I M P L ©

“M a k i n g O p t i m i z a t i o n a n d E s t i m i z a t i o n F a s t e r, B e t t e r, S m a r t e r!”

IMPL© Console Manual

i n d u s t r i@ l g o r i t h m s

“IMPLementing Industrial Optimization & Estimization Applications Faster, Better, Smarter!”

(Better Data + Better Decisions = Better Business)

Release 1.10, January 2026, *IAL-IMPL-CM-1-10*

Copyright and Property of Industrial Algorithms Limited (2012 - 2026), All Rights Reserved.

Introduction

The IMPL Console executable (IMPL.exe) can be called from any Microsoft DOS, PowerShell or Windows Terminal command prompt window and used as a kernel program or computational back-end where the IMPL Console is specifically designed to be an "interfaceable" console or terminal as opposed to an "interactive" console such as Jupyter Notebooks, Mathworks' Matlab, Wolfram's Mathematica, Microsoft's Excel, etc., mathematical optimization solver command consoles such as CPLEX, GUROBI, MOSEK, XPRESS, COPT, etc. or algebraic modeling language (AML) command consoles such as AIMMS, AMPL, GAMS, LINGO, LINGO, LPL, MOSEL, MPL, OPL, etc. This stand alone and interfaceable IMPL Console is useful as it allows the user, modeler or analyst the ability to model and solve problems, sub-problems or puzzles configured in an IML file (Industrial Modeling Language) with other types of companion include files such as the UPS (Unit-Operation-Port-State Superstructure or Universal Plant, Production, Process Schematic) file or any other include file that contains valid IML frame-feature-field syntax using IMPL's comma-separated value (CSV) format as well as the non-standard- or foreign-models contained in the *.ILPet / *.ILP and *.INPet / *.INP foreign-files. The IMPL Console reads several input files (i.e., IMPL.lic, IMPL.set and IMPL.mem) including the IML file and can write several (optional) OML output files which are described further below as well as the EXL export file. There are several console flags that may be specified as command line arguments and are described below with their enumerations and defaults provided. The IMPL Console executable allows the user, modeler or analyst the ability to deploy IMPL for industrial calculation, simulation, estimization (estimation with constraints) and optimization problems via IMPL's unique frame-oriented CSV formatted IML and OML files.

A DOS console window or screen is automatically created which will display all of the IMPL.exe output such as the interfacing, modeling and solving progress log output when USELOGFILE is set to zero (0). If USELOGFILE equals one (1) or two (2), then only the immediate IMPL.exe console output will be displayed in the console window where the other log output messages from the various IMPL shared objects (DLL's / SO's) will be contained in the *.ldt file (see details below). When other computer programming or scripting language interfaces are developed and deployed such as calling the IMPL dynamic link libraries / shared objects directly from C, C++, Fortran, Java, Javascript, C#, VB.Net, Excel/VBA, Python, Julia, Matlab and R, then their console windows may be used to display the IMPL log output messages including of course those log messages written by the user, modeler or analyst in their respective computer programming or scripting language. This is a standard way for the user to view the progress of the problem or sub-problem being modeled and solved and is IMPL's basic Human-Machine Interface (HMI).

If the IMPL Console runs or executes successfully, then zero (0) is returned as the status and one (1) if any error has occurred. The IMPL Console return status of INNON (-99999, default) is returned if the IMPL.lic file is not found, if the IML fact / subject file is not found, if the IMPL setting (IMPL.set or fact. / subject.set) and if the IMPL memory files (IMPL.mem or fact. / subject.mem) are not found or if the *.stp stop file cannot be set or reset. The stop file is IMPL's simple mechanism to gracefully abort, exit, quit or terminate the solving process only when the user, modeler or analyst presses Control-C / CRTL-C on the console screen, window or terminal. And, a related keystroke to pause or suspend the execution of the console screen, window or terminal output, toggle Control-S / CRTL-S.

IMPL Console Flags (Program Arguments)

There are forty-one (41) flags or command line arguments that can be specified, and they are as follows with their respective possible values / enumerations and defaults (shown in ***bolded italics***). The words `obligatory` and `optional` in parentheses are used to indicate which flags must be explicitly specified or configured in the console program call and which ones can be implied from their defaults. **These flags must also have no blanks or spaces between the start of the "-flag=" to its last character as we show below.** These IMPL Console flags, as stated, are the usual or typical command line or system arguments available when calling or invoking main executable programs and are analogous to subroutine and function input arguments.

`-fact=IMLfile (obligatory) & *.bdt (optional)` 1

The fact flag string value must specify the path name and file name where the path name is the usual prefix and **without the *.iml extension or any file type**, of the IML file to be modeled, presolved and solved where the IML file is a domain-specific comma-separated value (CSV) text file which can be edited with any text editor such as Notepad++, Notepad, Wordpad, etc. The fact (or subject) name is simply a label, tag or string identifier used to refer to the single or mono problem or sub-problem contained in IMPL. Only one problem or sub-problem (model and cycle data) instance is configurable and contained in IMPL at a time though it is straightforward to call the IMPL Server routines render(), refresh() and restore() to load and unload different problems or sub-problems quickly and easily when multiple problems (or sub-problems) need to be modeled, presolved and solved iteratively or interchangeably or even simultaneously / concurrently / in-parallel on multi-CPU, -core or -processor machines i.e., model-side parallelism or more technically multi-process parallelism. As mentioned, the fact / subject file name may also have a path name prefixed whereby IMPL will open the file in the directory specified by the path name.

There is also an optional fact.bdt / subject.bdt binary or unformatted file, which is a rendered or serialized file without any variable, constraint, derivative or expression (model) resource-entities, and may be restored or de-serialized if the *.bdt file exists. To render or serialize, set the IMPL setting EXPORTSERIALIZEDFILE to one (1) and run the IMPL Console program which creates the fact.bdt / subject.bdt file without the model resource-entities i.e., variable, constraint, derivative and expression resource-entities. Then, set IMPORTSERIALIZEDFILE equal to positive one (+1) on subsequent or successive runs of the IMPL Console which reads, inputs, loads or imports the serialized file immediately after the fact.iml / subject.iml file is input i.e., after the IMPL Interfaceri() call. Yet, if the fact.bdt / subject.bdt file exists and IMPORTSERIALIZEDFILE equals negative one (-1), then the unformatted binary file will be restored or de-serialized immediately after IMPL Server's root() and reserve() routines are invoked and before the forefact, formulas and formulasfile flags are processed which are before the call to the IMPL Interfaceri() routine and, as mentioned, reads, inputs or imports the IML fact / subject file.

```
-firstfact=”” | SETfile *and* MEMfile (optional)                         2
```

An initial, beginning, starting or first fact flag or subject signal intended solely to read, load, import or input a problem- / model-specific, fit-for-purpose or tailor-made firstfact.set / startsubject.set file *and* a firstfact.mem / startsubject.mem file instead of the fact.set / subject.set and fact.mem / subject.mem or the IMPL.set and IMPL.mem files processed by the IMPLroot() and IMPLreserve() routines respectively. The firstfact flag / startsubject signal can be employed to initialize IMPL with adhoc, bespoke or customized firstfact.set / startsubject.set *and* firstfact.mem / startsubject.mem files especially if the IMPL setting WRITEMEMORYFILE >= 1.0 which must be run or executed previously in order to create or generate a problem- / model-specific memory *.mem file based on the actual or existing memory attributes for the IMPL resource-entities usually default as fact.mem / subject.mem.

```
-forefact=”” | IMLfile | *.bdt (optional)                                3
```

A preliminary or a priori IML file that is read, input or imported immediately *before or prior* to the formulas flag, formulasfile flag and the fact flag. The forefact IML file may contain any of the IML frames and is especially useful for example to provide initial, beginning, starting, opening or default calc-scalar or -singleton and data-set, -list or -vector values for the formulas and fact.iml / subject.iml files. However, if a forefact.bdt file exist (and previously rendered or serialized of course), then this binary or unformatted file is restored or de-serialized in lieu of the forefact.iml file i.e., the forefact.iml file is not read, inputted or imported but the forefact.bdt file is. The forefact flag essentially breaks the IML file processing into two (2) parts, components or sections i.e., one before the formulas and formulasfile flags (forefact.iml or forefact.bdt) and one after them (fact.iml / subject.iml).

As previously mentioned in the fact flag description, see also the IMPL setting IMPORTSERIALIZEDFILE where if set to negative one (-1) will read, input or import a previously serialized binary unformatted fact.bdt / subject.bdt file created when EXPORTSERIALIZEDFILE is set to one (1). Please note that this binary unformatted serialized file is read, inputted or imported when IMPORTSERIALIZEDFILE is minus one (-1) before or prior to the forefact, formulas and formulasfile flags. The convenience of the forefact flag IML file input, import or read however is that no preliminary step is required to serialize a fact. / subject.iml into a fact. / subject.bdt or a forefact.iml into a forefact.bdt although this is optional and available. See also the IMPL setting IMPORTDATATYPE which if equal to zero (0), does not compile (lex and parse) and compute every data frame's value field as a calc-scalar / data-vector-element expression or formula hence reducing the time to read, input or import the data frames. In terms of read, input or import speed, the fact. / subject.bdt and the forefact.bdt binary unformatted files are read the fastest followed by the fact. / subject.iml and forefact.iml files second when the IMPL setting IMPORTDATATYPE equals zero (0).

`-finalfact=”” | IMLfile (optional)` 4

A final, post, last or ultimate IML file to be read, loaded, inputted or imported immediately after, post or following the reading, loading, inputting or importing of the fact flag's / subject$ signal's IML file and the feed flag's / source$ signal's EXL file if it exists.

`-form=`***`sparsic`***` | symbolic (optional)` 5

For linear problems, the form flag values of `sparsic` and `symbolic` are equivalent given that the problem matrix has no nonlinear or non-constant derivatives or expressions and IMPL supplies the first-order partial derivatives or coefficients internally and analytically. Yet for nonlinear problems, IMPL represents the model in a “sparsic” form where only the sparsity-pattern or variable involvement for each nonlinear constraint is supplied by IMPL and small numerical perturbations are used to compute the non-constant or nonlinear 1st-order partial derivatives using machine-coded constraint residual calculations i.e., the constraint equations are hard-coded into IMPL’s native computer programming language (Intel Fortran). Nevertheless, IMPL also supports a “symbolic” form whereby the constraint residual calculations are computed using nonlinear expressions supplied by IMPL in a byte-coded or parsed tokenized format sometimes referred to as postfix or reverse Polish notation (RPN). This byte-coded symbolic format for nonlinear problems is required by nonlinear solvers such as XPREES-SLP and LINDO-SLP and is the primary reason why IMPL supports the symbolic version for all nonlinear constraint.

This symbolic form is useful for two (2) reasons. First, if there are a considerable number of components (quality) in the problem or sub-problem, then the number of variable-groups / -partitions required by the sparsic form will be directly proportional to the number of components causing the first-order derivatives calculations to be slower in the nonlinear solving. Since the symbolic form does not employ variable-groups as each constraint is processed separately and contains the sparsity-pattern as part of its nonlinear expression or formula, the symbolic form may be considerably faster than the sparsic form. For example, if there a total of one hundred (100) assigned, associated or attached components in the quality problem, then at least 100 variable-groups or -partitions are necessary to perturb the model every time 1st-order derivative sensitivity information is required to be computed in order to uniquely and independently calculate or stimulate them. And second, the symbolic form can be used to output the nonlinear constraint expressions into a human readable flat-file (i.e., fact.ndt / subject.ndt) via IMPL Server’s writesymbology() routine which may be useful to the (developer) user, modeler or analyst. **However, the *.ndt file is only available internally to Industrial Algorithms Limited (IAL) for debugging and troubleshooting support and externally if a development license of IMPL is purchased.**

`-fit=`***`discrete`*** `| distribute (optional)` 6

The fit flag value of `discrete` assumes a uniform digitization of time i.e., discrete-time (or dense-time) with a single time-period, time-interval or time-step duration and can be applied to quantity, logistics and quality types of problems whereas the `distribute` fit flag value is not valid for logistics problems at this time. IMPL’s distributed-time formulation is similar to a continuous-time (or sparse-time) model with a common or global time-grid except that in the former the time-points are known exogenously. Specifically, distributed-time is only supported for quantity and quality problems and is not supported for logistics problems. The time-points for distributed-time are computed automatically from the dynamic, cycle, order, proviso, transient, transactional and/or command data of the problem.

`-filter=`*`quantity`* `|` **`logistics`** `| quality (obligatory)` 7

The filter flag value of `quantity` assumes that all logic variables are fixed to either 0 (zero) or 1 (one) and there are no quality variables present and solves a linear or quadratic programming (LP, QP) problem only. The filter flag value of `logistics` (default) solves a mixed-integer linear or quadratic programming (MILP / MIQP / MIP) problem where the logic variables can be finite or free i.e., lie between 0 or 1 or fixed / frozen / forced to be either at 0 or 1. The filter flag value of `quality` solves a nonlinear programming (SLP / SQP / NLP) problem where the logic variables must be fixed to either 0 or 1 similar to quantity. A fourth filter flag value of `qualogistics` is possible but is not available where this would model and solve a mixed-integer nonlinear programming (MINLP) problem (perhaps a future implementation). With regard to the foreign-modeling with Xnnn / Xname and Fmmm / Fname foreign-variables and foreign-constraints respectively, the ILPet / ILP foreign-files are only recognized when the filter equals `quantity` or `logistics` and INPet / INP foreign-files are recognized when the filter equals `quality`.

`-focus=calculation | simulation | estimization |` ***`optimization`*** `(optional)` 8

The focus flag value allows for `calculation`, `simulation` and `estimization` types of problems *but* currently only the `calculation` and `optimization` values are supported given that both simulation and estimization problems may be solved using the appropriate solver with the focus flag set to optimization. The `calculation` value will not call the IMPL Modeler nor the IMPL Presolver as no modeling and solving capability is required. For information purposes, simulation problems require that the degrees-of-freedom must equal zero (0) or a “square” problem i.e., the number of variables equals the number of fixed variables plus the number of equations or equality constraints (exactly specified). Optimization problems have a positive degrees-of-freedom (under-specified) and estimation / estimization (i.e., reconciliation and regression) problems have a negative degrees-of-freedom (over-specified).

`-factor=`***`1D+0`*** `(optional)` 9

The factor flag value is a real positive number and can be used to scale "extensive" quantity variables only i.e., rates, flows and holdups. The factor flag does not alter the "intensive" yields, objective function weights nor the logic and quality variables given that these are known as intensive variables and are not affected by the size or extent of the system. It should be noted that the factor flag is also useful to possibly find different paths to solutions in the logistics optimizer where each new factor value may find different logistics-feasible solutions during the solving / search process. **It is very important to be cognizant that the factor flag should not deviate from one (1.0) for quality optimization problems ONLY if there are any blackbox condition formulas, expressions or relationships where they contain or involve flows and/or holdups as these quantities will NOT be properly and automatically scaled inside their relationships by IMPL and infeasibilities / inconsistencies will occur. Otherwise, all other instances or occurrences of flow and holdup quantities in IMPL are properly and automatically scaled.**

`-fob=`***`0`*** `|` *`+ve`* `|` *`-ve`* `(optional)` 10

The fob flag value is a 64-bit / 8-byte integer number (i.e., can have up to 19 digits) and is used to internally decrypt or un-obscure the IML file if positive (+ve) and non-zero so that it may be read, input, loaded or imported into IMPL encrypted. If the fob flag is negative (-ve), then an EML file is output, printed, written or exported by the IMPL Interfacer which is the encrypted IML file using the ABS(fob) value where a security frame (cf. cipher data) is placed at the top of the file with the positive fob integer number. IMPL's encryption / decryption algorithm is proprietary and will provide reasonable security for the model and data when required i.e., no other user, modeler or analyst will be able to see the details of your model and data in the IML file because it is obscured / encrypted i.e., not human readable.

In addition, when the fob flag is negative (-ve) any ILPet / ILP and INPet / INP foreign-files that exist are also encrypted into corresponding *.elpet, *.elp and *.enpet, *.enp encrypted foreign-files using the absolute value of the fob flag (i.e., sipher signal). In turn, IMPL will also encrypt or obsure the intermediate foreign-files *.ilpetxf1, *.ilpetxf2, *.ilpxf1, *.ilpxf2 and *.inpetxf1, *.inpetxf2, *.inpxf1, *.inpxf2. Decrypting / un-obsuring will occur when the fob flag / sipher signal is positive (+ve) and the encrypted or obfuscated foreign-files are internally decrypted and properly used to configure the foreign-model.

