Degeneracy Hunter Documentation

Alex Dowling

June 10, 2014

1 Overview

Degenerate constraints (i.e. constraints that violate LICQ) are prevalent in flowsheet optimization problems, both due to modeler error and features such as recycle loops. Furthermore, they are very problematic when solving NLPs, resulting in a singular matrix being inverted during a simple Newton step. As a result, most modern solvers implement counter-measures to detect and eliminate degenerate threats. We have observed these methods require noticeable computational effort and don't always work. The best approach is to reformulate the original NLP. Unfortunately, this is difficult for complex models with thousands of equations.

Degeneracy Hunter is a MATLAB script/function that enables advanced optimization model analysis, debugging and improvement. It was originally developed as a tool to identify irreducible sets of degenerate equations, which allows expert modelers to focus on only a handful of equations when developing reformulations. The *Degeneracy Hunter* code has also expanded to include optimality condition verification and a search feature. It is organized into four modules:

1. **Setup**. Important Jacobian information into MATLAB. Analyze KKT multipliers to classify inequality constraints and bounds as strongly active, weakly active, or inactive.

2. **Classic *Degeneracy Hunter***. Analyze Jacobian information to determine the smallest set of active equations that are linearly dependent. There are three subsections for this task.

   (a) **Rank Check**. Determine if the active Jacobian is rank deficient and identify non-pivot columns (equations).

   (b) **Heuristic Mode**. Use a heuristic search to find the smallest set of degenerate constraints.

   (c) **Optimal Mode**. Solve a mixed integer linear program to find the smallest set of degenerate constraints.

3. **Second Order Conditions Checker.** Analyze the Jacobian and Hessian to verify second order optimality conditions. If second order conditions are violated, produce a demonstration search direction.

(a) **Calculate Null Space**. Calculate a basis for the null space using equality constraints and *strongly* active inequalities/variable bounds.

(b) **Check Reduced Hessian**. Using a basis for the null space calculate the reduced Hessian.

(c) **Find Search Direction**. If at least one eigenvalue of the reduced Hessian is negative, solve a quadratic program in search of an allowable direction (considering weakly active constraints/bounds) with negative curvature. This approach accommodates the half spaces created by inequality constraints and variables bounds.

4. **Other**. A collection of additional tools.

Armed with the output from each module, modelers are empowered to reformulate their optimization problems to avoid rank deficiencies. This is especially useful when solving large NLPs.

Note: Modules 3 & 4 are still under development. Their usage is not covered in this document.

2 Motivation & Examples

Degenerate equations pose two problems for optimization. First, degeneracies cause the *linearly independence constraint qualification* (LICQ) to be violated. Without LICQ, there is no guarantee the Lagrange (i.e. KKT) multipliers are unique, and thus they are unreliable for sensitivity analysis or solution interpretation. Furthermore, degeneracies in equality constraints cause a "vanilla" Newton step to be ill-defined. As such, most modern solvers include countermeasures to identify and mitigate the effects of degenerate equations. However, these features don't always work and solvers may still get stuck at non-optimal points. Furthermore, these features may also be computationally expensive. The best option is to reformulate optimization problems to remove degenerate equations whenever possible.

There are two classifications of degenerate equations: "global" and "local". Global degeneracies are always present and do not depend on the value of primal (i.e. decision) variables. These commonly occur due to problem over-specification, such as shown in examples 1 and 2 (below). In contrast, local degeneracies are only active at specific values for the primal variables, such as with zero flowrates in chemical equipment (example 3).

2.1 Pressure Relationships and Over-specification

Consider a small synthetic example, shown in Figure 1, containing two vessels, a splitter and seven streams. Stream 1 is fed into the first vessel, which has two outlets: streams 2 and 3. Without loss of generality imagine streams 2 and 3 represent two different phases in equilibrium (*e.g.* vapor and liquid). These streams are feed into the second vessel, producing in effluent streams 4 and 5. Stream 5 is split into streams 6 and 7, and the later is recycled into the first vessel.

Out of simplicity, assume the vessel and splitter effluents are in pressure equilibrium, as shown in (1) - (3). Similarly, assume the splitter can be modeled with no pressure drop,

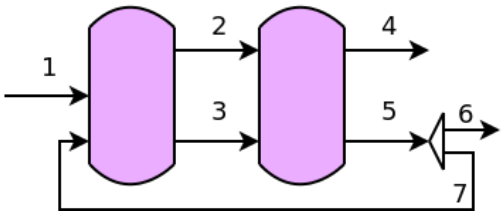

Figure 1: Degeneracy Example: Two Vessels in Series

resulting (4). Also assume the pressure cannot increase in the vessels, resulting in (5) - (7). For inlet/outlet pressure relationships, only one of the outlet streams for each unit is used because the outlet streams are in pressure equilibrium. Finally, assume the pressure for stream 1 is specified (8).

