f(x, p) = 0$  are the discrete nonlinear state simulation equations, and  $g(x, p)$  is the scalar-valued objective function. Here the state Jacobian  $\frac{\partial f}{\partial x}$  must be square and nonsingular. The partitioning of variables into state variables  $x$  and optimization variables  $p$  must be known *a priori* and this partitioning can not change during an optimization solve.

Comparing the MOOCHO notation for optimization problems using variable decomposition methods which is

$$\begin{aligned}
&\text{minimize} && f(x_D, x_I) \\
&\text{subject to} && c(x_D, x_I) = 0 \\
&&& x_{D,L} \leq x_D \leq x_{D,U} \\
&&& x_{I,L} \leq x_I \leq x_{I,U}
\end{aligned}$$

we can see the mapping between the MOOCHO notation and the `Thyra::ModelEvaluator` notation as summarized in the following table:

It is unfortunate that the notation used with the Model Evaluator interfaces and software are different than those used by MOOCHO. The reason for this change in notation is that the Model Evaluator had to first appeal to the forward solve community where  $f(x, p) = 0$  is the standard notation for the parameterized state equation and changing the notation of all of MOOCHO after the fact to match this would be very tedious to perform. We can only hope that the user can keep the above mapping of notation straight between MOOCHO and the Model Evaluator.

Currently, and more so in the near future, a great deal of capability will be automatically available when a user provides an implementation of the `EpetraExt::ModelEvaluator` interface (as shown in the section `moocho_simulation_constrained_nlps_examples_sec`). For these types of NLPs, a great deal of linear solver capability is available through the linear solver and preconditioners wrappers in the `Stratimikos` package. In addition, the application will also have access to many other nonlinear algorithms provided in Trilinos (see the Trilinos packages NOX, LOCA, and Rhythmos).

See examples above in the section `moocho_simulation_constrained_nlps_examples_sec`.

## 1.7 Other Trilinos Packages on which MOOCHO Directly Depends

MOOCHO has direct dependencies on the following Trilinos packages:

- **teuchos**: This package supplies basic utility classes such as `Teuchos::RCP` and `Teuchos::BLAS` that MOOCHO software is dependent on.
- **rtop**: This package supplies the basic interfaces for vector reduction/transformation operators as well as support code and a library of pre-written RTop subclasses. Much of the software in MOOCHO depends on this code.

MOOCHO also optionally directly depends on the following Trilinos packages:

- **thyra**: This package supplies interfaces and support software for SPMD and other types of computing platforms and defines the interface `Thyra::ModelEvaluator` for simulation-constrained optimization that MOOCHO can use to define NLPs.
- **epetraext**: This package provides an Epetra-specific interface for the model evaluator called `EpetraExt::ModelEvaluator` and contains some concrete examples that are used by MOOCHO.
- **stratimikos**: This package supplies Thyra-based wrappers for several serial direct and massively parallel iterative linear solvers and preconditioners.

## 1.8 Individual MOOCHO Doxygen Collections

Below are links to individual doxygen collections that make up MOOCHO:

- **MoochoUtilities**: Collection of a small amount of utility code that is peculiar to MOOCHO. Some of the software that is now in **Teuchos** such as `Teuchos::RCP` and `Teuchos::CommandLineProcessor` were once in this collection.
- **IterationPack**: "Framework" for building iterative algorithms that MOOCHO is based on.
- **RTopPack**: Legacy RTop code that predates Thyra the Trilinos RTop package but it still used by MOOCHO. The current version of the Trilinos **RTop** package was developed from refactored code that once lived in this collection.
- **DenseLinAlgPack**: A C++ class library for dense, BLAS-compatible, serial linear algebra that is similar to classes like `Teuchos::SerialDenseVector` and `Teuchos::SerialDenseMatrix`. This class library is used exclusively by MOOCHO to deal with serial dense linear algebra.