`-factorizer=`***`semisolverless`***`|y12m|pardiso (obligatory for simulation)` 11

The factorizer flag value is used to factorize or perform sparse LU decomposition on a square system or set of linear and/or nonlinear equations if it exists i.e., the number of degrees-of-freedom equals zero (0). If the number of free and/or finite quantity and quality variables equals the number of linear and nonlinear equality constraints, then simulation using direct factorization of the sparse Jacobian matrix of first-order partial derivatives can be used to solve the problem ignoring all inequality constraints and variable bounds. The factorizer flag `pardiso` requires the Intel Math Kernel Library (MKL) to execute a parallel implementation of sparse LU decomposition**.**

```
-fork=solverless |coinmp|glpk|lpsolve|scip|highs|
                  cplex|gurobi|lindo|optonomy|xpress|mosek|copt|
                  ipopt|uno|conopt|knitro|
                  slpqpe_coinmp|slpqpe_glpk|slpqpe_lpsolve|
                  slpqpe_scip|slpqpe_highs|
                  slpqpe_cplex|slpqpe_gurobi|slpqpe_lindo|
                  slpqpe_optonomy|slpqpe_xpress|slpqpe_mosek|slpqpe_copt|
                  secqpe_y12m|secqpe_pardiso|
                  secqpe_sorve|secqpe_y12m_sorve|secqpe_pardiso_sorve|
                  slpqpe_coinmp_sorve|slpqpe_glpk_sorve|slpqpe_lpsolve_sorve|
                  slpqpe_highs_sorve|slpqpe_scip_sorve|
                  slpqpe_cplex_sorve|slpqpe_gurobi_sorve|slpqpe_lindo_sorve|
                  slpqpe_optonomy_sorve|slpqpe_xpress_sorve|slpqpe_mosek_sorve|
                  slpqpe_copt_sorve|ipopt_sorve|uno_sorve|
                  slpqpe_coinmp_spve|slpqpe_glpk_spve|slpqpe_lpsolve_spve|
                  slpqpe_highs_spve|slpqpe_scip_spve|slpqpe_cplex_spve|
                  slpqpe_gurobi_spve|slpqpe_lindo_spve|slpqpe_optonomy_spve|
                  slpqpe_xpress_spve|slpqpe_mosek_spve|slpqpe_copt_spve|
                  ipopt_spve|uno_spve|secqpe_y12m_spve|secqpe_pardiso_spve|
                  sbte_coinmp|sbte_glpk|sbte_lpsolve (obligatory)                12
```

The fork flag value is used to solve LP, QP, MIP and NLP related problems and is where the "magic" occurs or happens so to speak. The solvers `slpqpe_` solve nonlinear LP or QP problems using the various LP and QP solvers suffixed with `coinmp`, `highs`, `scip`, etc. The solvers `secqpe_` solve nonlinear equality-constrained QP problems using the various LU decompositions suffixed `y12m` for example which are useful for data reconciliation and regression problems (i.e., least squares, error-in-variables, maximum-likehood and parameter estimation types of estimization problems). The suffixed solvers with `_sorve` perform a post sensitivity analysis to compute the observability, redundancy and variability estimates using the known sparse Jacobian matrix computed from the `secqpe_` solve i.e., constant first-order derivatives at the solution point which can be derived for any hard inequality bounded and constrained problem (i.e., `slpqpe_highs_sorve` and `ipopt_sorve`). When variable hard bounds and constraint inequalities are required, then solve with an NLP first with warm-start flag to save its variable results data and then solve with `secqpe_sorve` to perform the post sensitivity analysis which is conveniently consolidated for IPOPT (SQP) using `ipopt_sorve` as well for the SLPQPE solvers i.e., `slpqpe_highs_sorve`. Another post analysis techniques are the `sbte_` which "tests" (similar to tightening) each quantity variable's lower and upper bound one-at-a-time for consistency or feasibility. It uses the quantity-only network or superstructure explicitly and thus is a "path-ology" approach to aid in the diagnosis of infeasibilities which is complementary to using presolve to detect infeasibilities in the linear and mostly primal part of the problem and the use of penalty, excursion, exception, elastic, excess, out-of-bound or artificial variables which are minimized in the objective function. And `_spve` computes the variance estimates plus its 95% confidence-intervals for the static / scalar coefficients or parameters in multivariable linear / nonlinear static / dynamic data regression problems. *"May the fork be with you!"*

Note that the solvers `cplex, gurobi, lindo, mosek, optonomy, xpress, copt, conopt` and `knitro` all require separate commercial licensing agreements from their vendors i.e., "bring your own solver" (BYOS). The solver `xpress` can also be used to solve nonlinear problems provided its XPRESS-SLP solver is licensed which is similar to IMPL's SLPQPE but with more convergence algorithmic details as SLPQPE employs a somewhat simpler convergence control criteria for the nonlinear variables and less escalation of the augmented nonlinear constraint error, elastic or excess variables.

In order to view in a Windows terminal the console log when the UNO nonlinear solver is executed, the user, modeler or analyst must change the control panel's regional settings invoked by running <u>"intl.cpl"</u> from the Windows search prompt and in the Administrator tab click on the <u>"Change system locale…"</u> button and select the check box that displays <u>"Beta: UTF-8 worldwide language support"</u> and then Windows must be restarted in order to take effect.

`-flashback=binaryram | binaryfile | `***`flatfile`***` (optional)` 13

The flashback flag value is used to save or retain logistics- or integer-feasible solutions found during the branch-and-bound type of enumerative searches employed in MIP (MILP and MIQP) solvers. All logistics-feasible solution variable results or responses only are saved to binary or unformatted files with extension *.bdt where a "_nnn" is suffixed to the IML file problem name with "nnn" denoting the solution number found. The flashback flag value of `binaryram` means that the solution is also saved to RAM memory (currently not available / supported) and a value of `flatfile` will save the solution in a formatted *.exl file. Furthermore, the user, modeler or analyst specified output *.dat, *.dta, *.adt and/or *.csv files configured in the OML file will be exported, written, printed or outputted with the logistics- / integer-feasible solution number appended with double underscores ("__nnn") similar to the appending of the frequency solution run number when the frequency flag is non-zero.

`-fuse=`***`cold`*** `| warm | warmrandomize (optional)` 14

The fuse flag value is only valid for quality (nonlinear) optimization problems where a `warm` will use the solution variable result values from a previous solve if they exist as initial-values, starting-points/-guesses or default-results for the next solve - see the **fact.vv / subject.vv** file. This warm-start capability assumes that the data structure of the problems or sub-problems from run-to-run or execution-to-execution are identical structurally; essentially, the model is fixed but the data is variable. If the *.vv file is not found or cannot be open, then of course a cold-start is performed as the safe default. It is the user's, modeler's or analyst's responsibility to ensure that the sequential, successive or consecutive runs or executions have the identical IMPL data structures, especially with respect to the variables, and this requirement is usually met when modeling and solving on-line or real-time problems / sub-problems. The `warmrandomize` option uses the existing variable values and randomizes them according to the equation or formula found in the description of the IMPL setting or option RANDOMIZEWARMSTART found in the IMPL.set file.

`-frequency=`***`0,`*** *`> 0 or < 0`* `(optional)` 15

The frequency flag value sets or specifies the number of solving cycles within a solving execution or run and is typically known as a "multi-start" type of search method. It is essentially a do-loop starting from 1 to frequency which will randomize the initial-values, starting-points or default-results starting from the random seed setting and simply incrementing it by the loop iterate or counter i.e., `initial_value = (lower_bound + upper_bound) / 2.0 + (upper_bound - lower_bound) * (random_number(random_seed) - 0.5)` where `random_number` varies uniformly from zero (0.0) to one (1.0). It should be noted that this is for quality variables only i.e., densities, components, properties, conditions and coefficients, where if the model / static data "target" field exists and is non-zero (not RNNON and not zero (0.0)), then this target value will be used as the initial-value, starting-guess or default-result for the quality variable. Furthermore, *.exl files and the output data *.dat / *.dta / *.adt / *.csv files configured in the OML file will be exported, written or outputted with the frequency solution run number appended with double underscores ("__nnn") similar to the appending of the solution number when logistics-/integer-feasible solutions are output in the logistics optimization (MIP).

If the frequency flag is negative (< 0, -ve), then the IMPL Console executable will abort, terminate or exit the frequency loop when the solver status equals either "converged" or "optimal". This option is especially useful during the preliminary configuring of nonlinear (quality) and non-convex problems when the initial-values, starting-points or default-results are admittedly and temporarily poor. *In addition, if frequency < 0 (-ve) and the initial, starting, beginning or first solution or sub-solution is successfully solved i.e., feasible, converged or optimal, then the frequency loop is preempted bypassing the extra frequency solves, runs or executions.*

When foreign-variables and nonlinear foreign-constraints exist in a foreign-model, setting the IMPL setting RANDOMIZEX to one (1) will randomize these variables identical to the technique described above. If the IMPL setting EXPORTSERIALIZEDFILE is set to one (1), then the IMPL Console will serialize the problem's IMPL resource-entities (except its variables and constraints) using the IMPLrender() routine and instead of calling IMPLinterfaceri() to import, read, load or input the data found in the IML file and its companion or include files, the rendered unformatted binary file (fact.bdt / subject.bdt) will be loaded via the IMPLrestore() routine. Implementing the frequency loop with the IMPLrestore() versus the IMPLinterfaceri() (i.e., *.bdt v. *.iml / *.imlet) will reduce the IMPL data processing time.

As well when the fuse flag / start$ signal equals `warmrandomize` and a converged or optimal (sub-)solution is found, this incumbent will be randomized as a warm-start until the next converged / optimal (sub-)solution is found provided the IMPL setting or option RANDOMIZEWARMSTART is non-zero. When the fuse flag / start$ signal equals `warmrandomize` and the frequency flag is non-zero (i.e., frequency > 0), then any existing fact.vv / subject.vv file is deleted and only when a converged / optimal (sub-)solution is found inside the frequency-loop will it be employed as the incumbent and consequently randomized accordingly. Until a valid (sub-)solution is found, the frequency-loop performs the usual or default cold-start.

`-furcate=`***`-99999`*** `(optional)` 16

The furcate flag value is a 32-bit or 4-byte integer number and is used to selectively / conditionally / optionally read / import / input / load certain frames only in the IML file and several uses are further described below for the furcation of different sections of configuration within the same IML file. If this furcate flag value matches, parities, coincides or equals the value in the IML file as indicated by an "#if nnn" directive statement starting exactly in column one (1), where "nnn" is a 32-bit integer or a calculation / calc-scalar expression rounded to the nearest integer, then the frames enclosed from the "#if nnn" to its corresponding "#endif" statement will be imported, inputted, loaded or read (see also the #exit, #error, #warning, #print, #prompt, #omit and #eof directives). In addition, a simple and single OR compound logical / conditional statement is accepted using the "|" pipe character may be configured i.e., "#if nnn | nnn2" where either "nnn" OR "nnn2" will be recognized if it matches the furcate flag value. Since the "nnn" and "nnn2" may be calc-scalars or calculation expressions, elaborate and sophisticated in-line IF(), NOT(), etc. logical and/or relational expressions may be formulated to automate the inclusion and exclusion of certain frames in the IML file. The main purpose for the single furcate flag and the "#if nnn | nnn2" … "#endif" statement is to configure a single IML file that may be re-purposed for many or multiple different and diverse kinds of modeling and solving problem / sub-problem instances, runs, cases, scenarios, situations, samples, etc. such as differentiated by the configuration of quantity, logistics and quality details. Therefore, the furcate flag (and the furcates flags) provide what we call a "conditional configuration" capability to IMPL's IML file and its companion include files.

It is important to note that complete frames with their leader and trailer features must be configured between the "#if nnn | nnn2" and the "#endif" and not individual features, rows or lines as these furcate symbols or statements will not be recognized or respected once a frame is being read, inputted, imported, loaded, processed or parsed i.e., any lines / rows in between the leader and trailer features of a frame. That is, the furcate flag configures only conditional frame imports, inputs, loads or reads in the IML file and not individual conditional feature, line or row imports. In addition, nested "#if" and "#endif" are not supported at all hence the requirement that these keywords must start, begin or commence in precisely column one (1) of the IML file.

The furcate flag is particularly useful to provide a *seamless* integration when solving logistics then quality optimization sub-problems and vice versa known as our Phenomenological Decomposition Heuristic (PDH) which is part of our Industrial / Interactive / Incremental / Inductive / Iterative / Intuitive Decomposition Heuristic (IDH). This is achieved by IMPL automatically selecting the particular logistics / quality sub-solution's frames in the EXL file i.e., the logic setups when the filter flag equals `quality` and the intensive quantity yields when the filter flag equals `logistics`; this is referred to in IMPL as conjunction / congruence data. Similarly, refer to the figure and feed flags as these may also be used to read, input, import or load any particular EXL file containing logistics and quality sub-solutions and logistics- / integer-feasible solutions. In addition and as mentioned previously, the furcate flag may be used to support the user, modeler or analyst to configure one super-problem IML file which contains both the logistics and quality frame details i.e., one IML file with "#if logistics … #endif", "#if quality … #endif" and "#if logistics | quality … #endif" statements included where "logistics" and "quality" are calc-scalar / calculation expressions.

The furcate flag can also be employed to implement a user, modeler or analyst form of what is commonly known as "constraint dropping" by removing or eliminating difficult or complicating constraints where the optimization will take less time to solve when these constraints are not included and may find feasible solutions without them where otherwise the problem is either infeasible or intractable. The concept of the furcate flag is inspired by the notion of "suspensions" and "situations" versus scenarios, samples, surveys, snapshots, speculations, simulations or "substitutions" (submissions, suggestions). A suspension is a temporary removal / exclusion / dropping of a constraint or bound set and a situation is a set of circumstances / conditions, a state of affairs or a collection of pre-scheduled commands, orders, provisos or transactions based on known activities or events and generated by the user, modeler or analyst through typically inductive heuristic rules of thumb usually based on past operational and/or other processing knowledge of the system; although randomizing suspensions, situations and substitutions is also possible. Suspensions and situations allow for adhoc and practical "retractions", "restrictions", "refinements" and "reductions" including even "relaxations" in the problem formulation with the goal of finding good globally feasible (qualogistics) solutions (in reasonable time) to difficult industrial optimization problems through the "process of elimination" and "trial-and-error" (guess-and-check) where the trial is likened to a suspension, situation or substitution and the error relates to whether the problem solution is feasible and useful or not.

It should be highlighted that our use of the word substitution is similar to a suspension or situation except that the constraint is replaced by a similar constraint or construct which achieves a similar result and is particularly useful for data reconciliation and regression problems as well as infeasibility handling. For instance, temporal flow-equaling can be replaced by flow-smoothing when exact flow matching from time-period/-interval/-step to time-period is not required. It is also worth mentioning that suspensions and particularly situations are used for “complexity management” whereas scenarios (speculations, simulations) are used for “uncertainty management” and are sometimes referred to as hypothetical, speculative or theoretical situations. And finally for completeness, samples, surveys and snapshots are primarily used as a combination of both complexity and uncertainty management in terms of estimating key parameters and coefficients to various types and levels of detail in the modeling (i.e., reduced / simplified / lumped / macro versus rigorous / detailed / distributed / micro / molecular) from actual system measurement / sensor / instrument feedback. A relevant example of fitting, training, regressing, matching or calibrating a simplied or reduced model is for finished or final product fuels blending where we approximate the nonlinear and non-ideal blending equations (cf. Ethyl, Dupont / Interaction and Mobil transformation methods) with continuously updated linearized blending equations but with estimated blending-values, blending-numbers or blending-indices from past blending field and laboratory analyzer and flow (recipe, yield) data.

When using suspensions or substitutions, we refer to this as “feasibility rectification / restoration” (infeasibility reduction) as it is possible to help diagnose, debug, troubleshoot and perhaps remedy infeasible problems due to conflicting constraints and/or variable bounds by isolation through successively removing, replacing, dropping or eliminating complicating and possibly inconsistent constraints or bounds. For example, if a logic inconsistency exists in an uptiming constriction for some reason, then removing or deleting this constraint from the problem for a particular unit-operation may lead to a feasible solution which otherwise would not exist and can be used to gain valuable insight into why the problem is infeasible i.e., to possibly find the root cause and remedy the issue. Another well-known technique called “feasibility relaxation” (cf. CPLEX’s and GUROBI’s “feasrelax” routine for example) and also known as an “elastic formulation” can be employed to help debug infeasible problems which involves adding elastic, excursions, exceptions, excesses, errors, infeasibility-breakers, out-of-bounds, safety- / relief-valves or slack / artificial variables that appear in the objective function with penalty-weights. IMPL supports feasibility relaxation by localling configuring non-zero penalty-weights in the cost data frames which can be globally activated using the QUANTITYEXCURSIONWEIGHT, LOGICEXCURSIONWEIGHT and QUALITYEXCURSIONWEIGHT settings in the IMPL.set file if non-zero or in the IML settings frame. IMPL creates excursion variables for all variable bounds and some key logic / logistics constraints although the global excursion- or penalty-weights are not applied to these key logic / logistics constraints. Feasibility relaxation in essence “softens” the hard constraints and bounds to find at least some form of a solution to the otherwise infeasible or inconsistent problem. Interestingly, the interior-point SQP solver IPOPT (and other SQP's) uses the term "restoration phase" during the path to a nonlinear local solution to manage feasibility when hard infeasibilities are detected typically in the nonlinear part of the problem by internally adding and dropping penalty-errors as required. IMPL's SLPQPE solver employs the well-known SLP algorithmic technique of automatically augmenting the problem formulation with penalty-errors for all nonlinear constraints only and only when all penalty-errors are zero (0.0) is the step-bounding applied.