$$P_2 = P_3 \tag{1}$$
$$P_4 = P_5 \tag{2}$$
$$P_6 = P_7 \tag{3}$$
$$P_5 = P_7 \tag{4}$$
$$P_1 \geq P_3 \tag{5}$$
$$P_2 \geq P_5 \tag{6}$$
$$P_7 \geq P_3 \tag{7}$$
$$P_1 = \bar{P} \tag{8}$$

This formulation is degenerate as there are seven variables and eight equations. As currently written, the inequality constraints, (5) - (7), will always be active. The recycle pressure constraint, (7), always ensures there is no pressure drop in the vessels. This can be resolved by either adding a pump/compressor to allow for pressure drop in the vessels (also adding another stream and pressure variable) or removing (7) and converting (5) - (6) into equality constraints.

2.2 Reactor Model and Over-specification

In another example, consider the reaction of A to form B ($A \rightarrow B$) in a continuous stirred tank reactor (CSTR) with inert C. Equations are shown in Table 1. The example is over-specified, as the `SumMoleFrac` equations are not necessary (i.e. redundant).

2.3 Zero Flow and Local Degeneracies

In the third example, consider another flowsheet containing seven streams, two separation vessels and one splitter, shown in Figure 2. The operating conditions of the separation vessels are fixed such that the equilibrium coefficients (K_c) are constant. If the flowrate in one of these vessels go to zero, the component and overall mole balances, (9) & (10), become degenerate. This can be seen by examining the model and its Jacobian.

Table 1: Equations for CSTR Example

`TotalStreamFlow`	$F^s = \sum_c F_c^s$
`StreamComp`	$F_c^s = F^s \; x_c^s$
`SumMoleFrac`	$1 = \sum_c x_c^s$
`CSTRDesignEqn`	$-r_A \; V = F_A^{in} \; X$
`ConvDef`	$F_A^{in} \; X = F_A^{in} - F_A^{out}$
`FormB`	$F_B^{out} - F_B^{in} = F_A^{in} - F_A^{out}$
`RateLaw`	$r_A = k \; C_{A,out}$
`StreamConc`	$F_A^{out} = v \; C_{A,out}$
`Inerts`	$F_C^{in} = F_C^{out}$

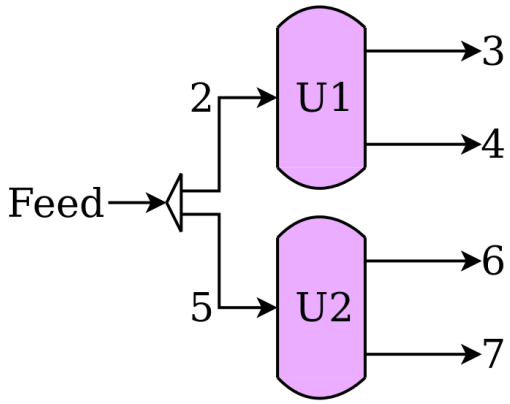

Figure 2: Degeneracy Example: Two Vessels in Parallel

$$s \in \{Streams\},\ c \in \{Components\},\ u \in \{Units\}$$
$$s^l, s^v \in \{Outlet\}(u)$$

$$\sum_{s \in \{Inlet\}(u)} f_{s,c} = \sum_{s \in \{Outlet\}(u)} f_{s,c}, \quad \forall u, c \tag{9}$$

$$\sum_{s \in \{Inlet\}(u)} F_s = \sum_{s \in \{Outlet\}(u)} F_s, \quad \forall u \tag{10}$$

$$y_{s^v,c} = K_{u,c} \; x_{s^l,c}, \quad \forall u, c \tag{11}$$

$$\sum_c (y_{s^v,c} - x_{s^l,c}) = 0, \quad \forall u \tag{12}$$

$$F_s x_{s,c} = f_{s,c}, \quad \forall s, c \tag{13}$$

This example can be extended to include three vessels. If the flows in streams 2 & 4 are both zero, there will be two different sets of degenerate equations.

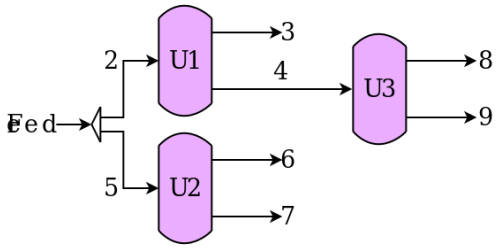

Figure 3: Degeneracy Example: Three Vessels

2.4 Irreducible Degenerate Sets

Although factorization of the active Jacobian is sufficient for the identifying individual degenerate equations (i.e. non-pivot columns), it is still difficult to debug large NLP models. This inspired the creation of *Degeneracy Hunter* to find the smallest sets of degenerate equations, allowing the modeler to focus on a handful of equations instead of thousands (in large problems). Thus, the algorithms in *Degeneracy Hunter* are intended for post optimization analysis and not realtime use embedded in a NLP solver.