- **AbstractLinAlgPack**: A C++ class library for abstract linear algebra. These interfaces predate and helped to inspire Thyra but at this point should be considered legacy software that should only be used within MOOCHO. It is likely that a future refactoring of MOOCHO will involve largely removing these classes and using Thyra-based software directly instead.
- **NLPInterfacePack**: Set of abstract interfaces based on `AbstractLinAlgPack` for representing nonlinear programs (NLPs) (i.e. optimization problems). These interfaces serve a similar role as the `Thyra::ModelEvaluator` interface but there are many differences. In the future, it is likely that these interfaces will be refactored to look more like the `Thyra::ModelEvaluator` interface but are likely to remain distinct.
- **ConstrainedOptPack**: Collection of utility software for building constrained optimization algorithms that is based on `AbstractLinAlgPack`. Included here are interfaces and adapters for QP solvers (with `QPSchur` being included by default), line search interfaces and implementations, range/null space decompositions and other such capabilities.
- **MoochoPack**: Provides nonlinear optimization algorithms for primarily rSQP methods based on the `IterationPack` framework. This is where the real algorithmic meat of nonlinear programming is found in MOOCHO. This collection provides the "Facade" class `MoochoPack::MoochoSolver`.
- **MOOCHO/Thyra Adapters**: Provides adapter classes for allowing MOOCHO to solve simulation-constrained optimization problems presented as `Thyra::ModelEvaluator` objects. Also included is the higher-level "Facade" class `MoochoPack::MoochoThyraSolver`.

## 1.9 Browse all of MOOCHO as a Single Doxygen Collection

You can browse all of MOOCHO as a [single doxygen collection](#). Warning, this is not the recommended way to learn about MOOCHO software. However, this is a good way to browse the [directory structure of MOOCHO](#), to [locate files](#), etc.

## 1.10 Links to Other Documentation Collections

- **Thyra**: This package defines basic interfaces and support software for abstract numerical algorithms.
- **Thyra ANA Operator/Vector Adapters for Epetra**: This software includes the basic adapters needed to wrap Epetra objects and Thyra objects.

- **Various Thyra Adapters for EpetraExt:** Included here are adapters and interfaces that allow a perspective nonlinear application to specify everything needed to define a wide range of nonlinear problems in terms by subclassing an Epetra-based version of the `Thyra::ModelEvaluator` interface (called `EpetraExt::ModelEvaluator`). This software allows an appropriately defined Epetra-based model to be used to define a Thyra-based model to be used to define an optimization problem that MOOCHO can then solve.
- **Stratimikos: Unified Wrappers for Thyra Linear Solver and Preconditioner Capabilities:** Stratimikos contains neatly packaged access to all of the Thyra linear solver and preconditioner wrappers. Currently, these allow the creation of linear solvers for nearly any `Epetra_RowMatrix` object.

## 2 Module Index

### 2.1 Modules

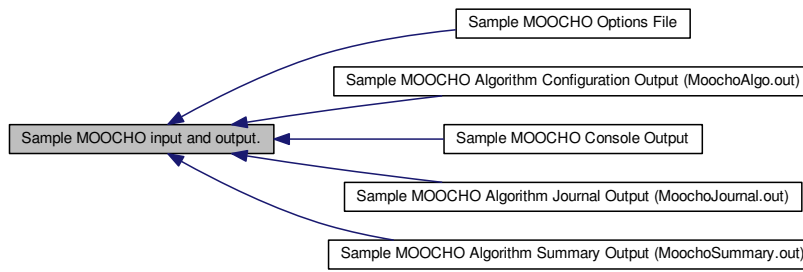
Here is a list of all modules:

<b>Sample MOOCHO input and output.</b>	<b>21</b>
<b>Sample MOOCHO Options File</b>	<b>23</b>
<b>Sample MOOCHO Console Output</b>	<b>25</b>
<b>Sample MOOCHO Algorithm Configuration Output (MoochoAlgo.out)</b>	<b>26</b>
<b>Sample MOOCHO Algorithm Summary Output (MoochoSummary.out)</b>	<b>27</b>
<b>Sample MOOCHO Algorithm Journal Output (MoochoJournal.out)</b>	<b>28</b>

## 3 Module Documentation

### 3.1 Sample MOOCHO input and output.

Collaboration diagram for Sample MOOCHO input and output.:



#### Modules

- [Sample MOOCHO Options File](#)
- [Sample MOOCHO Console Output](#)
- [Sample MOOCHO Algorithm Configuration Output \(MoochoAlgo.out\)](#)
- [Sample MOOCHO Algorithm Summary Output \(MoochoSummary.out\)](#)
- [Sample MOOCHO Algorithm Journal Output \(MoochoJournal.out\)](#)

MOOCHO Notation	Thyra::Model-Evaluator Notation	Thyra::Model-Evaluator Description
$m$	$n_x$	Number of state variables
$n - m$	$n_p$	Number of optimization parameters
$n$	$n_x + n_p$	Total number of optimization variables
$x_D \in \mathbb{R}^m$	$x \in \mathbb{R}^{n_x}$	State variables
$x_I \in \mathbb{R}^{n-m}$	$p \in \mathbb{R}^{n_p}$	Optimization parameters
$c(x_D, x_I) \in \mathbb{R}^n \rightarrow \mathbb{R}^m$	$f(x, p) \mathbb{R}^{n_x+n_p} \rightarrow \mathbb{R}^{n_x}$	State equation residual function
$f(x_D, x_I) \in \mathbb{R}^n \rightarrow \mathbb{R}$	$g(x, p) \mathbb{R}^{n_x+n_p} \rightarrow \mathbb{R}$	Objective function
$C \in \mathbb{R}^{m \times m}$	$\frac{\partial f}{\partial x} \in \mathbb{R}^{n_x \times n_x}$	Nonsingular state Jacobian
$N \in \mathbb{R}^{m \times n-m}$	$\frac{\partial f}{\partial p} \in \mathbb{R}^{n_x \times n_p}$	Optimization Jacobian
$\nabla_D f^T \in \mathbb{R}^{1 \times m}$	$\frac{\partial g}{\partial x} \in \mathbb{R}^{1 \times n_x}$	Derivative of objective with respect to state variables
$\nabla_I f^T \in \mathbb{R}^{1 \times n-m}$	$\frac{\partial g}{\partial p} \in \mathbb{R}^{1 \times n_p}$	Derivative of objective with respect to optimization parameters

Table 1: Mapping of notation between MOOCHO and Thyra::ModelEvaluator for simulation-constrained optimization problems.



## 3.2 Sample MOOCHO Options File

Collaboration diagram for Sample MOOCHO Options File:



Below is a sample MOOCHO options file for some of the typical options that a user might want to manipulate. The full set of options with documentation are shown here.

```
begin_options
```

```
options_group NLPsSolverClientInterface {
    max_iter = 20;
    max_run_time = 2.0; *** In minutes
    opt_tol = 1e-2;
    feas_tol = 1e-7;
    * journal_output_level = PRINT_NOTHING; * No output to journal from algorithm
    * journal_output_level = PRINT_BASIC_ALGORITHM_INFO; * O(1) information usually
    journal_output_level = PRINT_ALGORITHM_STEPS; * O(iter) output to journal [default]
    * journal_output_level = PRINT_ACTIVE_SET; * O(iter*nact) output to journal
    * journal_output_level = PRINT_VECTORS; * O(iter*n) output to journal (lots!)
    * journal_output_level = PRINT_ITERATION_QUANTITIES; * O(iter*n*m) output to journal (big lots!)
    * null_space_journal_output_level = DEFAULT; * Set to journal_output_level [default]
    * null_space_journal_output_level = PRINT_ACTIVE_SET; * O(iter*nact) output to journal
    * null_space_journal_output_level = PRINT_VECTORS; * O(iter*(n-m)) output to journal
    null_space_journal_output_level = PRINT_ITERATION_QUANTITIES; * O(iter*(n-m)^2) output to journal
    journal_print_digits = 10;
    calc_conditioning = true;
    calc_matrix_norms = true; *** (costly?)
    calc_matrix_info_null_space_only = true; *** (costly?)
}

options_group DecompositionSystemStateStepBuilderStd {
    * null_space_matrix = AUTO; *** Let the solver decide [default]
    null_space_matrix = EXPLICIT; *** Compute and store D = -inv(C)*N explicitly
    * null_space_matrix = IMPLICIT; *** Perform operations implicitly with C, N (requires adjoint)
    * range_space_matrix = AUTO; *** Let the algorithm decide dynamically [default]
    * range_space_matrix = COORDINATE; *** Y = [ I; 0 ] (Cheaper computationally)
    range_space_matrix = ORTHOGONAL; *** Y = [ I; -N'*inv(C') ] (more stable)
}

options_group NLPAlgoConfigMamaJama {
    * quasi_newton = AUTO; *** Let solver decide dynamically [default]
    quasi_newton = BFGS; *** Dense BFGS
    * quasi_newton = LBFGS; *** Limited memory BFGS
    * line_search_method = AUTO; *** Let the solver decide dynamically [default]
```

```
*   line_search_method = NONE;           *** Take full steps at every iteration
*   line_search_method = DIRECT;        *** Use standard Armijo backtracking
*   line_search_method = FILTER;        *** [default] Use the Filter line search method
}

end_options
```