"Feasibility refinement / remodeling" is the term we use for "pre-scheduling" (preliminary scheduling or even partial scheduling) using situations (cf. our "depooling" heuristic descriped below with the feasibility flag) and the power of parallel processing and high performance computing. Pre-scheduling is the straightforward notion of creating and configuring situations to accelerate the time to find good globally feasible solutions through inductive, implicative and inferential reasoning and/or randomizing heuristic rules, applied before the solving, based on prior knowledge of the problem which of course can be iterative (PDH, etc.). Situations can be created or generated in any computer programming or scripting language as these are basically derived by "if <proposition> then <action> else" statements (rules) and should be based on sound user, modeler or analyst judgement typically gathered from past historical and operational experience as well as engineering knowledge such as user-defined static or dynamic and hard- or soft-segregation rules when discharging or unloading diverse feedstock cargos where soft segregations are fuzzy, non-sharp, loose or leaky hard segregations metaphorically speaking (and conversely hard-segregations are non-fuzzy, sharp or tight soft segregations). Parallel, concurrent or simultaneous scheduling optimization can then be executed by spawning multiple situation runs on separate threads, cores, processors or CPU's to find good feasible solutions quickly.

`-furcates1..furcates9 =`***`-99999`*** `(optional)` 17

Identical to the furcate flag , nine (9) other furcates flags are available and are supported in the IML and OML files only (i.e., not currently supported in ILP / INP foreign-files). Furcates flags or selectives signals are indicated by **"#1if nnn | nnn2 … #1endif" to "#9if nnn | nnn2 … #9endif"** to be found in column one (1) only and may be layered, nested, embedded or cascaded.

`-feasibility=`**`satisficeless`** `|` *`logics`* `|` *`quantitylogics`* `(optional)` 18

The feasibility flag allows a previously solved logistics (MILP / MIQP or MIP) optimization *.exl file solution, partial-solution or sub-solution to be selectively read / imported / inputted / imputed using the furcate flag or more appropriately the figure and feed flags. The term "satisfice" is a portmanteau of the words "satisfy" and "suffice" and was first coined by Noble Lauriat Hubert E. Simons to describe feasible (and not optimal) heuristic solutions to computationally difficult problems where heuristic in this context means to find or discover by trial-and-error (guess-and-check) or the process-of-elimination.

The feasibility flag is very practical and powerful to enable the user, modeler or analyst to interactively, incrementally, iteratively and inductively search for globally feasible logistics solutions with the overriding goal of finding good optimized logistics-feasible solutions quickly. This type or kind of primal heuristic is what we refer to as our Industrial / Interactive / Incremental / Inductive Iterative / Intuitive Decomposition Heuristic (IDH) which is related to the other logistics decomposition heuristics known as the Chronological Decomposition Heuristic (CDH, divide-and-conquer, rolling-horion), the Flowsheet Decomposition Heuristic (FDH, relax-and-fix) and the Stock Decomposition Heuristic (SDH, priority dispatching rules). The IDH enables the implementation of some form of a "relaxation, fixation and exclusion" (RFE) strategy or scheme for the discrete Boolean / binary logic variables, typically along its temporal and structural (spatial) dimensions, as these are well-known to be the most difficult and complicating variables in the logistics optimization. A reduction in these variables will in general reduce the time to find logistics-feasible solutions but this in only a generality. Decomposition is also a way to decouple, divide, divisionalize and of course decompose a problem into several sub-problems where each sub-problem can be solved in series or sequentially integrating sub-solution data until the full or entrie problem or sub-problem can be solved to some level of optimized feasibility.

A straightforward implementaion of the IDH and pre-scheduling (de-planning, preliminary or partial scheduling) is to create or generate multiple situations for a problem via user-, modeler- or analyst-defined rules based on it's a priori model/master (static) and cycle/transactional (dynamic) data. Then for each situation or a related set of situations, run IMPL on a separate thread, processor, core or CPU in parallel to determine if one or more of the situation solutions are globally logistics-feasible. A practical example of this type of heuristic is what IMPL calls "depooling" and is related to the known issue of tank or vessel rotation, round-robin, swinging and sweeping when there are multiple tanks or storage-vessels in parallel to store the same material. Decisions on which tanks to fill and draw over time is the required outcome and is logic and logistics focused. Depooling is the technique of temporally disaggregating, segregating or splitting a "pooled" or aggregated solution into a sub-solution modeled with multiple tanks per pooled aggregate. Depending on the opening, starting, initial or current lineup, route, transfer or movement to a particular tank, simple rules can be coded to produce multiple situations in terms of which tank is selected next for the transfer and so on by means of configuring external-stream / tee-stream setup orders i.e., TK2 -> TK3 -> TK4 -> TK1 -> TK2, etc. over the time-horizon, -profile, -perspective or -purview. As mentioned previsously, each situation can then be executed in parallel, concurrently or simultaneously to hopefully discover good globally feasible sub-solutions quickly to the depooling sub-problem. It should be emphasized that only setup (logic) orders and not rate (quantity) orders need to be configured as the flows will be computed automatically by IMPL as variables or degrees-of-freedom in the logistics optimization.

When this feasibility flag is set to `logics`, then the unit-operation setup values and the unit-operation-port-state to unit-operation-port-state setups will be selectively imported, inputted, loaded or read using the fade and fix flags described below.

`-fadebefore=`**`-1D+23`**` (optional)` 19

The fadebefore flag is the complement to the fade flag to exclude, ignore, passover, forget or skip commands, orders, transactions, activities, events or provisos found in the *.exl file which have begin- / start-times less than of equal to (<=) or before the fadebefore flag value where fadebefore is in the same time unit-of-measure as the configured time-horizon / -profile / -perspective / -purview duration. **It is useful to hightlight that with the fadebefore flag (internally known as the skipbefore signal), it is possible for the user, modeler or analyst to free the discrete setup logic binaries for the beginning or front of the time-profile, fix the setup variables in the middle via commands, orders, provisos, transactions, etc. and then free them at the end or back of the time-horizon. Hence, the possible combinations with the fadebefore and fade flags are "fix then free", "free then fix then free" and "free then fix" where the fade flag may equivalently be considered as the "fadeafter" flag.**

`-fade=`**`1D+23`**` | >= 0D+0 (optional)` 20

The fade (a.k.a. fadeafter) flag is a complement to the feasibility flag to exclude, ignore, passover, forget or skip commands, orders, transactions, activities, events or provisos found in the *.exl file which have begin- / start-times greater than (>) the fade flag value where fade is in the same time unit-of-measure as the configured time-horizon / -profile / -perspective / -purview duration. The fade flag is configurable to implement what is known as temporal "cross-over" found in the CDH / IDH which simply neglects importing / inputting activities, events, transactions or occurrences too far out into the future time-horizon as deemed by the user, modeler or analyst. The default of 1D+23, or essentially infinitity time, implies that no cross-over is configured as all begin-times will surely be less than this default. Conversely, IMPL will include, input or import all orders, events or occurrences that have begin-times less than or equal to the fade flag value; if fade is negative (< 0), then obviously no orders, activities or occurrences will be considered i.e., the figure and feed flags that point or locate to an existing logistics-feasible sub-solution will be ignored, skipped, passed-over or excluded as outlined below.

One surprisingly effective use of the fade flag is to solve a logistics optimization problem (retaining or saving the logistics- / integer-feasible solutions) with a shorter future time-horizon / -profile than the original problem as this will be easier to solve in MIP as there is less temporal or chronological degeneracy and less degrees-of-freedom due to the shorter time-horizon i.e., a temporal kick- or head-start. (This simple concept is based on the CDH's "divide-and-conquer" heuristic and is what we call a truncated-CDH as there is no backtracking nor backjumping in the depth-first search strategy.) Then, solve a second MIP with the original (longer) future time-horizon and pointing the furcate flag, or the figure and feed flags, to a selected logistics-feasible solution and specifying fade to be some value less than or equal to the shorter time-horizon configured in the first MIP. Different fade flag values for the same or different furcate flag value can be re-solved with the goal of finding logistics-feasible solutions quicker than solving the orginal problem where it should be over-stated that any logistics-feasible sub-solution to the second MIP run is of course a logistics-feasible solution to the original or overall problem. This is an example of the CDH / IDH and in practice has been quite efficient at locating or finding good (optimized) logistical solutions quickly albiet at the expense of manual or user intervention although this procedure can be automated. Furthermore, the divisionalization of the time-horizon or -profile may be achieved by careful inspection of key feed supply, product demand and/or utility order, event, transaction or proviso timings such as cargo delivery, pipeline lifting and shipping vessel arrival and departure time-planks i.e., multiple contiguous or consecutive time-periods/-intervals/-steps i.e., adjacent in time or touching time-periods.

`-fix=`**`0.99999D+0`**` (= `**`1D+0 – 1D-5)`**` | > 0D+0 and < 1D+0 (optional)                21`

The fix flag is a complement to the feasibility flag to cap, clip or limit commands, orders, transactions, provisos or occurrences found in the *.exl file which have fractional discrete Boolean / logic / binary variable values (i.e., > 0.0 and < 1.0). Fractional logic variables such as setups and startups will only occur when we "relax" these variables by declaring them to lie continuously between zero (0.0) and one

(1.0) instead of declaring them to be either zero (0) or one (1) (i.e., [0.0,1.0] versus {0,1}) as is the case for independent binary variables. In IMPL all independent logic variables are declared to be binary unless their lower bound is configured as any negative (-ve) number in the command data. Then they are declared as non-negative and unity continuous to enable the FDH / IDH i.e., they lie between zero (0.0) and one(1.0) and not zero (0) or one (1). The FDH / IDH uses the well-known "relax-and-fix" heuristic to first relax "least" important (complicated, difficult, etc.) or secondary logic variables whereas the "most" important or primary logic variables are left as binary. It should be noted that this is similar to the more recent "fix-and-optimize" heuristic found in the Operations Research, Management Science and Mathematical Programming literature.

And more specifically, a surprisingly effective relax-and-fix strategy is to first relax external-streams, tee-streams or upstream unit-operation-port-states to downstream unit-operation-port-states (UOPSUOPS / UOPS2 / UOPSPSUO) by setting their lower setup bounds to some negative (-ve) number and then to solve the logistics optimization (MIP). The IMPL Modeler will recognize the negative lower bound and will declare the setup or startup logic variable to be continuous (or non-binary/-boolean/-logic) instead of a discrete binary i.e., [0.0,1.0] versus {0,1}. After several logistics-feasible solutions have been found and retained, then reset the external- / tee-stream setup logic variables to zero (0) (or some positive number, +ve) and then re-solve the logistics optimization specifying the furcate flag or the figure and feed flags that matches one of the logistics-feasible *.exl files. If a logistics-feasible solution results from this second logistics optimization re-solve, then its solution successfully defines an optimized globally and logistics-feasible solution to the logistics optimization problem. Similarly, and following the FDH protocol, arbitrary physical or procedureal unit collections, clusters, partitions, segregations, constellations or groups may be defined where a sequence or succession of relax-and-fix runs can be performed to find good, globaly feasible logistics solutions, heuristically. This is required when solving the entire problem takes too long to solve where these UO and UOPSUOPS / UOPS2 / UOPSPSUO collections may be conveniently congifured using the IML aliases.

Furthermore, the fix flag is used primarily to implement a form of "cut-off" for the fractional logic variables where a fractional value found in the previous partial- or sub-solution (as configured by the furcate flag) must be above the fix flag value before its lower and upper bound order, command, proviso or transaction will be considered as fixed to one (1), else they will still be configured / declared as free / finite binary variables. This is to curtail too many fractional logic variables with values greater than 0.5

to have their lower and upper logic bounds set to one (1.0) when they are rounded in the digitization / discretization engine.

It should be reminded again that all commands, provisos, transactions or orders are cumulative when digitized (discretized) so the lower and upper bounds will be summed together with any other overlapping in time transaction or order for the same unit-operation-port-state structure. Importantly, the default implies that only fractional values that are very close or near to one (1) should have their lower and upper logic bounds fixed to one (1). All logic variables below the fix flag will have their bounds retained at zero (0) and one (1) as it is not recommended to fix or freeze logic variables to zero (0) unless they have been explicitly forced to zero (0) outside of the heuristic i.e., by configuring a setup or startup command, transaction or order for example in a situation. Configuring the fix flag close to one (1) and only fixing logic binary / Boolean variables that are at or above one (1) has proven to be an effective fixing strategy given previous work on many MIP primal heuristics.

`-flip=`***`0`***`, `*`1`*`, `*`2`*`, `*`3`*` (optional)` 22

The flip flag simply randomizes using the seed supplied by the RANDSEED IMPL setting to flip-a-coin or spin-a-wheel so to speak to determine if the unit-operation (UO) and/or unit-operation-port-state-unit-operation-port-state (UOPSUOPS / UOPS2 / UOPSPSUO) setup logic order is included when the feasibility flag equals *`logics`*. If flip = 0, then all of the setup logics near or at one (1) are fixed and included as usual, if flip = 1, then only UO setup logic orders will be randomized to determine their inclusion, if flip = 2, then only UOPSUOPS / UOPS2 / UOPSPSUO setup logic orders are randomized and if flip = 3, then both UO and UOPSUOPS / UOPS2 / UOPSPSUO setup logic orders near or at one (1) have randomized inclusions. The intent of the flip flag is to increase the independent binary logic degrees-of-freedom so that possibly other nearby or “k-best” / “k-nearest” integer-/logistics-feasible solutions may be found or discovered when searching around the neighborhood or local search space of some specified or configured incumbent logistics-feasible solution via the figure and feed flags when feasibility = *`logics`*. This is known as the “principle of proximity” (POP) where good discrete (logistics-/integer-feasible) solutions are found near other good discrete solutions.

`-fish=`***`0D+0`***` (optional)` 23

The fish flag, if its magnitude is non-zero, can help to find *different* logistics- / integer-feasible solutions using a previous logistics-feasible or even a logistics-infeasible solution and is called our

Industrial / Incremental / Incumbent Diversification Search (IDS). The IDS is a straightforward technique to perform what is known as *incumbent elimination* which maximizes the difference from an incumbent, candidate or trial solution. The incumbent can be specified using the furcate flag and the include frame or the figure and feed flags where the feasibility flag must be set to `logics` in order for IMPL to recognize this as an incumbent. The fish flag specifies the weight in the total objective function and if zero (0D+0), the IDS is not invoked. In our implementation, only incumbent logic binary variable results / responses that are considered as one (1) (and not zero (0)) using the fix flag, are included in the incumbent diversification constraint and objective function. If the fish flag is positive (+ve) and non-zero, then IMPL maximizes the sum of the terms, one (1.0) minus the discrete binary logic variables, and if negative (-ve) and non-zero, then IMPL maximizes the sum of the terms, one-half (0.5) minus the logic variables, where currently the binary variables are the unit-operation (UO) and unit-operation-port-state-unit-operation-port-state (UOPSUOPS / UOPS2 / UOPSPSUO) setup logics. The fish flag (cf. IDS) is particularly effective when used in conjunction with the "solution-pool" techniques for finding multiple logistics- / integer-feasible solutions from the MIP available in the commercial solvers. In addition, we have found for some problems that including special-ordered-sets-one (SOS1) and declaring all dependent binary variables to be independent (i.e., explicitly re-typed as binary) may be helpful – see the IMPL settings INCLUDESOS1, INCLUDEDEPBINARIES and especially INCLUDEMIPSTARTS which provides the partial incumbent solution for the active UO and UOPS2 setup logics as the initial, starting or beginning discrete solution.