3 Algorithms

Before detailing the algorithms for each module, it is helpful to establish some nomenclature. NLP information (Jacobian, Hessian, etc) in GAMS is extracted in the form of (14). Inequality and equality constraints are lumped together. The Jacobian is assembled in MATLAB for analysis corresponding to a NLP in the form of (15) with variables bounds incorporated as inequality constraints. In the full Jacobian, A, shown in (16), the order of constraints, $c(x)$, depends on declaration in GAMS. Variables bounds, $b(x)$, follow the constraints.

$$\begin{aligned} \min_{x} \quad & f(x) && \text{(14a)} \\ \text{s.t.} \quad & c(x) = 0 \text{ or } c(x) \leq 0 && \text{(14b)} \\ & x_L \leq x \leq x_U && \text{(14c)} \end{aligned}$$

$$\begin{aligned} \min_{x} \quad & f(x) && \text{(15a)} \\ \text{s.t.} \quad & c^e(x) = 0 && \text{(15b)} \\ & c^i(x) \leq 0 && \text{(15c)} \\ & b(x) \leq 0 && \text{(15d)} \end{aligned}$$

$$A = \begin{bmatrix} \frac{\partial c_1}{\partial x_1} & \frac{\partial c_1}{\partial x_2} & \cdots & \frac{\partial c_1}{\partial x_n} \\ \frac{\partial c_2}{\partial x_1} & \frac{\partial c_2}{\partial x_2} & \cdots & \frac{\partial c_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial c_m}{\partial x_1} & \frac{\partial c_m}{\partial x_2} & \cdots & \frac{\partial c_m}{\partial x_n} \\ \\ \frac{\partial b_1}{\partial x_1} & \frac{\partial b_1}{\partial x_2} & \cdots & \frac{\partial b_1}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial c_p}{\partial x_1} & \frac{\partial c_p}{\partial x_2} & \cdots & \frac{\partial c_p}{\partial x_n} \end{bmatrix} \tag{16}$$

3.1 Setup Module

In the first module, Jacobian and constraint information from jacobian.gdx and hessian.gdx are imported into MATLAB. Constraints are classified as equality or inequality by analyzing their bounds (`.lo` and `.up` fields). Inequality constraints are further classified as active or strictly active using by examining their multipliers. This is done by applying Algorithm 3.1 to each constraint.

Algorithm 3.1: CHECK EQUATIONS/CONSTRAINTS(c)

if $upperbound - lowerbound \leq \epsilon$
then Equation c is an equality constraint

else
- Equation c is an inequality constraint
- **if** $(upperbound - level < \epsilon) \vee (level - lowerbound) < \epsilon$
  - **then**
    - Equation c is active
    - **if** $\mid multiplier \mid < \epsilon$
      - **then** Equation c is weakly active
      - **else** Equation c is strongly active
  - **else** Equation c is inactive

Bounds are also classified as inactive, weakly active or strongly active using a similar algorithm.

Algorithm 3.2: CHECK VARIABLE BOUNDS(b)

if $(upperbound - level < \epsilon) \vee (level - lowerbound) < \epsilon$
- **then**
  - Variable bound b is active
  - **if** $\mid multiplier \mid < \epsilon$
    - **then** Variable bound b is weakly active
    - **else** Variable bound b is strongly active
- **else** Variable bound b is inactive

Using these two algorithms, the NLP is transformed from (14) to (15) and the full Jacobian, A, is assembled. Subsets of the rows of A are used for analysis in the remaining modules.

3.2 Classic *Degeneracy Hunter* Module

In this module, the Jacobian of active constraints (A_{dh}) is checked for rank deficiency. If A_{dh} is degenerate, the module seeks to find the smallest subset of equations that contain the rank deficiencies. Using this information, the user can reformulate the original optimization model to avoid degeneracies.

3.2.1 Part A: Rank Check

First, the Jacobian of active constraints is assembled (A_{dh}). For this module, the user selects if the analyzed Jacobian should contain weakly active constraints and/or variable bounds. Next, the rank of A_{dh} is calculated using the `rank` command in MATLAB. Currently this employs singular value decomposition. If A_{dh}^T is determined to be rank deficient, it is factorized into reduced row echelon form. This is done using the `frref` command available on MATLAB Central, which also identifies the pivot columns via sparse QR factorization. Equations (and bounds) not identified as pivot columns are considered linearly dependent.

3.2.2 Part B: Heuristic Mode

Heuristic Mode was the first incarnation of *Degeneracy Hunter*. The central idea is to expand a set of equations and bounds until the rank decificies in A_{dh} are fully enclosed. Next, this set of degenerate equations is shrunk using a remove one and check strategy. Ultimately the algorithm produces the smallest set equations with the same rank deficiency as A_{dh}. *Heuristic Mode* essentially obsolete as *Optimality Mode* has theoretical guarantees and is more straightforward to understand. Nevertheless, *Heuristic Mode* remains in the code and this documentation.