### 3.3 Sample MOOCHO Console Output

Collaboration diagram for Sample MOOCHO Console Output:



Below is the console output generated by the program `ExampleNLPBanded.exe` using the command-line arguments

given the `Moocho.opt` options file shown [here](#).

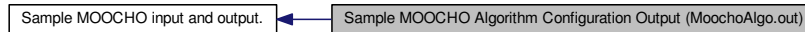
Here is the other types of output that is associated with this run:

- [Sample MOOCHO Algorithm Configuration Output \(MoochoAlgo.out\)](#)
- [Sample MOOCHO Algorithm Summary Output \(MoochoSummary.out\)](#)
- [Sample MOOCHO Algorithm Journal Output \(MoochoJournal.out\)](#)

**Console output:**

#### 3.4 Sample MOOCHO Algorithm Configuration Output (MoochoAlgo.out)

Collaboration diagram for Sample MOOCHO Algorithm Configuration Output (MoochoAlgo.out):



Below is the output file `MoochoAlgo.out` from the program `ExampleNLPBanded.exe` using the command-line arguments

given the `Moocho.opt` options file shown [here](#).

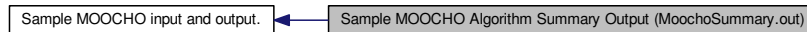
Here is the other types of output that is associated with this run:

- [Sample MOOCHO Console Output](#)
- [Sample MOOCHO Algorithm Summary Output \(MoochoSummary.out\)](#)
- [Sample MOOCHO Algorithm Journal Output \(MoochoJournal.out\)](#)

**Output file `MoochoAlgo.out`:**

### 3.5 Sample MOOCHO Algorithm Summary Output (MoochoSummary.out)

Collaboration diagram for Sample MOOCHO Algorithm Summary Output (MoochoSummary.out):



Below is the output file `MoochoSummary.out` from the program `ExampleNLPBanded.exe` using the command-line arguments

given the `Moocho.opt` options file shown [here](#).

Here is the other types of output that is associated with this run:

- [Sample MOOCHO Console Output](#)
- [Sample MOOCHO Algorithm Configuration Output \(MoochoAlgo.out\)](#)
- [Sample MOOCHO Algorithm Journal Output \(MoochoJournal.out\)](#)

**Output file `MoochoSummary.out`:**

### 3.6 Sample MOOCHO Algorithm Journal Output (MoochoJournal.out)

Collaboration diagram for Sample MOOCHO Algorithm Journal Output (MoochoJournal.out):



Below is the output file `MoochoJournal.out` from the program `ExampleNLPBanded.exe` using the command-line arguments

given the `Moocho.opt` options file shown [here](#).

Here is the other types of output that is associated with this run:

- [Sample MOOCHO Console Output](#)
- [Sample MOOCHO Algorithm Configuration Output \(MoochoAlgo.out\)](#)
- [Sample MOOCHO Algorithm Summary Output \(MoochoSummary.out\)](#)

**Output file `MoochoJournal.out`:**

## 4 Example Documentation

### 4.1 ExampleNLPBandedMain.cpp

### 4.2 NLPThyraEpetraAdvDiffReactOptMain.cpp

### 4.3 NLPThyraEpetraModelEval4DOptMain.cpp

### 4.4 NLPWBCounterExampleMain.cpp