These two techniques combined will aid the user, modeler or analyst in finding *many* logistics-feasible solutions for the same problem data. Fundamentally and as a matter of insight, decomposition and diversification are useful techniques to aid in the search for globally feasible good solutions where another technique called diving, which fixes integral discrete variable values during the heuristic (cf. dive-and-fix heuristics,) is also a key weapon in the on-going war against combinatorics. Refer to the IMPL settings prefixed with IDA_ and IDA2_ representing the Industrial / Iterative Diving Accelerators (IDA's) called the Smooth-and-Dive Accelerator (SDA / IDA, cf. Kelly, "Smooth-and-Dive Accelerator: A Pre-MIP Primal Heuristic Applied to Scheduling", *Computers & Chemical Engineering*, 2003) and the Staged Branch-and-Bound Accelerator (SBBA / IDA2). These techniques or methods are known as MIP primal heuristics to help encourage binary logic variable integrality and to possibly find "good" integer-/logistics-feasible solutions early in the discrete enumerative search known as the branch-and-bound or branch-and-cut. The IDA is currently available with COINMP, SCIP, CPLEX, MOSEK and XPRESS and IDA2 is supported with COINMP, SCIP and HIGHS.

`-folio=`***`0`*** `(optional)` 24

The folio flag can be specified as any positive 64-bit (8-byte) integer to simply offset the logistics- / integer-feasible solution number index or indice found during the logistics optimiztion (MIP / MILP / MIQP) and is especially useful with our Industrial / Incremental / Incumbent Diversification Search (IDS) and the commercial solvers' solution-pool to *intentionally / deliberatiely* (versus incidentally / accidentally) find other logistics-feasible solutions. When a logistics-feasible solution is found, IMPL outputs or exports the solution via the MIP's callback functionality with the fact flag string value suffixed with the solution number plus or added to the folio flag value using the underscore character ("_") as the separator or delimiter (cf. *_nnn.exl and *_nnn.bdt files). In addition, the folio flag (or the scroll signal) is also respected when the filter flag is quantity or quality and in OML (Output Modeling Language) (cf. IAL-IMPL-OML-RM-1-8).

`-folioprefix=`*`""`* `(optional)` 25

The folioprefix flag (scrollprefix signal) can specify a base string length of up to 64 characters as a prefixed or front text string fragment to the folio flag – see also the figureprefix flag. The folioprefixand figureprefix flags can be used to provide more flexibility when managing multiple logistics-feasible solutions or sub-solutions especially with the IDH and IDS.

`-figure=`***`-99999`*** `(optional)` 26

The figure flag can be specified as any positive 64-bit (8-byte) integer number including zero (0) to supply to IMPL a logistics- / integer-feasible solution or a logistics / quality sub-solution. The figure flag value is used in conjunction with the feed flag value above (e.g., "feed_figure.exl" where "_" is used as the delimiter or separator) to provide a logistics-feasible solution to the IDH / IDS and/or to provide a logistics / quality sub-solution for other IMPL (decomposition) heuristics such as the Phenomenological Decomposition Heuristic (PDH) which solves logistics- and quality sub-problems iteratively to find optimized solutions to MINLP (mixed-integer nonlinear programming) types of industrial qualogistics problems and is a form of cross-validation via sample splitting i.e., test or validate a logistics-feasible solution by running the quality optimizer to verify and validate that it is quality-feasible which is ultimately qualogistics-feasible and globally-feasible.

`-figureprefix=”” (optional)` 27

The figureprefix flag (statueprefix signal) can specify a base string length of up to 64 characters as a prefixed or front text string fragment to the figure flag.

`-feed=EXLfile (optional)` 28

The feed flag value specifies both the path and file name (without the *.exl extension or file type) of the base, root or source EXL file to be read, imported or inputted as a logistics-feasible solution or a logistics / quality sub-solution. This EXL (export) file is appropriately suffixed or appended with the figure flag value to point to the exact or specific EXL file containing a logistics- / integer-feasible solution or either a logistics or quality sub-solution. If the figure flag value equals zero (0), then the feed flag is only appended with the *.exl file type / extension and no figure solution number index or indice is appended / suffixed. If the feasibility flag equals “logics”, then the feed flag string value is initialized or defaulted to be equal to the fact flag string value. As well, if the feed.exl / source.exl file is not found on the first try of the open statement, then the fact.iml’s / subject.iml’s path name is prefixed to the feed flag / source signal text string value and a second open command is tried.

`-folder=“drive:\directory\” | ”” (optional)` 29

The folder flag value specifies the directory or path name for all output files generated by IMPL, specifically IMPL’s Interfacer and Presolver, where the last backward (“\”) or forward (“/”) slash is obligatory in the path name. The folder flag is useful to create a separate output directory or sub-directory to be employed as the solution repository for the output of IMPL. **Please note that the folder directory must exist and no error or exception is raised if it does not. Also, the OML file is first expected to be contained in this folder directory as the IMPL Interfacer will point or be directed to this folder in order to read, import or input the OML file. However if it is not found, then for convenience IMPL will immediately try to open the OML file from the same directory / sub-directory as the location of its IML file.**

`-factprefix=”” (optional)` 30
`-factsuffix=”” (optional)` 31

The factprefix and factsuffix flags specify the front and back string name fragments respectively applied to prefix and suffix the fact flag or subject signal string in order to rename its output specifically the EXL export file and the OML output files. For instance, if the fact flag for the IML file is arbitrarily configured as “XXX”, with the factprefix = “” (blank) and the factsuffix = “999”, the resulting EXL file output will have the file name “XXX999.EXL and the OML file name that IMPL is expected to open is not “XXX.OML” but “XXX999.OML”. Furthermore, if the OML output files do not have any prefixed directory / sub-directory path name specified in their file name configuration in the OML file, then the output file names will also have the IMPL Server’s factprefix / subjectprefix and factsuffix / subjectsuffix signal string name fragments applied to rename these output files appropriately based on the fact / subject (root) name.

`-file1=””,…,-file9=””,-fileA=””,…,-fileZ=”” (optional)` 32

The file flag (i.e., file1 to file9 and fileA to fileZ) will replace, substitute or subsume the keywords **INCLUDEFILE1**, **…, INCLUDEFILE9, …, INCLUDEFILEA, …, INCLUDEFILEZ** found in the IML file (cf. the include file frame) with the file1, …, fileZ flag strings. This is useful for the user, modeler or analyst at the IMPL Console to re-direct the importing, inputting or reading of the IML include file frame to any specified file when IMPL is run or executed. These file names may also have a path name prefix whereby IMPL will open the files in the directory specified by the path name.

These file indirections are also handy for on-line or real-time application implementations when there are time-varying or dynamic field sensor, meter, instrument and/or laboratory readings referenced using a tag,value duple. For example, a date- / time-stamped file of tag data containing calc and data frames for instance, may be written at each scheduled cycle, sampling or execution time-period duration and stored in the file problemname_yyyymmmdd_hhmmss.tgd where ***.tgd / *.tag / *.tdg** file type or extension is short for “tag data” of course and yyyy = year, mmm = month, dd = day, hh = hour, mm = minute and ss = second; we have not included milli-seconds given that usually the data processing time to perform file input / output handling operations is somewhat slow comparatively and the milli-seconds are not relevant.

`-formulas=”” (optional)` 33

The formulas flag, allows the user, modeler or analyst to configure, set or initialize one or many calc-scalars / calculations at the command line and before the IMPL Interfacer is called which imports, inputs or reads the IML file. The string length may contain upto **4096** characters (64 * 64) where both the calc-scalar’s symbol name and formula-expression must be delimited by commas i.e., comma-separated values (CSV), for example, “PERIOD,1.0,START,-1.0,BEGIN,0.0,END,10.0” which specifies that the PERIOD = 1.0, START = -1.0, BEGIN = 0.0 and END = 10.0 in the time unit-of-measure of the problem data. It should be notably highlighted that although the right-hand-side (R.H.S.) formula-expression may can data-vector-elements, -rows or -points indexed by square brackets ([]), the left-hand-side (L.H.S.) must be a calc-scalar and not a data-vector-element.

Please note that the quotes ("") are not required after the formulas flag's text when typed in the command line i.e., with and without quotes are both valid. However, if spaces are required in the formulas flag's character string, then quotes ("") are necessary to properly lex and parse the text. However, if the IMPL Console is called from or within Microsoft's PowerShell instead of DOS, then quotes are required / obligatory / mandatory.

The formulas flag is useful to calcize, parameterize, tokenize or symbolize a problem's or sub-problem's IML file with calc-scalars that can be configured, set or initialized outside or external to an IML file at the IMPL Console's command prompt to run or execute several problems / sub-problems simultaneously, concurrently or in parallel. More specifically, the calc-scalars configured at the command line contained inside the formulas flag are ideal to configure what we have referred to as situations similar to but not necessarily the same as what-if's, cases, scenarios, samples, suspensions, substitutions, speculations, etc. for what IMPL calls "batch parallel solving" (BPS) or "model-side parallelism" (MSP, multi-process v. multi-thread). The concept of batch or model-side parallel solving especially for scheduling optimization is simply the notion of solving many / multiple situations simultaneously or concurrently in a batch typically with the logistics optimizer (MIP) which may search both incidentally and intentionally for logistics-feasible solutions or sub-solutions (see also the solution-pools in commercial MIP solvers). Then, these logistics-feasible solutions may be batch / model-side parallel solved for quality-feasible solutions in the quality optimizer yielding of course qualogistics-feasible solutions / sub-solutions maintaining global feasibility versus global optimality. These situations, as mentioned previously, may be generated based on combining inductive reasoning, operating rules, logic implications or conditionals, engineering knowledge, heuristics, preferences, rules of thumb, tricks-of-the-trade, practical know-how, etc. and help to steer, guide and/or manoeuvre via hints, indirect suggestions, subtle implications, etc.

```
-formulasfile="" (optional, *.fms)                                                    34
```

The formulasfile flag is similar to the formulas flag except that a CSV-formated text- / flat-file, with a file type or extension (optionally ***.fms**) and an optional path name, where virtually an unlimited number of multi-lines, multi-rows or multi-features of calculation expressions may be specified containing one line per comma-separated calculation formula formatted as “calc_name1,calc_expression1” on line one then “calc_name2,calc_expression2” on line two, and so forth. The IMPL comment character (“!”) is respected including in-line comments when “!” does not appear exactly in column one (1) of the formulas file. As mentioned previously with respect to the formulas flag, each formulasfile flag may configure a single or mulitple situations with their own unique calcized, tokenized, symbolized or parameterized calc-scalars being able to modify IMPL settings, solver options, hyperparameters and/or any of the capacity data related to quantities, logics, logistics and qualities where the formulas flag is processed before the formulas file. Each of the formulasfile flags may be spawned, run or executed in separate operating-system IMPL Console processes solved in-parallel, simultaneously or concurrently to significantly shorten the time to find good qualogistics-feasible solutions i.e., both logistics- and quality-feasible in reasonable time.

Please note that if the IMPL Console is called from or within Microsoft’s PowerShell, then quotes are required for all file names that include or contain a file type or extension.

`-footingsfile=”” (optional, *.fts)` 35

The footingsfile flag is another CSV-formated text- / flat-file, with a user, modeler or analyst supplied file type or extension (optionally ***.fts**) and an optional path name, where multi-line, multi-row or multi-feature calc-scalar expression formulas separated by commas with a header feature configure the comma-separated calc-scalar / calculation names or symbols including any comments prefixed by IMPL’s comment character (“!”) are respected identical to the formulas file. The footings file must be a grid, table, matrix, block or 2D-array formatted file separated by commas (CSV) containing one or more lines, rows or features of multiple calc-scalar expressions / formulas representing footings which may represent different situations although they may also be equally interpreted as suspensions, substitutions, samples, cases, etc. or even scenarios. Each footing line, row, feature or instance in the file is run sequentially similar to the frequency flag and therefore supports executing a batch, block or collection of IML executions by changing / modifying / altering the specified calc-scalar values before the IML file is imported, inputted or compiled into the IMPL Console program. Please be aware that the string length for any line, row or feature in the footings file may contain upto **<u>4096</u>** characters similar to the size or length of the formulas flag string. And, similar to the frequency flag, each sequential run or execution exports a corresponding EXL file and any OML output *.csv, *dta, *.adt or *.dat files with the suffix of "__-nnn" where nnn is the sequential footings run number index or indice but it is negative (-ve) to distinguish it from the other export and output files generated with the frequency flag.

It is important to note that either the formulas flag or the formulasfile flag are necessary to properly implement the footingsfile flag as the calc-scalars to be reset, replaced or revised by the footings file must not be assigned anywhere in the IML file (i.e., fact.iml / subject.iml or forefact.iml) or its include files as the footings file's calc-scalar resets, replacements or revisions will be incidentally overridden when the IML file is imported, read, loaded or inputted. The forumlas or formulasfile flags however allow the calc-scalars to be assigned outside of or external to the IML file and these may be viewed as an opening, initializing or default footing, starting situation, sample, suspension, substitution, etc. or base case and if the fuse flag equals warm, then the opening footing may provide useful initial-values, starting-guesses/-points or default-results.

`-freeze=`***`0`*** `(optional)` 36

The freeze flag, if greater than zero (0), will pause the execution of the IMPL Console to allow third-party DLL's or SO's to attach to this console program for debugging and troubleshooting of their third-party source code such as the IMPLsupplierlib() callback routine for blackblanks, foreign-models and user, modeler or analyst adhoc, bespoke or custom models, UOfunction() / UOFCN() for blackblocks, dynamic coefficients or the external / extrinsic XFCN(), DATAFCN(), MODELFCN(), WRITEFCN() and DCFCN() callback functions; this flag is typically used by the developer user, modeler or analyst.

For instance, in order to troubleshoot user, modeler or analyst coded XFCN()'s, etc. with the Intel Fortran compiler using Microsoft Visual Studio Community as its integrated development environment (IDE), the user, modeler or analyst must first compile their Intel Fortran source code in debug mode (i.e., not release mode), run the IMPL Console or terminal IMPL.exe with the freeze flag not equal to zero (0), and then in the "Debug" section of the Microsoft Visual Studio IDE, select "Attach to Process" selecting the IMPL.exe process listed.

`-foreign=`*`""`* **`|`** ***`"yes"`*** `|` *`"no"`* `(optional)` 37

The foreign flag, if not equal to "no", will always try or attempt to process the ILPet / ILP or INPet / INP foreign-file(s) if it (they) exists. However, if the foreign flag equals "no" (lower case, small letters), then no foreign-files are processed even if they exist. This is useful when the user, modeler or analyst sub-models foreign-variables and -constraints that are not mandatory / optional.

`-fanfare=`***`shownone or 0`*** `| showall or 1073741823 (= 2^30 - 1) (optional)` 38

The fanfare flag value is an integer bit map or mask with 0 to 30 bits. Depending on the value of fanfare, the various IMPL output files will be written to the same directory as the IML file (i.e., *.dta and *.adt files, etc.) – see IMPLshownone (0) and IMPLshowall (2^30 – 1 = 1073741823). Also refer to the IMPL.hdr file for the other fanfare flag enumerations which are the base two (2) exponents i.e., 2^IMPLshowreport + 2^IMPLshowsummary will only output, write, print, export or dump the *.rdt and *.sdt files respectively. If the fanfare flag equals zero (0 = IMPLshownone = shownone), then no IMPL dump files are written of course as all bits are set to zero (0).

The other configurable fanfare flag keyword enumerations are: *showseriesset, showsymbology, showcatalog, showlist, showparameter, showvariable, showconstraint, showformula, showderivative, showexpression, showreport, showsummary, showspoilings, showscalings, showslackings, showspannings* and *showsensativities.* If the -fanfare flag argument appears multiple times in the DOS / PowerShell command prompt statement or line, then the fanfare flag values are summed, aggregated or added together resulting in multiple IMPL dump files to be outputted, written, printed or exported.

`-fancy=`***`100`*** `| 0 (optional)` 39

Maximize the level of console log output when the fancy flag equals one hundred (100) i.e., the most console log output and minimize console log output when zero (0) i.e., the least.

`-flag=`***`1`*** `| 0 (optional)` 40

Opens with the append attribute an **IMPLperformanceyyyy-mmm-dd.txt** text flat-file which outputs, exports, writes, prints or logs the fact/subject name, the total objective function value, the equality closure, inequality closure, status and total time for the modeling and solving of a problem and is primarily used for performance (regression) testing of IMPL. Coupled with a corresponding IMPLperformance.bat file for example (not shown) containing hundreds of IMPL test instances (i.e., referencing many IML, IMLet, ILP, INP, OML, etc. files), the IMPL Console executable can be run programmatically to test, validate and verify the overall performance of IMPL. **It is important to note that the date corresponds to the date that the IMPL.exe console was last compiled and linked.**

-flagsfile="" (optional, *.fgs)

The flagsfile flag is similar in format to the CSV- or ESV- / =SV-formatted formulas file (*.fms) whereby the user, modeler or analayst may collect a list of some or all of their flags into a flat-file where the left-hand-side is the flag's name and the right-hand-side separated by a comma is the flag's value e.g., **flag_name, flag_value** or **flag_name = flag_value**. The flags file (*.fgs) is processed immediately after the command-line console flags have been processed where the obligatory fact flag may also be found in the flags file (*.fgs).