Expand Set of Suspect Equations/Bounds

The algorithm starts with a set of suspect linearly dependent equations (and bounds), S_r, either from QR factorization (Part A) or specified by the user. Next, the rows of A_{dh} corresponding to elements of S_r are analyzed. Variables with non-zero elements in these rows are identified and added to S_c (set of suspect columns). Next, the columns of A_{dh} corresponding to the elements of S_c are analyzed and rows with non-zero elements are added to S_r, the set of suspect equations and bounds. Finally the rank of $A_{dh}(S_r)$ is calculated. If the original order of rank deficiency is not found, the algorithm repeats.

This is shown below in Algorithm 3.3. Let A_c be the columns (variables) of A_{dh} whose row indices are in S_c. Likewise, let A_r be the rows (constraints and variable bounds) of A_{dh} whose indices are in S_r. Finally, let r_A and r_r be the ranks of A_{dh} and A_r, respectively.

Algorithm 3.3: EXPAND SUSPECT EQUATIONS(S_r, A_{dh})

$r_A \leftarrow \text{rank}(A_{dh})$
$n_A \leftarrow$ Number of rows in A_{dh}
Assemble A_r using S_r
$r_R \leftarrow \text{rank}(A_r)$
$n_R \leftarrow$ Number of elements in S_r
while $n_A - r_A > n_R - r_R$
do
- $S_c \leftarrow$ indices of columns in A_r with non-zero elements
- Assemble A_c using S_c
- $S_r \leftarrow$ indices of rows in A_c with non-zero elements
- Assemble A_r using S_r
- $r_R \leftarrow \text{rank}(A_r)$
- $n_R \leftarrow$ Number of elements in S_r

By inspection, one can see S_c and S_r only grow as the *while* loop executes. Thus, in the worst case S_r will grow to include all of the rows in A_{dh}. At this time the algorithm will terminate, assuming the calculations of rank are deterministic (which is not true for all sparse rank approximation routines).

Alternately, Algorithm 3.3 can be replaced with singular value decomposition (SVD), as shown in Algorithm 3.4. Assume SVD(A, k) outputs the k smallest singular values (σ) and corresponding (left) singular vectors v of the matrix A. Let n_d represent the number of degenerate equations.

Algorithm 3.4: ALTERNATE EXPANSION(A_{dh})

$r_A \leftarrow \text{rank}(A_{dh})$
$n_A \leftarrow$ Number of rows in A_{dh}
$n_d \leftarrow n_A - r_A$
$\sigma, v \leftarrow \text{SVD}(A_{dh}, n_d)$
if $\sum_{i=1}^{n_d} |\sigma_i| > \epsilon$
  then Display warning: Numeric inconsistency between SVD( ) and rank( )
$S_r \leftarrow \emptyset$
for each $i \in 1, ..., n_d$
$\{S_r \leftarrow S_r \cup \{\text{Indices of non-zero elements in } v_i\}$

Algorithm 3.4 is still in the experimental phase of implementation. Performance differences between Algorithms 3.3 and 3.4 depend on the number of rank( ) calls versus the computation expense of SVD (to obtain vectors), along with the size of A_r produced.

Shrink Set of Degenerate Equations/Bounds

The shrinking phase starts with S_r, a (large) set of rows (constraints and bounds) of A_{dh} that contain all of the rank deficiencies in A_{dh}. These rows are systematically checked; if their removal does not decrease the order of rank deficiency, they are discarded. Ultimately this isolates degenerate rows in A_{dh}. In a final step, A_r is assembled from the shrunk S_r and singular value decomposition is applied. This informs the modeler of how equations in A_r

can be combined to cause rank deficiencies.

Algorithm 3.5: SHRINK DEGENERATE EQUATIONS(S_r)

Assemble A_r from S_r
$r \leftarrow \text{rank}(A_r)$
$iter \leftarrow 1$
$flag \leftarrow$ **true**
while $flag$
do
- $flag \leftarrow$ **false**
- $S'_r \leftarrow S_r$
- **for each** $i \in S_r$
  - Remove equation/bound index i from S'_r
  - Assemble A'_R from S'_r
  - $r' \leftarrow \text{rank}(A'_R)$
  - **if** $r' = r - 1$
    - **then**
      - **comment:** Equation/bound i should remain removed from S'_r
      - $r \leftarrow r'$
      - $flag \leftarrow$ **true**
    - **else**
      - **comment:** Equation/bound i should be added back into S'_r
      - $S'_r \leftarrow S'_r \cup i$
- Print summary of actions for this iteration
- $iter \leftarrow iter + 1$
- $S_r \leftarrow S'_r$