IMPL Input Files

IMPL.lic | *.lic

License file for both development and deployment versions of IMPL and is expected to be in the same directory where IMPL is executed from and not necessarily where the IMPL Console executable and the IMPL DLL's or SO's (also referred to as binary files) are located or installed into.

For deployment licenses there also must be another *.lic file with the file name as the fact flag string value i.e., fact.lic / subject.lic. The *.lic file is generated by the development version of IMPL (cf. the IMPL Presolver) and is a simple checksum of the sets, lists, parameters, variables, constraints and formulas used in the problem or sub-problem based on the resource-entity's number of roster-enumerators and their index or indice number but not their number of record-entries, reference-events or range-exhibits.

IMPL.set | *.set

Settings file for IMPL with the same directory location as the IMPL.lic file. However, the *.set file may be located in the same directory as the IML file with the same name (i.e., fact.set / subject.set) and this specific file is opened first unless it cannot be found else the IMPL.set file is read next from the same directory as the IML file and last from the same directory as the IMPL.lic file. In-line documentation for each setting is provided in the IMPL.set file supplied with the IMPL installation.

All of the settings in this file may be rearranged in any order from top to bottom to support the user, modeler or analyst positioning or locating frequently changed settings to the top of the settings file. In addition, all of the IMPL.set settings may be modified, updated or changed by configuring the settings frame in the IML file although **please be aware that in the *.set file the delimiter is the equals sign ("=") and in the IML settings frame the delimiter is the comma (",").**

`IMPL.mem | *.mem`

Memory file for IMPL with the same directory location as the IMPL.lic file. However, the *.mem file may be located in the same directory as the IML file with the same name (i.e., fact.mem / subject.mem) and this file is opened first unless it cannot be found else the IMPL.mem file is read next from the same directory as the IML file and last from the same directory as the IMPL.lic file. See also the IMPL setting WRITEMEMORYFILE which will export, print or write a new IMPL.mem memory based on the existing, present or current memory settings scaled or factored by WRITEMEMORYFILE's user, modeler or analyst supplied positive (+ve) non-zero real number value as further described below.

Please note that the order or arrangement of the memory settings in this file must not be altered. The memory file is used to allocate the various resource-entities of IMPL and is especially helpful to minimize or tune, tweak, tinker, trim, touch-up, tidy-up or top-up the amount of memory overhead when IMPL's batch or model-side parallelism (multi-process v. multi-thread) is employed i.e., spawning, spinning-up or starting multiple IMPL problem (or sub-problem) instances / sessions on different CPU's, cores or threads simultaneously / concurrently on the same shared memory machine. This is useful to execute several IMPL runs at the same time and is what we call a "poor man's parallelism" or "batch parallel solving" and is also referred to as "model-side parallelism" which is multi-process versus multi-thread.

It should be noted that the IMPL.lic, *.set and *.mem files are not required to be explicitly read by IMPL as all setting and memory values have defaults provided by IMPL. The license strings, settings and memory values may be input or received directly into IMPL via any computer programming language without these files by calling the IMPLverify(), IMPLreceiveSETTING(), IMPLreceiveMEMORY(), etc. routines. This is to support the real-time and on-line implementation of IMPL without employing the reading and writing of files if required. There is also a memory frame that can be found anywhere in an IML file which allows the user, modeler or analyst the convenience of changing any of the memory sizes for any resource-entity except the trivial series- / range-set. The IMPL Server routines IMPLresize(), IMPLrelease() and IMPLreserve() are used to be enable this capability. In addition, please see the IMPL setting WRITEMEMORYFILE whereby depending on the its real value, IMPL Console will write the fact.mem / subject.mem files in the same directory as the IML file with the actual or existing memory attribute requirements sized, scaled or factored by the real value of WRITEMEMORYFILE if and only if it is greater than or equal to one (1.0).

`IMPL.solver | *.solver`

Individual solver settings files for the various IMPL solvers and they may be located in the same directory as the IMPL executable or another *.solver file located in the same directory as the IML file with the same name (i.e., fact.solver / subject.solver). Identical to the IMPL.set file, all of the solver settings may be rearranged in any order.

In addition, all of the IMPL.solver settings may be modified, updated or changed by configuring a solver settings frame in the IML file although in the *.solver file the delimiter is the equals sign (“=”) and in the IML solver settings frame the delimiter is the comma (“,”). To set a solver setting in the solver setting frame, only upper case capital letters are recognized for the solver settings found in the IMPL. /*.solver file and the solver name plus an underscore (“_”) must prefix the solver setting name. For example, in the `IMPL.highs` solver settings file the solver setting “presolve” exists. To set or reset its value in a solver setting frame, the name HIGHS_PRESOLVE must be configured or used.

`*.iml`

Industrial (input) modeling (IML) language files containing the comma-separated value (CSV) IML frames with their features and fields configuring a problem or sub-problem (cf. the IAL-IMPL-RMQ, IAL-IMPL-RMQL and IAL-IMPL-RMQQ Manuals).

`*.oml`

Output modeling language (OML) files, also in CSV-format, with the same name as the IML file where selected variable results and constraint residuals can be written to one or more user-specified output *.csv / *.dat files (cf. the IAL-IMPL-OML-RM Manual) including selected calc’s and data-sets/-list/-sequence/-series/-vectors. The OML file is useful to pinpoint the results / responses of any particular variable for quick review and may also output the variable responses in column, grid, matrix or table views. The OML output *.csv / *.dat files are straightforward and useful to integrate IMPL to any third-party system or sub-system such as spreadsheets, transactional or time-series databases, etc.

`*.ups`

UOPSS schematic files (also short for universal plant / production / process superstructure, system or schematic) usually written by the IALConstructer.py Python 2.3.5 code found in the Dia open-source application, may be included into any IML file via the include frame. This *.ups file may contain the complete or even partial construction data of the network, graph, block flow diagram, flowsheet, topology, system or superstructure of the problem or sub-problem. It should be noted that the contents of the *.ups may be configured by any means where the Dia with the IALConstructer.py is available as a convenience to visualize the UOPSS using its UOPSS scheme, stencil or palette with its corresponding and Industrial Algorithms Limited (IAL) supplied shapes and sheet files (*.shape and *.sheet) found in IALConstructer.py.

`*.upt`

UOPSS text files (also short for universal plant / production / process text) usually written by the IALConstructer.py Python 2.3.5 code found in the Dia open-source application. All of the Dia "Standard – Outline" object name strings are output to this *.upt in the order they are created on the drawing. The "Standard – Text" objects are the text objects used by IALConstructer.py Python 2.3.5 code to display on the drawing the UOPSS shape or structure name strings and these are found organized in the *.ups file.

`*.ilpet | *.ilp | *.inpet | *.inp`

Foreign-files with scalar-based and somewhat sparsely-formed / simple set-based foreign-variable (Xnnn) and -constraint (Fmmm) names may be used to extend, in an adhoc, bespoke or custom manner, the IMPL modeling with user, modeler or analyst created and/or shared variables and created constraints with respect to adding or creating a full model or a partial sub-model. The format of these files is similar to the defacto standard CPLEX LP formatted files where for nonlinear model extensions, the *.inp file, allows nonlinear (infix versus prefix / postfix) expressions or formulas to be specified. If a quantity or logistics (LP, QP, MILP or MIQP) problem is being solved, then only the *.ilp (and *.ilpet) file is recognized and if a quality (NLP) problem is being solved, then only the *.inp (and *.inpet) file is respected (cf. the IAL-IMPL-ILP-INP-RM Manual).

Both the *.ilp and *.inp files include the following sections: "Objective", "Constraints" and "Bounds". There is also a "Binaries" section for *.ilp files and an "Initials" section for *.inp files where both files must have the last statement as “End”. In-line comments are indicated by the “\” backslash character and the usual IMPL comment character “!” (exclamation mark).

`*.stp`

Stop file to abort, interrupt or preempt both the logistics (MIP) problem’s branch-and-bound (B&B) enumerative search and the quality (NLP) problem’s successive linear / quadratic programming iterative search (SLPQPE and IPOPT). If a one (1) is found in line or row one (1) and character or column one (1) of the stop file, then the solver’s search will be signaled to terminate via its callback functionality; any other number or character will be ignored and treated as zero (0). The stop file is useful when one or many good locally logistics-feasible (integer-feasible) solutions have been found during the B&B search for example and it is required to properly, systematically or gracefully terminate in the same way as finding a globally optimal (provably optimal) solution or locally converged solution does. The stop file is also useful when a quality solution is very close to being converged in IMPL’s SLPQPE and IPOPT but the user, modeler or analyst manually decides to terminate prematurely without losing the pre-converged solution results.

Please note that the stop file is automatically reset or reinitialized by the IMPL Console to include a zero (0) and if the stop file does not exist then the IMPL Console will be created with a zero (0) in line / row one (1) and character / column one (1) which is an indication or signal not to stop IMPL. Also note that Control-C / CRTL-C keystroke events or interrupts are trapped via a signal handler system routine called or invoked by the IMPL Presolver which will automatically write a one (1) into the stop file when the IMPL Presolver detects a Control-C / CRTL-C keystroke event / interrupt and a “STAYED” log message will be sent to the log file or the console window. If the stop file is not found or does not exist and the Control-C / CRTL-C keystroke event is detected, then the IMPL Presolver will immediately stop, exit or terminate.

`*.fms`

The optional formulasfile flag file to import, input, read or load a text or ASCII (UTF-8) flat file containing the calc-scalar formulas or expressions where the *.fms is an optional file type (i.e., the formulasfile flag does not assume the *.fms extension) and each row or line of the *.fms file configures a single calc-scalar via its right-hand-side value expression or formula. The format for the formulas files may be comma-separated-value (CSV) or equal-sign-separated-value (ESV / =SV) e.g., **formula_name, formula_expression** or **formula_name = formula_expression**.

`*.fts`

The optional footingsfile flag file to import, input, read or load a text or ASCII (UTF-8) flat file containing the calc-scalar formulas or expressions where the *.fts is an optional file type (i.e., the footingsfile flag does not assume the *.fts extension) and each row or line of the *.fts file configures multiple calc-scalars in a table, grid, block, matrix or 2D-array format. The very first non-comment line or row must contain the header which separates each calc-scalar name with IMPL's standard comma delimiter.

`*.fgs`

The optional flagsfile flag file to import, input, read or load a text or ASCII (UTF-8) flat file containing the IMPL console flags where the *.fgs is an optional file type (i.e., the flagsfile flag does not assume the *.fgs extension) and each row or line of the *.fgs file configures a single console flag. The format for the formulas files may be comma-separated-value (CSV) or equal-sign-separated-value (ESV / =SV) e.g., **flag_name, flag_value** or **flag_name = flag_value**.

`*.pid & *.spo`

The *.pid (PSV) and *.spo CSV (pipe- and comma-separated-value) formatted files are suggested file types or extensions only and may be inputted, loaded or imported with the include frame's attribute DATAFRAME to read PID retuning related data required for the routines SISOARX() and RETUNEPID() as well as SIMULATEPID() i.e., identification, estimation (estimization), optimization and simulation (simulization) of the process model and the PID settings respectively. The "pid" data-vector found inside *.pid must be consistent with the PID data-vector argument required by the RETUNEPID() and SIMULATEPID() datadata functions where each field may itself be a calc-scalar expression or formula. The *.spo stands for setpoint variable value ("sp"), process output variable value ("pv") and controller output / process input variable value ("op") related to single-loop proportional, integral (re-set) and derivative (pre-act) PID controllers where the sp,pv,op tuple of data-sets, -lists or -vectors, with lower case names only, may be arranged in any order, permutation or sequence and other data-sets, -lists or -vectors may also be present in the *.spo file such as an integer or real number for the time-stamp, time-step, time-interval, time-index, time-period (ts, ti, tp), etc.

An example *.pid file in IMPL's pipe-separated-value (PSV) format is provided below with symbols and nomenclature description for more clarity on its configuration.

```
pid, Kp     | KpL    | KpLs   | KpU    | KpUs   |
   , Ti     | TiL    | TiLs   | TiU    | TiUs   |
   , Td     | TdL    | TdLs   | TdU    | TdUs   |

   , Ts     |
   , PIDeq  |
   , opL    | opU    |
   , pvL    | pvU    |

   , Unt    | Unl    |

   , mL     | mU     |
   , nL     | nU     |
   , kL     | kU     |
```

The `Kp`, `KpL`, `KpLs`, `KpU` and `KpUs` are the proportional gain, proportional gain lower bound plus step-size and proportional gain upper bound plus step-size where `Kp` acts as the center-point or centroid for the brute-force grid (or exhaustive) search inside the RETUNREPID() datadata function i.e., `KpL <= Kp <= KpU`. Similarly, `Ti`, etc. and `Td`, etc. are the integral-time and derivative-time with lower, upper and steps / strides. Please note that if `KpU < Kp, TiU < Ti` or `TdU < Td`, then the upper bound range intervals are not iterated over in the PID retuning brute-force grid search and the upper bounds for the lower bound range intervals are `KpU`, `TiU` and `TdU` and not `Kp`, `Ti` and `Td`. The `Ts` represents the sampling-interval / -period / -rate or scan-interval / -period / -rate time and must be in the same time unit-of-measure as `Ti` and `Td`. The `PIDeq` currently supports the three well-known PID equation types, forms or modes A = 0, B = 1 and C = 2 as described further below. The `opL`, `opU`, `pvL` and `pvU` are the lower and upper bounds for the process input or controller output (op) and process output (pv). The `Unt` and `Unl` are the upper norm type (1, 2 or 3 for infinity) and the upper norm limit required by RETUNEPID() to constrain the controller output, manipulated variable or process input minus its previous or immediate past value (i.e., input-move-error) variance for **self-regulating processes** (stable) whereby the setpoint (reference or target) minus its process output (i.e., output-error) is minimized. And if negative (-ve), then the output-error variance is constrained instead for **integrating processes** (marginally stable) whereby the input-move-error variance is minimized known as "level-flow smoothing" (LFS)) or "averaging level control" (ALC) and typically employed for buffer or surge vessel "loose" versus "tight" level or volume control loops – see also the NORMOE() and NORMIME() datacalc functions. It should also be mentioned that even for self-regulating (or non-integrating) processes, the upper norm limit (`Unl`) may also be negative (-ve) whereby the input-move-error variance norm is minimized and is constrained or subject to the output-error norm upper limit implying "loose" versus "tight" control performance for the self-regulating / non-integrating process under PID feedback control. It should also be noted that retuning **runaway** processes (unstable) is technically not addressed but of course may still be applied though bearing in mind that for proper control of unstable processes always requires some form of derivative, anticipatory or future-related action.

The `mL`, `mU`, `nL`, `nU`, `kL` and `kU` are the lower and upper degree bounds for the brute-force grid (or exhaustive) search performed with the SISOARX() datadata function in order to determine the best numerator (m), denominator (n) and dead-time / time-delay (k) ARX (Auto-Regressive eXogenous) linear parametric dynamic (transfer function) model for the process from closed- and/or open-loop operating data found in the *.spo file. After the SISOARX() datadata function has completed, the dead-time or time-delay of the process (`Ttd`) is identified as the `k` number of time-periods or sampling-intervals with the least, minimum or smallest sum-of-squares of residuals or prediction errors and the the "effective" process time-constant (`eTtc`), assuming a first-order plus dead-time (FOPDT) response, may also be evaluated where the dynamic process model's steady-state gain (`Gss`) may also be estimated. *Interestingly, even though the single-input and single-output linear dynamic process model is most likely structurally over-parameterized (i.e., estimate, fit or learn more parameters than necessary), the phenomenon of self-regularization helps to inherently prevent over-fitting. That is, redundant parameters tend to converge to zero (0.0) as more data becomes available and the remaining parameters tend to converge to their true low-order system parameter or coefficient values without the requirement for explicit parameter regularization via 1- (absolute, Manhattan) or 2- (squared, Euclidean) norm penalty-errors or -elastic (artificial) variables cf. Du, Liu, Weitze and Ozay, "Sample complexity analysis and self-regularization in identification of over-parameterized ARX models", IEEE 61st Conference on Decision and Control (CDC), 2022.*

In terms of the number of PID retuning brute-force grid (or exhaustive) search simulations required to be performed by RETUNEPID(), we can easily calculate this assuming all steps or strides (`KpLs`, etc.) are positive (+ve) and non-zero as:

```
Number of Simulations = (nKpl + nKpU) * (nTil + nTiU) * (nTdl + nTdU)
```

where

```
nKpL = INT((Kp - KpL) / KpLs) + 1,
nKpU = INT((KpU - Kp) / KpUs) + 1,
nTiL = INT((Ti - TiL) / TiLs) + 1,
nTiU = INT((TiU - Ti) / TiUs) + 1,
nTdL = INT((Td - TdL) / TdLs) + 1, and
nTdU = INT((TdU - Td) / TdUs) + 1.
```

In order to elucidate some guidance and clarity on how to configure the brute-force grid (or exhaustive) search for the PID retuning, suggested values for `KpL`, `KpU`, `TiL`, `TiU` and `TdL`, `TdU` are provided below based on the default or existing PID parameter settings multiplied by a user, modeler or analyst supplied factor or multiplier (e.g., `2.0`):

```
KpL =       0.0,
KpU = 2.0 * Kp,
TiL =        Ts,
TiU = 2.0 * Ti,
TdL =       0.0, and
TdU = 2.0 * Td.
```

Suggested step or stride values for `KpLs`, `KpUs`, `TiLs`, `TiUs` and `TdLs`, `TdUs` may be also chosen using a user, modeler or analyst supplied factor or multiplier (e.g., `0.1 = 10^-1`) which is directly related to the inverse of the suggested number of steps, strides, increments or discrete elements within its lower and and/or upper bound sub-ranges or -domains (e.g., `10`):

```
KpLs = 0.1 * (Kp - KpL),
KpUs = 0.1 * (KpU - Kp),
TiLs = 0.1 * (Ti - TiL),
TiUs = 0.1 * (TiU - Ti),
TdLs = 0.1 * (Td - TdL), and
TdUs = 0.1 * (TdU - Td).
```

Therefore, the theoretical number of PID retuning brute-force grid (or exhaustive) search simulations performed by RETUNEPID() is proportional to the given factors or mulitpliers discussed above.

And for completeness, we provide the three (3) most popular PID equations A, B (a.k.a. PI-D) and C (a.k.a. I-PD) as implemented in IMPL below in **velocity-form** where `t` and `t-1` indicate the current / present and the immediate past / previous values respectively:

```
PIDeq = 0 (A)
op,t = op,t-1 + Kp*       ((sp,t - pv,t) - (sp,t-1 - pv,t-1)) +
                Kp*Ts/Ti* (sp,t - pv,t) +
                Kp*Td/Ts*((sp,t - pv,t) -
                          2*(sp,t-1 - pv,t-1) + (sp,t-2 - pv,t-2))

PIDeq = 1 (B)
op,t = op,t-1 + Kp*       ((sp,t - pv,t) - (sp,t-1 - pv,t-1)) +
                Kp*Ts/Ti* (sp,t - pv,t) -
                Kp*Td/Ts* (pv,t - 2*pv,t-1 + pv,t-2)


PIDeq = 2 (C)
op,t = op,t-1 - Kp*       (pv,t - pv,t-1) +
                Kp*Ts/Ti*(sp,t - pv,t)    -
                Kp*Td/Ts*(pv,t - 2*pv,t-1 + pv,t-2)
```

Please note that interestingly, PID equations B and C can also be re-formulated simply as zeroing-out the setpoint, target or reference signal in their respective proportional and derivative terms i.e., 0 * sp,t, etc.

Finally, it should be mentioned that the sign of the controller gain `Kp` may be either positive (+ve) or negative (-ve) depending on the sign of the process gain where it is important to understand that the product of the process gain times the controller gain must its be positive (-ve). *Or stated more simply, the sign of the process gain and the sign of the controller gain must be the same i.e., if the process gain is negative (-ve), then the controller gain (`Kp`) must also be negative (-ve).* This is also related to whether the actuator or final control element is direct- or reverse-acting (DIR / REV) whereby a direct-acting controller increases its output (OP) as the process variable (PV) increases (i.e., they move in the same direction) and a reverse-acting controller increases its output (OP) as the process variable (PV) decreases (i.e., they move in opposite directions).

IMPL Output Files

`IMPL.hdr`

The header file is written by the IMPL.exe console program, only if the IMPL setting WRITEHEADERFILE equals one (1) where zero (0) is is default, and contains all of the IMPL Server constants and its routine prototypes or signatures to call IMPL from other computer programming or scripting languages. The header file is output in the same directory as the location of IMPL.exe; see also the IMPLwriteheader() routine. The IMPL.hdr file should be reviewed and referred to when interacting with IMPL inside computer programming and scripting languages and should be used as the basis for their "modules". The IMPL.hdr file also contains the IMPL.set setting and IMPL.mem memory enumerated constants as well as the modeling signals and solver statuses, etc. **Please note that if in the rare chance, occurrence or occassion that any of the header information has an identifier changed or modified, as opposed to being added to in terms of a new identifier, the IMPL release notes will properly document these updates when they arise.**

`*.eml (*.elpet, *.elp, *.enpet and *.enp)`

Output file containing the obscured or encrypted IML file when the fob flag number is negative (-ve, < 0) and created or generated by IMPL's Interfacer. At the top of the IML file only a security frame i.e., &iSecurity will be placed with the positive fob number (+ve, > 0) in the *.eml file only. This file can be copied, renamed with an IML extension or type and the security frame (cipher data) removed and then can be used with IMPL and the positive (+ve) fob number to obscure, obfuscate, encrypt or encode your model and data.

`*.tml`

Output file containing the traced, echoed or shadowed IML file when the setting WRITETRACEIMLFILE = 1 (one) or two (2) where its default is zero (0) to not trace include files. This file contains all of the lines read in from the IML file and is especially helpful when there are several include file frames and #if directives which are useful for the user, modeler or analyst to view all of the read / inputted / imported / loaded problem data in one file. The *.tml file can be renamed to an *.iml and imported back into IMPL provided the include file frames have been either commented out or removed as there will be much duplication i.e., both the include file will be imported and the traced, echoed or shadowed content. However if WRITETRACEIMLFILE = 2, then these include-file frame leader and trailer features, rows or lines are automatically commented out. And, if the WRITETRACEIMLFILE setting is set to one (1) or two (2) in a settings frame located anywhere in the IML file, then the *.tml file will only trace the input IML frames and features from that point onwards.

`*.io`

Input and output file containing the inputs and outputs for any problem construncted using UOPSS-QLQP© / UQF©.

`*.oi`

Output and input file containing the outputs to inputs for any problem constructed using UOPSS-QLQP© / UQF© and similar but the reverse so to speak to the *.io file.

The *.io and *.oi files are helpful to confirm the UOPSS flowsheet or flow diagram especially if they are generated by a drawing software such as Dia as they feedback the configured / constructed unit-operations (UO), unit-operation in-/ out-ports-states (UOPS) and the tee- / external-streams (UOPSUOPS, UOPSPSUO or UOPS2) also referred to as connections, paths, routes or transfers.

`*.iod`

Input and output data or industrial output/object data file containing the model (static, master) and cycle (dynamic, transactional) UOPSS-QLQP© / UQF© data for each problem using the OML mnemonic codes. Currently, only quantity and quality data are written out. This file is relatively self-explanatory in terms of viewing and relating to the model and cycle data as it is simply a reformatting of the problem data in hopefully a somewhat human readable form. For clarity however, the symbol “>-“ means in-port-state and “->” means out-port-state. Unit-operation data is written first followed by its in-port-states, then its out-port-states and last each downstream in-port-state connected to the upstream out-port-state is written out i.e., the external- or tee-stream data.

This *.iod file is especially useful when configuring quality optimization problems in terms of providing configuration feedback when unit-operation-port-state aliases and density, component and property templates are employed to conveniently bulk / en bloc configure the assignment, association or attachment of diverse qualities to different UOPS's as all UOPS's usually do not have all qualities i.e., sparse versus dense UOPS-Q configuration if you will.

`*.exl | *_nnn.exl`

Export files with the same name as the IML file where selected variable result values are written, outputted, printed or exported in various EXL frames similar to the IML frames. The EXL files can be read / imported / inputted to the IALViewerQL.py and IALViewerQQ.py. These EXL files may also be easily imported into spreadsheet and database software as they are essentially CSV files with leader and trailer features similar to the IML files. Please note that during the write / output / print / export, the file type or extension is *.EXLX / *.exlx with an appended or suffixed "X" / "x". Then upon completion, the file is renamed with no suffixed "X" / "x". The EXL files are intended to provide a mostly complete CSV frame and formatted export of virtually all of the variable solution results or reponses known to IMPL with the execption of a few internal variables. The output of *all* variables may be found in the *.dtv dump files and the output of *all* constraints are found in the *.dtc dump files.

`*.bdt`

Binary unformatted files storing the independent sets, catalogs, lists, parameters and formulas; this file is written by the IMPL Console via IMPL Server's render() routine. These unformatted binary files are useful when re-running IMPL without having to re-read the fact.iml / subject.iml file. The IMPL Server's restore() routine can be used to read this binary file to populate the resource-entities appropriately without reading the *.iml file. The rendered or serialized *.bdt file is not human readable in contrast to other well-known serialization file formats such as CSV, XML, YAML and JSON.

For example, it is possible to configure an IML file which contains only model or master-data (static) and one or more IML files which contain only cycle or transactional-data (i.e., only dynamic content and command data). The IMPL Server render() routine can export or output the *.bdt file after the model data has been received into IMPL via IMPL's Interfacer which can be imported or inputted back into IMPL at any time using the IMPL Server's refresh() and restore() routines. Then, each cycle data IML file can be inputted or received sequentially or successively into IMPL in order to model and solve different cycle data user, modeler or analyst cases, scenarios, situations, samples, etc. Since the *.bdt file is an unformatted binary file and all calculations are computed into literal numbers, the model data will be inputted significantly faster than from the IML file.

Please be cognizant that for problem solutions containing foreign-models (i.e., with ILP / INP foreign-files), the incidentally and/or intentionally occurring logistics-feasible solution results pertaining to both the non-foreign- and foreign-constraint residuals or responses are not updated as the IMPL Modeler is not called or invoked internally in the solvers' respective integer-feasible callback routines during the MIP branch-and-bound (B&B) search. However after the termination of the MIP, the final *.exl, *.bdt, etc. results are properly updated as expected.

`*_nnn.bdt`

Binary unformatted files suffixed by a logistics- / integer-feasible solution number "_nnn" with the same name as the IML file where all variable results for a single integer-feasible or logistics-feasible solution found in the MIP solvers are written in unformatted binary form; see also the *_nnn.exl files.

`*.ldt` (`USELOGFILE = 1 or 2`)

IMPL log files with the same name as the IML file where IMPL's output messages are displayed to monitor, steward and track the progress of IMPL when `USELOGFILE = 1 or 2`, else all log message output will be sent to the IMPL Console window or terminal. Please note that all messages written to this log file are appended to it so that multiple IMPL runs or executions will be logged accordingly in the same *.ldt file if USELOGFILE = 1 else if USELOGFILE = 2, then the log output is not appended.

`*.rdt` (see IMPLshowreport)

Report files with the same name as the IML file where IMPL resource-entity details are displayed. This report file summarizes IMPL's proprietary data memory structure and storage metrics described in the SSIIMPLE Infrastructure Manual.

`*.sdt` (see IMPLshowsummary)

Summary files with the same name as the IML file where IMPL modeling and presolving details are displayed such as the number of continuous, discrete, linear and nonlinear variables, linear and nonlinear constraints and first-order partial derivatives included or excluded from the problem i.e., before and after IMPL's presolve and referred to as the original and the organized / optimizable problems respectively.

`*.tdt` (see IMPLshowstatics)
Statics files with the same name as the IML file where if any IMPL presolved constraints are found to be infeasible or inconsistent (i.e., violated greater than the CONTOL closure tolerance setting) at a solution, are displayed. "Static" constraints are IMPL's term for constraints that are presolved, eliminated, excluded or removed from the problem by IMPL before the solvers (with their own presolving) are called where "non-static" constraints in this context mean constraints that are known and variable in / to the solver. The IMPL Presolver may or may not "prempt" the presolving if an infeasible or inconsistent constraint is found. If any static constraints are found to violate its constraint closure tolerance, then IMPL declares this problem to be infeasible / inconsistent and an error results. Obviously, if any static constraints are not feasible at a solver solution, then the original problem is infeasible even though the organized / optimizable problem known to the solver is feasible.

`*.udt` (see IMPLshowspoilings)
Spoilings, also referred to as stragglers or shortings, are simply the worst "non-static" constraint residual, response or result violations for the organized / optimizable problem's model compared against the IMPL setting or option CONTOL (short for constraint tolerance) and written, printed, exported or outputted with the original model's indice or index numbers. "Non-static" constraints are IMPL's term for constraints that are not presolved, not eliminated, included or not removed from the problem by IMPL before the solvers (with their own presolving) are called where "static" constraints in this context mean constraints that are not known and either fixed or floating/following in / to the solver. The user, modeler or analyst can view all of the equality and inequality constraint imbalances after the problem has been modeled and solved whether or nor it is feasible, converged, infeasible or unconverged.

`*.jdt` (see IMPLshowderivative)
Sensitivity or first-order partial derivatives files with the same name as the IML file (fact.jdt / subject.jdt) where the sparse Jacobian matrix elements, parameters or first-order derivative coefficients are displayed. The right-hand-side constraint constants or base, balance or bias coefficients are not provided in these *.jdt files. The *.jdt files essentially output the first-order Taylor series expansion of

the problem's model / matrix without the constant base, balance or bias. The *.jdt files are helpful to perform what we call "configure-and-check" and/or "code-and-check" to validate and verifiy the problem's model (and matrix) details.

IMPL uses the nomenclature of "=", "=<", "=>" and "~", "~<", "~>" for linear and nonlinear constraint senses / types respectively. The constraints in the *.jdt file should be read or interpreted as constraint name, constraint sense / type and constraint terms (i.e., parameter constants, mathematical operators and variables). However, the *.jdt file is less readable as the constraints and variables are shown in their orginal (i.e., unorganized / unoptimizable) number indexes / indices only. Constraints are enclosed by "<" and ">" and variables are enclosed by **a)** "[" and "]" if the derivative is "derived" by the complex-step (or finite-difference) method or "defined" by the user, modeler or analyst, **b)** "(" and ")" if the derivative is "declared" explicitly as it is linear or it is "detected" to be zero (0.0), **c)** "{" and "}" if none of the above and finally **d)** by "|" and "|" if the variable is presolved out by IMPL's presolver before being transferred to the third-party solvers.

`*.ddt` (see IMPLshowderivative)
Sparsity-pattern files with the same name as the IML file are a human readable form of the linear and nonlinear constraint meta-data or sparsity-pattern written for every variable and constraint known to the problem *prior to / before* any of the IMPL (primal) presolving (cf. the separability() and shrinkability() routines) has been performed i.e., shows the complete and original problem matrix sparsity-pattern. The *.ddt file is especially useful when modeling (i.e., building mathematical programs) with user, custom, bespoke or adhoc variables and constraints in IML, foreign-models with ILP / INP and IMPC as it displays all of the variables coded or contained in all constraints *before or prior to* the problem matrix is presolved by IMPL i.e., the original or unorganized / unoptimizable problem model / matrix. The *.ddt files are helpful to perform what we call "configure-and-check" and/or "code-and-check" to validate and verifiy the problem's model (and matrix) details.

`*.ndt` (see IMPLshowexpression)
Symbolic or mnemonic files with the same name as the IML file are a human readable form of both the linear and nonlinear constraints written for every variable and constraint known to the problem typically *after* the IMPL (primal) presolve has been performed**.** The *.ndt files are helpful to perform what we

call “configure-and-check” and/or “code-and-check” to validate and verifiy the problem’s model (and matrix) details.

IMPL uses the nomenclature of “=”, “<=”, “>=” and “~”, “<~”, “>~” for linear and nonlinear constraint senses / types respectively. The constraints in the *.ndt file should be read or interpreted as constraint name, constraint sense / type and constraint terms (i.e., parameter constants, mathematical operators and variables). For example, using an upper yield inequality on an out-port of a process unit-operation and its flow where the yield is set to one (1.0), a constraint instance such as “c2r_xjyupper(,) <= 1 v3r_xjif(, ,) -1 v2r_xmf(,)” is interpreted as “c2r_xjyupper(,): 1 v3r_xjif(, ,) -1 v2r_xmf(,) <= 0” or “c2r_xjyupper(,): 1 v3r_xjif(, ,) <= 1 v2r_xmf(,)”. It should be noted that the format of the *.ndt file assumes that all constraints have a right-hand-side of zero (0.0) i.e., = 0, <= 0 and >= 0.

As mentioned, the *.ndt file is only available internally to Industrial Algorithms Limited (IAL) for debugging and troubleshooting support and externally if a development license of IMPL is purchased.