Assemble A_r from S_r
$r \leftarrow \text{rank}(A_r)$
$n_r \leftarrow$ Number of rows in A_r
$n_d \leftarrow n_r - r$
$\sigma, v \leftarrow \text{SVD}(A_r, n_d)$
if $\sum_{i=1}^{n_d} |\sigma_i| > \epsilon$
then Display warning: Numeric inconsistency between SVD( ) and rank( )
for each $i \in 1, ..., n_d$
- Display "Degenerate Subset i"
- **for each** $j \in 1, ..., n_r$
  - **if** $|v_{i,j}| > \epsilon$
    - **then** Display name of equation/bound j and the value for $\sigma_{i,j}$

For every singular value smallest version of A_r, this algorithm reports a subset of S_r that causes the degeneracy. It also reports the elements of the left singular vector, which show how each equation/bound can be added together to cause the degeneracy. This relationship is shown in (17).

$$A^T \ v_i = 0 \quad \forall i \in 1, ..., n_d \tag{17}$$

3.2.3 Part C: Optimal Mode

Alternately, the task of finding the smallest subsets of degenerate constraints (and bounds) can be formulated as a mixed integer linear program, shown in (18). y_i are binary variables that activate/deactivate rows of A_{dh} for consideration as degenerate equations. x_i are elements of a left singular vector of A_{dh}^T. In other words, the vector x shows how the rows (equations and bounds) of A_{dh} can be added together to cause rank deficiency. In (18) the number of non-zero elements of x is minimized, which is equivalent to searching for the smallest set of constraints/active bounds that cause a degeneracy. In order to avoid the trivial solution ($\|x\| = 0$), one element of x, specifically x_j, is set to unity. This ensures $\|x\| > 0$ without explicitly including the non-convex constraint. If the number of degenerate equations in A_{dh} is greater than one, (18) is resolved for each $j \in S_r$, where S_r is the original set of non-pivot equations/active bounds identified via QR factorization (see Section 3.2.1).

$$\begin{aligned}
\min \quad & \sum_{i=1}^{n_{rows}} y_i && \text{(18a)} \\
\text{s.t.} \quad & A_{dh}^T x = 0 && \text{(18b)} \\
& -My_i \leq x_i \leq My_i \ , \quad \forall i = 1, ..., n_{rows} && \text{(18c)} \\
& x_j = 1 && \text{(18d)}
\end{aligned}$$

This approach offers two advantages over the heuristic mode discussed in Section 3.2.2. First, the MILP produces the smallest set of equations/active bounds with a rank deficiency of one. Instead the heuristic mode yields the smallest set of equations/active bounds with the same rank deficiency as A_{dh}. Subsets with a single rank deficiency as then created via SVD. These subsets are not guaranteed to be as small as possible, in contrast to the MILP approach. Second, the MILP approach can produce all subsets of minimal size with rank deficiency of unity containing equation/bound j if the MILP solver stores all solution nodes with the optimal objective function value.

Speed comparisons between the heuristic mode and optimal mode are problem specific. For certain problems, the heuristic algorithm may grow S_r to a very large size, which could cause the shrinking step to be very expensive. Similarly, solutions times with the MILP approach may grow dramatically with large problems (many integer variables).

3.3 Second Order Conditions Checker Module

Occasionally, popular optimization algorithms will terminate at solutions that violate second order conditions. Most concerning these algorithms will sometimes falsely classify these points as optimal local solutions. The *Second Order Conditions Checker* offers GAMS users a practical method to verify these conditions at a solution point.

Checking 2^{nd} order conditions is non-trivial due to the half spaces created by active inequality constraints and variable bounds. For example consider a NLP with both strongly

and weakly active inequality constraints and/or variable bounds. Sufficient 2^{nd} order conditions require reduced Hessian of the Lagrange function to be positive semi-definite. This gives rise to the question *should weakly active constraints be included in Jacobian while calculating the null space?* Including all weakly active constraints may shrink the null space to drastically, leading to falsely claiming second order conditions are satisfied. Similarly excluding all weakly active constraints from null space calculations can result in erroneous negative eigenvalues and search directions that violate the excluded constraints. Necessary conditions require there is no search direction that both maintains feasibility of linearized constraints and reduces the Lagrange function. In this approach, the necessary conditions are checked by solving a quadratic program.