`*.qdt` (see IMPLshowscalings)
Scalings are also easily calculated by finding the largest absolute magnitude of any first-order derivative coefficient in any linear or nonlinear constraint (row) and similarly for the variables (columns). This is identical to the “equilibration scaling” method found in linear programming scaling. Constraints or variables with the worst scalings (sorted worst first) are an indication that they have large absolute coefficient magnitudes in the Jacobian matrix and re-scaling of their variables or columns should be considered.

Of note, these scalings are not scaling multipliers as scaling multipliers are the inverse or reciprocal of these. It should also be noted that if there are no convergence issues observed when solving nonlinear (quality) optimization problems, then it is less important to screen and scrutinize the *.kdt and *.qdt files for large slackings and scalings respectively.

`*.kdt` (see IMPLshowslackings)
Slackings are easily calculated by substituting into the constraints the upper and lower variable bounds and taking the difference i.e., all upper bounds in the constraint and then all lower bounds. The slackings are sorted in descending order with the worst first and only nonlinear constraints are

considered. Large slackings are an indication of excessive variable domain degree-of-freedom and/or scaling issues and may cause excessive iterations during the NLP solving process or even non-convergence and apparent infeasibilities. The worst slackings should be screened and scrutinized in terms of their included variables as well as referencing the scalings *.qdt file.

`*.gdt` (see IMPLshowspannings)
Spannings are also easily calculated by determining if the variable value, result or response is at or very near to its lower bound (cf. EPSIL) as indicated by "L" in the spannings file and if the variable is at or very near to its upper bound indicated with "U". In LP terminology, a variable at or near its lower / upper bound is "non-basic" or also termed an active / working variable and a variable between or not at or near its lower / upper is declared as "basic".

`*.dtr` (see IMPLshowseriesset)
Series-set (or range-set) dump files with the same name as the IML file where all SERIES-SET resource-entity roster-enumerators are output.

`*.dts` (see IMPLsimpleset)
Simple-set dump files with the same name as the IML file where all SIMPLE-SET resource-entity roster-enumerators are output with their reference-events / record-entries and row-elements.

`*.dty` (see IMPLsymbolset)
Symbol-set dump files with the same name as the IML file where all SYMBOL-SET resource-entity roster-enumerators are output.

`*.dtg` (see IMPLshowcatalog)
Catalog dump files with the same name as the IML file where all CATALOG resource-entity roster-enumerators are output.

`*.dtl` (see IMPLshowlist)
List dump files with the same name as the IML file where all LIST resource-entity roster-enumerators are output.

`*.dtp` (see IMPLshowparameter)
Parameter dump files with the same name as the IML file where all PARAMETER resource-entity roster-enumerators are output.

`*.dtv` (see IMPLshowvariable)
Variable dump files with the same name as the IML file where all VARIABLE resource-entity roster-enumerators are output. The second last integer number column is the variable's row-element number before IMPL's presolve ("original" index or indice) and the last integer number column is the variable's row-element number after IMPL's presolve ("optimizable" or "organized" index or indice). If the second last column's index / indice is negative (-ve), then it reports the included variable index / indice in the original problem or sub-problem that is the feeder, leader or primary floater / follower variable when the IMPL settings REMOVEFLOATERS equals one (1). *It should be noted that for properties that are transformed or converted to a blending index or number (see property-transform), the *.dtv file will show the transformed variable result and not the untransformed value.*

`*.dto` (see IMPLshowvariable)
Variable dual variable, reduced cost, opportunity cost, marginal-value or Lagrange-multiplier dump file for the last or final LP / QP iterated solution found during IMPL's SLPQPE solver and the IPOPT solver or when an LP or QP is solved i.e., for quantity and quality optimizations only. Instead of containing the variable activity, result, response or value as in the *.dtv file, the *.dto file has the variable reduced cost, opportunity cost or Lagrange-multipler values also known as the dual variable values (cf. DUALVALUE$).

`*.dtw` (see IMPLshowvariable)
"Vetistics" (vetting) or statistics dump files with the same name as the IML file where all VARIABLE and CONSTRAINT resource-entity roster-enumerators are output when solvers suffixed with `_sorve` are called for observability, redundancy, variance, etc. This file outputs in the following order: the variable value ("value"), observability and redundancy metrics ("viability" / "validation"), variances ("variance", square-root equals the standard-deviation), "maximum power" gross error detection statistics ("vetistic" / "veracity", reconciled revision or residual divided by its standard-deviation), nominal 95% confidence-intervals ("variability1" and "variability2" / "valuation1" and "valuation2") and violations ("violation") from their lower and upper hard bounds set to one (1.0) and two (2.0) respectively.

`*.dtc` (see IMPLshowconstraint)

Constraint dump files with the same name as the IML file where all CONSTRAINT resource-entity roster-enumerators are output. The values shown in this file are the constraint residuals for all constraints whether the problem is optimal, feasible, converged or unconverged. The second last integer number column is the constraint's row-element number before IMPL's presolve ("original" index or indice) and the last integer number column is the constraint's row-element number after IMPL's presolve ("optimizable" / "organized" index or indice). If the second last column's index / indice is negative (-ve) and greater than or equal to the number of original constraints, then it reports the excluded constraint index / indice in the original problem or sub-problem that is deleted, dropped or removed when the IMPL settings REMOVEFLOATERS equals one (1). And, if the second last column's index / indice is negative (-ve) and less than the number of original constraints, then it reports or records the excluded original constraint index or indice that has zero (0) or no variable inclusion or involvement in the optimizable / organized problem or sub-problem as these are trivial, redundant or vacuous constraints and are deleted, dropped or removed accordingly in the IMPL separability() and shrinkability() routines.

Further description of this constraint residual or results file is required. The real or floating-point numbers (i.e., 0.000000D+00) are the constraint residual values at the conclusion of the solving whether converged or unconverged. For equality constraints (c.f., "=" and "~") their absolute values should be less than the convergence tolerance of the LP, QP, MIP or NLP solver and for inequality constraints they should be either negative (-ve) for "<=" and "<~" and positive (+ve) for ">=" and ">~" constraints. At the moment, IMPL standard or non-foreign modeling only creates equality and "less than or equal to" ("<=" and "<~") inequality constraints as "greater than or equal to" (">=" and ">~") constraints can always be converted to "less than or equal to" constraints by multiplying by minus one (-1.0). However, >= or "greater than or equal to" constraints are supported in ILP and INP foreign-files as well as the user, modeler or analyst adhoc, bespoke or custom linear, logic, logical or logistics constraints. If the constraint residuals violate their convergence tolerances, then they are infeasible, inconsistent, unsatisfied or unconverged. These constraint convergence tolerances are defined by the LP, QP, MILP, MIQP and NLP solvers. Typically for LP's, QP's, MILP's and MIQP's, the constraint (or zero) tolerances are 1D-6 and for IMPL's SLPQPE its default constraint convergence tolerance is 1D-6 (cf. CONCONVTOL) but is usually slightly greater than this value depending on the problem or sub-problem. Unless the problem is infeasible, constraint convergence is the primary issue for nonliner solvers as finding or searching for an optimized feasible solution due to nonlinerities and non-convexities can be difficult and hence the reason NLP convergence tolerance is considered as an adjustable or tuneable hyperparameter, setting or option of the solver, and more often than not, is specific to a given problem or sub-problem whereas the constraint (or zero) tolerance (1D-6) is never adjusted or altered and for LP, QP and MIP is a defacto standard.

`*.dtd` (see IMPLshowconstraint)
Constraint dual variable, shadow price, marginal-value or Lagrange-multiplier dump file for the last or final LP / QP iterated solution found during IMPL's SLPQPE solver and the IPOPT solver or when an LP or QP is solved i.e., for quantity and quality optimizations only. Instead of containing the constraint activity, residual or value as in the *.dtc file, the *.dtd file has the constraint shadow price, marginal-value or Lagrange-multipler values also known as the dual variable value (cf. DUALVALUE$).

The dual variable solution may be used to infer what are known as the non-equilibrium transfer-prices for interface, interacting, interconnected, interchange or intermediate streams and are what IMPL calls the "worth" of a co-feed, co- / by-intermediate and co- / by-product stream.

`*.dtf` (see IMPLshowformula)
Formula dump files with the same name as the IML file where all FORMULA resource-entity roster-enumerators are output in a tokenized format.

`IMPL-QLQ.imp, IMPL-QLQ.imv, IMPL-QLQ.imc`
Model description files containing the *data-related* parameter (sets, catalog, list, parameter and formula) and the *model-related* variable and constraint names (notation and nomenclature) respectively with short descriptions.

SLPQPE Output Log

The console log for IMPL's Successive Linear / Quadratic Programming Engine (SLPQPE) connected to any LP or QP solver is described where its output solver log header is detailed below.

```
itr# lp?    objF   #uncX  nrmX   #uncF  nrmF     maxF#  #uncE  nrmE    maxE#    objT
```

itr# = iteration count where "b" = barrier, "p" = primal simplex, "d" = dual simplex is suffixed to indicate the LP or QP method or algorithm.

lp? (qp?) = lp (qp) status (optimal, infeasible, etc.) using the exact return codes from each third-party (sub-)solver where the suffixed "w" = warm-start may be present if the warm start setting or option is selected.

objF = objective function value without the contributions from any augmented error / elastic / excess / artificial / penalty variables and represents the problem's or sub-problem's objective function value.

Please note that if the objective function (objF) is positive (+ve), then SLPQPE is minimizing and if the objF is negative (-ve), then SLPQPE is maximizing. The default optimization sense or direction for SLPQPE is to minimize objF rather than maximize it and is similar to most nonlinear solvers.

#uncX = counter for the number of unconverged nonlinear variables where the suffix "s" = step-bounding applied to nonlinear variables only after the error variables have converged.

nrmX = 2-norm (squared) convergence for nonlinear variables only.

#uncF = counter for the number of unconverged nonlinear constraints as linear constraints always converge if the LP / QP is feasible.

nrmF = 2-norm convergence for nonlinear constraints only and if not near-zero after several iterations, is a reliable "leading indicator of infeasibility". ** Note that the average worst or largest nonlinear constraint residual = SQRT(nrmF / #uncF).*

maxF# = constraint index for maximum nonlinear constraint residual violation **(**after IMPL's presolve i.e., the organized / optimizable problem's model).

* Note that for the initial or zeroth (0th) iteration, this largest presolved constraint index reported may be used to assess the reasonableness of the initial-values, starting-guesses or default-results.

#uncE = counter for the number of unconverged nonlinear constraint error variables.

nrmE = 2-norm (squared) convergence for nonlinear constraint error variables where "r" = error weights revised where nrmE is used as a reliable *"leading indicator of infeasibility" (LII) or "Infeasibility Leading Indicator" (ILI)* given that if non-zero error variable results or responses exist, then there are infeasibilities in the nonlinear part of the problem. ** Note that the average worst or largest error variable and its corresponding average worst / largest nonlinear constraint residual = SQRT(nrmE / #uncE).*

maxE# = constraint index for maximum nonlinear constraint error variable violation (after IMPL's presolve) where if #uncE equals zero (0) then maxE# will also equal zero (0).

objT = total objective function value including the contributions from all augmented error / elastic / excess /artificial / penalty variables and computed with delta / difference and step-bounded (nonlinear) variables. The objT should eventually approach near-zero (~0.0) upon convergence or converge to some constant non-zero value usually if infeasible or inconsistent.

Interestingly objT can be considered as a *"leading indicator of convergence" (LIC) or "convergence leading indicator" (LCI)* especially when objT converges before nrmF as indicated by an appended "*" once objT has converged. However, if objT is persistently greater than the specified convergence tolerance for the constraints, then this may be a sign of some infeasibility / inconsistency in the problem or sub-problem.

The first row, line or feature of IMPL's SLPQPE log output before the above header, details the uncovergence for the zeroth (0th), starting, beginning or initial iteration prior to the first (1st) LP or QP sub-solve. This feature states the number of nonlinear variables with convergence control after IMPL's presolve or shrinkability() routine(s) (**`#uncX`**). The number of linear variables which have constant first-order partial derivatives in all of the constraints are not reported. Depending on the form flag or structure signal i.e., if sparsic or symbolic, the output log will display a different number of nonlinear variables; that is more for symbolic structure. Only for the sparsic form or structure are linear variables detected via IMPL's stationarity() routine as variables with at least one (1) non-constant 1st-order derivative coefficient are declared or defined as nonlinear requiring convergence control overhead (step-bounding).

It should be stated that IMPL's SLPQPE is a straightforward yet powerful numerical algorithm to solve nonlinear problems of industrial scale, size, scope and sophistication using any third-party LP and QP solver and may be seen as a more aggregressive optimization solver than SQP solvers such as IPOPT, CONOPT, KNITRO or SNOPT for example. Any non-zero SLPQPE penalty-error / elastic / excess variable, which are automatically augmented and applied to the nonlinear constraints only, can be a useful detector and indicator of one or more potential infeasibilities, inconsistencies, defects, faults or gross errors in the problem / sub-problem data. When the nonlinear problem cannot converge on the error, elastic or excess variables, then SLPQPE will report the worst or largest organized / optimizable problem's constraint index or indice number which is converted to the original problem's constraint number via IMPL's Presolver to aid in the troubleshooting / debugging / diagnosis of the problem. However, care must be exercised as apparent infeasibilities due to poor starting-guesses, initial-values or default-results and/or non-convexities may exist. If severe numerical errors are detected due to divide-by-zero, etc. resulting a NaN or Infinity, then SLPQPE identifies these bad or numerically unstable constraints via the log message **`badF#`** where the IMPL presolved constraint index or indice number of the organized / optimizable problem or sub-problem is displayed along with the bad constraint's residual, response or result real value.

That said, the section further below elaborates on how to effectively manage and handle infeasibilities with IMPL. In addition, please see the return statuses from IMPL's Presolver and SLPQPE which will return the original or unpresolved nonlinear constraint index or indice number of the worst or largest unconverged SLP error variable if the error / elastic / excess variables do not converge below their tolerance. This diagnostic can be helpful to locate and isolate infeasibilities / inconsistencies in the problem or sub-problem.

SECQPE Output Log

The console log for IMPL's Successive Equality-Constrained Quadratic Programming Engine (SECQPE) is described where its output solver log header is detailed below.

```
itr#          objF       nrmX       nrmF   maxF#
```

itr# = iteration count of the algorithm.

objF = quadratic objective function value which is always minimized.

nrmX = 2-norm (squared) convergence for both linear and nonlinear variables.

nrmF = 2-norm convergence for both linear and nonlinear constraints.

maxF# = constraint index for maximum constraint residual violation (after IMPL's presolve known as the organized / optimizable problem's constraint indice).

IMPL's SECQPE is a relatively standard implementation of an equality-constrained QP solver where no variable bounds nor inequality constraints are respected. The sparse LU factorizers Y12M and Intel's MKL PARDISO may be selected which are called at each iteration of the SECQPE similar to calling the LP or QP at each iteration of the SLPQPE solver. Convergence of the SECQPE solver is attained when the 2-norm of the (linear and) nonlinear constraints are at or below the specified convergence tolerance (CONVTOL = 1D-06, default) which includes the convergence of the objective function as another nonlinear constraint. Non-convergence or divergence is indicated when the maximum number of iterations is reached (MAXITER = 200, default) and the (linear and) nonlinear constraints plus the quadratic objective function have not converged to within the tolerance of convergence. Other SECQPE settings may be found in the IMPL.secqpe file which are modifiable or settable in IML file using the solver setting frame for example. For diagnostic and troubleshooting purposes, the constraint with the maximum absolute violation is also reported. The constraint index / indice number displayed is for the organized / optimizable constraint set as provided to the SECQPE solver after IMPL's primal presolver routines have been called or invoked i.e., shrinkability() and shrinkability2(). The IMPL file *.dtc may be used to relate the worst or largest constraint violation in its organized / optimizable index to the original problem's constraint index number and its name. Refer to the second last column just before the constraint name which is the presolved or organized / optimizable constraint index number with the third column being the original constraint number. Also for iteration or succession number zero (0), displayed immediately above the SECQPE solver log header, this is the maximum linear or nonlinear constraint index number for the constraint residual, response or result with the largest squared value at the start, beginning or commencement of the solver.

Tuning IMPL's Memory Requirement (IMPL.mem / Memory Frame)

IMPL allocates or reserves random-access memory (RAM) dynamically each time the IMPL Console or the IMPLreserve() routine is executed or invoked by reading or importing the fact.mem / subject.mem or the IMPL.mem file. Given that IMPL may be called simultanelously or concurrently on multiple CPU's or processors (i.e., model-side and multi-process parallelism), managing shared memory is vital to enable as many problems or sub-problems to be run in parallel or concurrently as is physically possible. In order to adjust or tune IMPL's memory for any specific problem, the IMPL report file fact.rdt / subject.rdt details the amount of memory consumed and committed per resource-entity as indicated in this report file's section with lines labeled `Series-Set:`, `Simple-Set:`, `Symbol-Set:`, `Catalog:`, `List:`, `Paramter:`, `Variable:`, `Constraint:`, `Derivative:`, `Expression:` and `Formula:`.