3.3.1 Part A: Calculate the Null Space

The first step is to calculate the null space of the equality constraints and *strongly active* inequalities and variable bounds (A_1). Including weakly active inequality constraints and bounds shrinks the null space, sometimes leading to false positives. The null space satisfies (19).

$$A_1 Z = 0 \tag{19}$$

3.3.2 Part B: Check the Eigenvalue of the Reduced Hessian

Using the basis of the null space (Z) from Part A, the reduced Hessian is calculated using (20), where W is the full space Hessian of the Lagrange function. Next the eigenvalues of the reduced Hessian, W_r, are calculated. If all of the eigenvalues are non-negative, 2^{nd} order conditions are satisfied. Otherwise weakly active constraints and bounds must be considered.

$$W_r = Z^T W Z \tag{20}$$

3.3.3 Part C: Solve a QP to Check Necessary Second Order Conditions

The half spaces introduced by inequality constraints and variables bounds are rigorously considered by solving the quadratic program in (21). This problem finds a search direction Δx that minimizes changes in the Lagrange function while satisfying linearized constraints. $c_e(x^*)$ denotes equality constraints evaluates at the point x^*. c_i denotes (strongly and weakly) active inequality constraints and variable bounds in the form $c_i(x) \leq 0$. The bounds on Δx_i should not impact the sign of the objective function.

$$\begin{aligned}
\min \quad & \Delta x^T \ \nabla_{xx} L \ \Delta x && \text{(21a)} \\
\text{s.t.} \quad & \nabla_x c_e^T(x^*) \ \Delta x = 0 && \text{(21b)} \\
& \nabla_x c_i^T(x^*) \ \Delta x \leq 0 && \text{(21c)} \\
& -1 \leq \Delta x_i \leq 1 && \text{(21d)}
\end{aligned}$$

If the solution to (21) has a negative objective function, 2^{nd} order conditions are not satisfied; in fact, Δx is a search direction that demonstrates a decrease in the Lagrange

function while maintaining feasibility. Concluding 2^{nd} order conditions are satisfied requires a global optimum to the solution for (21). Recall the algorithm terminates at the Part B unless W_r (and thus W) has negative eigenvalues, indicating (21) is non-convex. If the QP is solved to global optimality and the resulting objective function value is zero (ex: $\Delta x = 0$), then 2^{nd} order conditions are satisfied.

4 Usage

4.1 Dependencies and Setup

Before running *Degeneracy Hunter*, two setup issues must be addressed:

1. **Add the GAMS installation directory to the MATLAB path**, otherwise the MATLAB will complain about not finding `rgdx`.

2. **Specify the Jacobian output in GAMS** by selecting `CONVERTD` as the NLP solver and creating convertd.opt as shown below.

   Add this excerpt to the GAMS code after the original solve statement:

   ```
   option NLP = convertd;
   OptimizationProblem.optfile = 1;
   solve OptimizationProblem using NLP minimizing Z;
   ```

   Contents of convertd.opt:

   ```
   jacobian
   hessian
   dict
   ```

4.2 *Degeneracy Hunter* Options

There are five sections at the beginning of the input for user specified options.

4.2.1 Toggle On/Off Core Modules

In this section, the user may toggle on or off the core modules (Degeneracy Hunter, Second Order Condition Checker and Other). The setup module is automatically toggled on if any of the other modules are selected.

```
%% Toggle on/off core modules

% Run Degeneracy Hunter Module
module.degenHunt.tog = true;

% Run Second Order Condition Checker Module
module.SOCcheck.tog = false;
```

```
% Run Other Modules
module.other.tog = false;

% Run Setup Module
if(module.degenHunt.tog || module.SOCcheck.tog || module.other.tog)
  module.setup.tog = true;
end
```

4.2.2 General Settings

In this section the user may specify whether to use sparse or dense linear algebra routes, specify tolerances for classifying constraints and bounds, specify tolerances for the dense rank command and toggle on/off verbose output. In almost all cases, the path to the rank helper function should not be changed:

```
%% General Settings

% Should the code use sparse or dense linear algebra routines?
module.sparse = true;

% Tolerance for classifying multipliers
module.multTol = 1E-10;

% Tolerance for rank command
module.rankTol = 1E-10;

% Verbose output
module.verbose = false;

% Path to rank helper function
myrank = @(A) rankHelper(A,module.sparse,module.rankTol);
```

4.2.3 Degeneracy Hunter Module Specific Settings

First, the user must specifies if (1) weakly active constants (and bounds) and (2) variable bounds should be considered when searching for degenerate equations.

```
%% Degeneracy Hunter Module Specific Settings

% Should weakly active constraints be considered when checking for
% degeneracies?
module.degenHunt.weakAct = false;

% Should variables bounds be considered when checking for degeneracies?
module.degenHunt.varBounds = true;
```

Next, the user must specify which mode of Degeneracy Hunter should be used. If both modes are selected, the heuristic algorithm executes first.

```
% Use the heuristic search to identify degenerate equations
module.degenHunt.heurstic = false;

% Solve MILPs to identify degenerate equations
module.degenHunt.optimal = true;
```

4.2.4 Second Order Condition Checker Settings

This feature is still under development.

4.2.5 Other Module Specific Settings

This feature is still under development.

4.3 Calling *Degeneracy Hunter*

Degeneracy Hunter was recently converted to a MATLAB function (from a script) to enable additional automation and access to three commonly used settings. Below are the steps to run *Degeneracy Hunter*.

1. Run GAMS with the target model. To analyze the initial point, set the iteration limit in GAMS to zero (`option iterlim = 0`).

2. Move `degeneracyHunter3.m` and `GenerateMinDegenerateSet_MIP2.gms` to the folder on step above the folder containing the GAMS output (`Jacobian.gdx`, `Hessian.gdx`, `dict.txt`). In this folder (containing `degeneracyHunter3.m`), run `addpath(pwd)` in MATLAB. This will add the folder to the MATLAB path for the current session. Finally, navigate to the folder containing the GAMS output. Alternately, the *Degeneracy Hunter* code and GAMS output files may be placed in the same directory after modifying a few file paths in the *Degeneracy Hunter* code.

3. Execute `degeneracyHunter3( )` or `degeneracyHunter3(a,b,c)` in MATLAB.

If no options are specified, `( )`, *Degeneracy Hunter* is executed with all of the options defined in the code (see 4.2). Alternately, if three inputs are specified, only the optimality mode of classic *Degeneracy Hunter* is executed. Input `a` toggles on/off weakly active constraints/bounds (i.e. overwrites `module.degenHunt.weakAct`) and input `b` toggles on/off bounds (i.e. overwrites `module.degenHunt.varBounds`). These inputs should be `true` or `false`. Input `c` is the file name to divert console output. If `[]` (an empty string) is specified, outputs is printed to the MATLAB console. Otherwise, *Degeneracy Hunter* attempts to open a text file with the name specified in input `c`; thus this input should be a string.

4.4 Examples

In this section, the outputs from classic *Degeneracy Hunter* for select examples are show and analyzed. Output is shown in "`code`" format, and analysis/commentary is shown in a standard font.

4.4.1 Over-specified CSTR

```
**************************************
********** Basic Information **********
**************************************

********** Equations **********
Total number: 16

Number of equality constraints: 16
Number of STRONGLY active equality constraints: 4
Number of WEAKLY active equality constraints: 12

Number of inequality constraints: 0
Number of ACTIVE inequality constraints: 0
Number of STRONGLY ACTIVE inequality constraints: 0
Number of WEAKLY ACTIVE inequality constraints: 0
Number of INACTIVE inequality constraints: 0

********** Variables **********
Total number: 18
Number of ACTIVE variable bounds: 6
Number of STRONGLY ACTIVE variable bounds: 4
Number of WEAKLY ACTIVE variable bounds: 5

********** Degrees of Freedom **********
At analyzed point, considering...
   all ACTIVE inequalities and bounds: -4
   only STRONGLY ACTIVE inequalities and bounds: -2
```

See comments in the next example. Degree of freedom calculations are not reliable with degenerate constraints.

```
**********************************************
********** Classic Degeneracy Hunter **********
**********************************************
Analyzed Jacobian contains all active inequality constraints
  and NO variable bounds
```

```
Rank of analyzed Jacobian: 16
Estimated number of dependent equations (using rank): 0
Estimated number of dependent equations (using dense rank calc.): 2
```

See comments in the next example. Experience has shown sparse rank calculations tend to underestimate the number of degenerate equations.

```
********** Suspect Equations (and Bounds) **********
Index Active Type Name

9 Weak Eqlty. SumMoleFrac(in)

10 Weak Eqlty. SumMoleFrac(out)


Consider degeneracy with SumMoleFrac(in)
Optimal: This equation is part of a degenerate set with 5 equations (total)
-1.000000   TotalStreamFlow(in)
-1.000000   StreamComp(in,A)
-1.000000   StreamComp(in,B)
-1.000000   StreamComp(in,C)
1.000000   SumMoleFrac(in)

Consider degeneracy with SumMoleFrac(out)
Optimal: This equation is part of a degenerate set with 5 equations (total)
-1.000000   TotalStreamFlow(out)
-1.000000   StreamComp(out,A)
-1.000000   StreamComp(out,B)
-1.000000   StreamComp(out,C)
1.000000   SumMoleFrac(out)
```

There are two sets of degenerate equations. Recall `SumMoleFrac` is redundant and appears in the model twice (once for both the inlet and outlet streams).

4.4.2 Zero Flowrates

Next, consider the example from Figure 3 at a point (possibly a local solution) where the flowrates in streams 2 - 4, 8 and 9 are all zero.

```
**************************************
********** Basic Information **********
**************************************

********** Equations **********
Total number: 38
```

```
Number of equality constraints: 38
Number of STRONGLY active equality constraints: 10
Number of WEAKLY active equality constraints: 28

Number of inequality constraints: 0
Number of ACTIVE inequality constraints: 0
Number of STRONGLY ACTIVE inequality constraints: 0
Number of WEAKLY ACTIVE inequality constraints: 0
Number of INACTIVE inequality constraints: 0

********** Variables **********
Total number: 43
Number of ACTIVE variable bounds: 19
Number of STRONGLY ACTIVE variable bounds: 5
Number of WEAKLY ACTIVE variable bounds: 18

********** Degrees of Freedom **********
At analyzed point, considering...
   all ACTIVE inequalities and bounds: -14
   only STRONGLY ACTIVE inequalities and bounds: 0
```

In this problem there zero degrees of freedom at the solution (when considering only strongly active inequality constraints and variables). Why does this happen? One expects there to be positive degrees of freedom. There are a few possible explanations:

- The degrees of freedom in the problem are the split fractions for each splitter. At this solution, the first split fraction is at its implicit bound and the second doesn't matter due to zero flowrates.
- Classification of strongly active constraints/bounds and the subsequent calculation of degrees of freedom uses the KKT multipliers. However, with degenerate constraints the KKT multipliers are not unique (LICQ is violated).

```
**********************************************
********** Classic Degeneracy Hunter **********
**********************************************
Analyzed Jacobian contains all active inequality constraints
  and NO variable bounds

Rank of analyzed Jacobian: 37
Estimated number of dependent equations (using rank): 1
Estimated number of dependent equations (using dense rank calc.): 2
```

For this run, sparse linear algebra routines are unable. Unfortunately the sparse rank calculation in MATLAB is approximate and non-deterministic. Experience has shown the

dense rank command to be more reliable. If the Jacobian is moderately sized, the estimated number of dependent equations using the full/dense rank command is also shown.

```
********** Suspect Equations (and Bounds) **********
Index Active Type Name

24 Strong Eqlty. EqStrMoleFrac(S4,B)

34 Strong Eqlty. EqStrMoleFrac(S9,B)
```

Factorization reveals two non-pivot equations, index 24 and 34.

```
Consider degeneracy with EqStrMoleFrac(S4,B)
Optimal: This equation is part of a degenerate set with 9 equations (total)
-1.000000   EqUnitComponentMoleBalance(1,A)
-1.000000   EqUnitComponentMoleBalance(1,B)
1.000000   EqUnitMoleBalance(1)
-1.000000   EqStrMoleFrac(S2,A)
-1.000000   EqStrMoleFrac(S2,B)
1.000000   EqStrMoleFrac(S3,A)
1.000000   EqStrMoleFrac(S3,B)
1.000000   EqStrMoleFrac(S4,A)
1.000000   EqStrMoleFrac(S4,B)

Consider degeneracy with EqStrMoleFrac(S9,B)
Optimal: This equation is part of a degenerate set with 9 equations (total)
-1.000000   EqUnitComponentMoleBalance(3,A)
-1.000000   EqUnitComponentMoleBalance(3,B)
1.000000   EqUnitMoleBalance(3)
-1.000000   EqStrMoleFrac(S4,A)
-1.000000   EqStrMoleFrac(S4,B)
1.000000   EqStrMoleFrac(S8,A)
1.000000   EqStrMoleFrac(S8,B)
1.000000   EqStrMoleFrac(S9,A)
1.000000   EqStrMoleFrac(S9,B)

>>
```

In this problem there are two irreducible sets of degenerate equations because there are two units with zero inlet flows. Both sets are identified with *Degeneracy Hunter*. The singular vector element for each equation in the irreducible set is also reported.

5 Dependencies

Degeneracy Hunter uses two linear algebra routines available from MATLAB Central. Both of these codes are included with *Degeneracy Hunter*.

5.1 `frref` - Fast Reduced Row Echelon Form

Link: http://www.mathworks.com/matlabcentral/fileexchange/21583-fast-reduced-row-echelon-form/content/frref.m
Developers: Armin Ataei-Esfahani (2008) & Ashish Myles (2012)
Copyright Notice/License: BSD

```
Copyright (c) 2008, Armin Ataei, Ashish Myles
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are
met:

    * Redistributions of source code must retain the above copyright
      notice, this list of conditions and the following disclaimer.
    * Redistributions in binary form must reproduce the above copyright
      notice, this list of conditions and the following disclaimer in
      the documentation and/or other materials provided with the distribution

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
POSSIBILITY OF SUCH DAMAGE.
```

5.2 `nulls` - Null space for sparse matrix

Link: http://www.mathworks.com/matlabcentral/fileexchange/42922-null-space-for-sparse-matrix/content/nulls.m
Developer: Martin Holters (2013)
Copyright Notice/License: BSD

```
Copyright (c) 2013, Martin Holters
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are
met:

    * Redistributions of source code must retain the above copyright
      notice, this list of conditions and the following disclaimer.
    * Redistributions in binary form must reproduce the above copyright
      notice, this list of conditions and the following disclaimer in
```