Once a successful execution of the problem has been completed, the first memory tuning is to adjust the `Derivative:` and `Expression:` memory line items using the column or field labeled `LENVAL` where the value to the left of the "|" pipe character is the actual memory consumed or used by the problem or sub-problem and the value to the right is the actual or existing memory allocated or committed to in the *.mem file. Based on these values, the user, modeler or analyst may trim, tweak, tinker or tune accordingly. The `LENVAL` column or attribute is the array length for the values for each resource-entity of type either integer, real or string. The second memory tuning should focus on the other resource-entities that have the greatest difference between their actual memory and their allocated or assumed memory. The third memory tuning is to adjust the memory required to store the multiple resource-entity keys (or subscripts / indices / iterators) found in the column labeled `LENKEY`. The fourth memory tuning is to adjust the `LEN` column which also requires the attribute `LENPRIME` to be adjusted.

The `LENPRIME` attribute should be at least 1.25 to 2.0 times (1.7 default) greater than the `LEN` attribute and should be, but not necessarily, a prime number although prime numbers are better in terms of more efficient IMPL memory storage and access. IMPL recommends the user, modeler or analyst to simply make `LENPRIME` an odd number and notionally or in-spirit a prime number where a prime number is not obligatory although it does help with our memory lookup search method efficiency i.e., less probing / collisions to find or locate available, empty or unused memory locations. Optionally, the user, modeler or analyst may use the calculation of p = 2^n - 1 known as the Mersenne prime number structure where n = ROUND(LN(p - 1.0)/LN(2.0)) and p is the desired nominal value for `LENPRIME`. For example, if the nominal `LENPRIME` is required to be approximately 500000, then n = ROUND(LN(500000 - 1.0)/LN(2.0)) = 19 and thus `LENPRIME` = 2^19 - 1 = 524,287. Obviously, the most reliable approach to find true or real prime numbers is to lookup published values in the desired range (i.e., 1.25 and 2.0 times `LEN`) and set this as the `LENPRIME` value. And finally, to compute an actual prime number, the datacalc function PRIMEN() may be invoked which calls IMPLpn() from the IMPL Server module.

By following the above recommendations, the user, modeler or analyst can significantly decrease the amount of RAM allocated / committed by IMPL from its pre-defined or default memory settings. Conversely, if IMPL tries to access more memory than specified and allocated in the *.mem file, then these guidelines may be also be employed to systematically increase the committed RAM. And for the convenience of the user, modeler or analyst, the memory frame configurable anywhere in the IML file may also be used which for the data-related resource-entities performs a resizing procedure using IMPL Server’s IMPLresize() routine and for the model-related resource-entities uses the IMPLrelease() and IMPLreserve() routines respectively.

A guide to help assist the user, modeler or analyst in identifying or locating which memory attribute to alter, change or modify in the event of a memory bounds checking error or exception is provided below where **`xx = ss (series-set), ys (symbol-set), g (catalog), l (list), p (parameter), v (variable), c (constraint), d (derivative), e (expression), f (formula):`**

```
xxr  -> LENXX
xxhi -> LENXXPRIME
```

```
xxki -> LENXX
xxk  -> LENXXKEY
xxvi -> LENXX
xxs  -> LENXX
xxcv -> LENXXVAL or xxtv -> LENXXVAL
xxv  -> LENXXVAL
```

Finally, the IMPL setting WRITEMEMORYFILE if greater than one (1.0), will call or invoke the IMPL Server routine IMPLwritememory() from the IMPL Console to create or generate a *problem-specific* or *model-specific* fact.mem / subject.mem memory file by sizing or scaling the actual or existing resource-entities' memory attributes by the real value of WRITEMEMORYFILE. Take for instance the situation where WRITEMEMORYFILE = 2.0, IMPL will write, print, export or output a fact.mem / subject.mem memory file upon completion of the IMPL Console with the current memory requirements multiplied, scaled or sized by 2.0 across-the-board for all of the above memory settings i.e., LENXX, LENXXPRIME, etc. This will allow the user, modeler or analyst to use this fact.mem or subject.mem memory file for the next run or execution of IMPL provided enough memory buffer or excess has been sized to accommodate a new but most likely similar problem or sub-problem.

Variable Initial-Value / Starting-Point / Default-Result Generation

IMPL has several techniques to manage initial-values, starting-guesses or default-results for variables which is especially advantageous for nonlinear, degenerate and non-convex (quality) problems where degenerate implies duplicate objective function values. For linear and mixed-integer linear (quantity and logistics) problems, initial-values are not explicitly required as these are handled automatically by the linear solver's "crash-basis" methods where LP and QP solvers are generally globally convergent. Fundamentally, the initial-value generation described below has essentially three (3) possible approaches, a) user-, modeler- or analyst-defined (pre-defined) starting-guesses based on known insight and understanding of the problem, b) default-results from previous or prior runs or executions of the problem or sub-problem known as a warm-start and c) randomization of the starting-points based on their known and finite lower and upper bounds or their existing initial-values. One interesting example of point b) is to partially solve the problem i.e., solve a sub-problem first with some constraints removed, ignored or dropped such as solving a volume balance first, then a volume-times-density mass balance second whereby the volume variables' sub-problem solution is used to warm-start the volume-times-density problem.

1. Set the RANDSEED setting in the IMPL.set file which can be used to change the randomization of the initial-values of all variables using the "Randomized Standard Heuristic" described in Ibrahim and Chinneck (2005) and Chinneck (2008). Each variable's initial-value is set to (lower + upper) / 2 + (upper – lower) * (RANDOM – 0.5) where RANDOM is a random number that varies uniformly between zero (0.0) and one (1.0) which is seeded using the RANDSEED setting.
2. Set the IMPL Console flag "-frequency=nnn" where nnn is any number from 0 to 2^31 for randomized re-starts. This will re-solve the problem by automatically incrementing RANDSEED by increments of one (1) up to and including nnn i.e., RANDSEED + nnn. A summary at the end of the randomized re-start loop is displayed in the console window where the *.exl and the OML *.dat / *.csv output files are appended with "__nnn" (double underscores) corresponding to the frequency run number or index. If nnn is negative (-ve), then the IMPL Console program will end, stop or terminate upon the first converged solution regardless of the total number of frequency instances specified or configured by ABS(nnn) or |nnn|.
3. Set the IMPL Console flag "-fuse=warm" which will save the variable results or responses from a previous run or execution using any solver which can be used as initial-values for the next run / execution of IMPL provided the data structure of the modeled problem has not changed since the previous run. After the IMPLmodelerv(fact, …, force=VARIABLE) has been called or invoked, then IMPLrevise2(VARIABLE, …) is called to revise or replace all of the variable result or response values with those from the previous run found in the file fact.vv / subject.vv. If nonlinear foreign-constraints are present, then IMPL will also revise or replace their default-results after the IMPLmodelerc(fact, …, force=CONSTRAINT) since the foreign-model is imported, read or inputted in the IMPLmodelerc() and the IMPLmodelerv().
4. Set the target field (soft bound) to your initial-value in the IML file for any quality variable only and this non-zero value will be used as a starting-point / initial-guess for all time-periods / -intervals / -steps irrespective if its performance 1-norm (absolute, LP, Manhattan) or 2-norm (squared, QP, Euclidean) weight is configured – this is of course a user-, modeler- or analyst-defined, static, master or model data initial-value and is not time-varying as opposed to the first three initial-value generation techniques.

5. And, for problems with foreign-variables and -constraints (i.e., with configured ILPet / ILP and INPet / INP foreign-files), please see the IMPL setting RANDOMIZEX which if one (1), randomizes all included problem variables using a similar approach to the above found in the IMPL Server's startability() routine invoked or called by the IMPL Presolver. If the foreign-variables lower and upper bounds are both free and infinte (not finite) i.e., less than or equal to -INFIN and greater than or equal to INFIN, the startability() routine simply adds a random number between zero (0.0) and one (1.0) to the existing initial-value, starting-guess or default-result.

Guidelines for Managing Infeasibilities / Inconsistencies

IMPL has several techniques to handle problems when hard infeasibilities / inconsistencies due to conflicting constraints and/or variable bounds are detected primarily as a result of gross errors, defects, faults, etc. in the data also referred to as "bad data" but may also be due to a "bad model" which is incomplete, inconsistent, erroneous, mismatched, etc. Infeasibilities occur when at least one hard constraint right-hand-side (R.H.S.) or variable bound is violated given the model and data of the problem which is most likely inconsistent or bad. From experience the other most likely or common cause of infeasibilities are notoriously related to at least one of the variables' (or constraints') lower bound which cannot be achieved or satisfied especially when the lower bound is positive and zero (0D+0) for its solution value does not respect it. Hence, we encourage the user, modeler or analyst to configure lower bounds with caution i.e., incrementally / one-at-a-time. Usually infeasibilities are detected in IMPL's linear primal presolver or in the other linear presolvers of the LP, QP, MILP and MIQP solvers though they typically only report or display the first or last infeasibility found. It should be emphasized that these are only the symptoms / faults / defects / manifestations of the issue and not necessarily the root cause or reason why the infeasibility has occurred. Therefore, diagnosing and troubleshooting the underlying root cause or source or the issue requires analysis by the user, modeler or analyst to ascertain the underlying reason for the contention / conflict / inconsistency in the problem. For example, if a pool, tank, drum or storage vessel reaches its upper or maximum holdup bound (i.e., symptom, defect, bug, effect or fault), is the root cause that there is too much flow in, not enough flow out, a faulty measurement or that the pool's upper holdup bound, and/or its opening holdup are configured wrong?

The guidelines and methodology below are recommendations on how to aid in the debugging of infeasible problems through the methods of intelligent problem-solving. They are intended to help answer the questions of what, where and when though the why usually requires deeper and ancillary knowledge, experience and insight into the underlying problem from the user, modeler, analyst, etc.

1. **Always try to maintain one or more *feasibility references or bases*** (i.e., both model and cycle data instances) of the problem for different days, weeks, months, quarters and years. This is known as a touchstone, benchmark, safe park, panic room, etc. These provide feasibility points, baselines or base cases where the consistency of the problem is known and can be referred back to as a feasibility basis from which the troubleshooting can be grounded so to speak. Generally speaking, a solid and sound place to start, initialize or begin the troubleshooting and debugging of infeasibilities is from a feasible / consistent point and to progress out towards the infeasibilities hopefully one-at-a-time. That is, if certain model and cycle data constructs, configurations and/or combinations were feasible in past IMPL runs, then why are they not feasible now? (Hence the notion of a touchstone, benchmark, feasibility point, etc.). Unfortunately when modeling and solving planning, scheduling and control problems, which have both static / master (model) and dynamic / transactional (cycle) data, each cycle creates essentially a completely new problem with its own unique nuasances and issues thus complicating and convoluting the analysis.

2. **IMPL supports configuring excursion variables for all quantity, logic and quality variables (known as *feasibility relaxation*) in the problem** as well as excursions or exceptions for key logic and logistics constraints (see the "@,@" attributes for the constraint penalty-weights); these excursion variables are also known as infeasibility-breakers, safety-valves, penalties, elastic, excess or artificial variables. It is recommended to always model and solve your problem first without excursion variables, but if the problem is detected to be hard infeasible after the presolvers or in the solvers themselves, then apply penalty-weights to all (globally) or some (locally) of the phenomenological variables using the global excursion weight settings found in the IMPL.set file (see constant data) i.e., QUANTITYEXCURSIONWEIGHT, LOGICEXCURSIONWEIGHT and QUALITYEXCURSIONWEIGHT and/or the local excursion penalty-weights specific to each quantity, logic and quality variable (see cost data). When excursion variables exist in the problem formulation, less presolving will be applied and thus the path to

solutions will inevitably (and unfortunately) take longer to solve also slowing down the debugging process. Especially for logistics optimization with MIP, it is recommeneded to continue solving the problem with the added excursion variables for as long as possible in order to find solutions with the smallest / least number or cardinality of active excursions irrespective of their values or amounts as these solutions are easier to understand and diagnose. Also, considering setting the MIP optimization to focus on optimality versus feasibility (cf. CPLEX's MIPEMPHASIS and GUROBI's MIPFOCUS settings). Solutions with many active excursion variables can be difficult to assess whereas solutions with the least number of excursion variables is preferred (i.e., Occam's razor / law of parsimony) as narrowing down or isolating areas or regions of the model and data is paramount in order to find and remedy / repair the issue or defect.

When solving quality (nonlinear) optimization problems with IMPL's SLPQPE solver or other SLP algorithms such as XPRESS-SLP, error variables are automatically augmented to the problem's IMPL presolved nonlinear constraints (standard technique in all SLP algorithms) and monitoring these errors, which cannot be driven to zero (0.0), can be an efficient and effective way to help determine why the problem does not converge or at least to circumscribe some problematic part of it. This may be due to an "apparent" or an "actual" infeasibility in the problem. Apparent infeasibilities are most likely casued by poor initial-values, starting-points or default-results. Please see the frequency flag which can be used to systematically randomize the initial-values of the problem or sub-problem and the problem re-solved to perhaps find a different path to a converged or even better solution. Other nonlinear programming solvers (NLP's) such as IPOPT use a restoration phase to automatically and hopefully recovery a feasible point but it has been found that this approach is not as effective as the augmented errors applied to nonlinear constraints only available in SLPQPE.

In addition, when solving quality optimization problems or sub-problems via SLP technology (e.g., IMPL-SLPQPE), it is important to set or specify the local (and global QUALITYEXCURSIONWEIGHT) penalty or excursion weights for the quality variables to be some real number less than (<) the error weight setting or option used in the SLP solver i.e., if the error weight is 1D+6, then the excursion (penalty) weight should be 1D+3 for example which also depends on the unit-of-measure for the quality variable. However, for quantity excursion variables, their excursion *weights should be greater than (>) than the error weight as these variables are not (usually) involved in constraints that are nonlinear. The issue when the excursion weight needs to be less than (<) the error weight is when an excursion variable is involved in a nonlinear constraints whereby SLP algorithms append the error variable automatically and therefore confounding and conflating the excursion and error variables.*

3. **IMPL supports configuring "penalty-tolerances" / "excursion-tolerances" for selected excursion variables applied to quantity and quality variables only *(known as fuzzy-feasibility)* in the problem** - see the "@,@" asperand special character attributes adjacent to the penalty-weights. Excursion- or penalty-tolerances can be absolute if positive (+ve) and relative if negative (-ve) whereby the user, modeler or analyst can provide a limited amount or level of bounded random and/or systematic error uncertainty IMPL calls a "de-infeasibility" zone (DIZ) or "fuzzy-feasibility" region i.e., the region where the limits and not black or white but grey. If the DIZ is insufficient for any given problem, then the problem is declared as hard infeasible where soft infeasible means that the problem or sub-problem is feasibly solvable but non-zero excursion variable activity exists in the solution where to clarify, excursions are departures from either its lower or upper bounds. *In essence, the de-infeasibility zone (DIZ) and the fuzzy-feasibility can be likened to having a constraint that is both soft and hard i.e., soft within the excursion-tolerances and hard outside of its lower and upper penalty-tolerances.*

4. **IMPL supports configuring "suspensions", "substitutions" and "situations" (i.e., *feasibility rectification / restoration / recovery)* using the furcate flag.** This is the concept of manually and temporally removing, replacing, reducing and/or relaxing certain potentially infeasible or inconsistent quantity, logic / logistics and quality constraints via suspensions / substitutions to test whether the problem can be made feasible in a trial-and-error (guess-and-check) fashion without these constraints. Keeping a record of where in the problem infeasibilities have occurred in previous runs, executions or cycles, is a useful way to establish infeasibility patterns or precedents in your problem and to locally add excursion variables in these locations. A very common infeasibility precedent is handling out of bounds opening quantities and qualities after a tank, storage vessel or pool unit is put back in to service from maintenance.

In addition, fixing, freezing and forcing or even favouring certain quantity, logic and quality variable situations, even at different points in time, can also be employed to understand why a problem or sub-problem is inconsistent or perhaps why a solution, albiet feasible, has unexpected results. It should be pointed out that feasible solutions with unexpected / unexplainable variable solution results or responses must also be debugged and diagnosed to ensure that solutions provide an acceptably accurate and expected representation of the physical / actual system being predicted and prescribed in the model (cyber-physical / digital-twin). An effective approach to aid in this endeavor is to always perform one or more simulations where we fix all or almost all of the degrees-of-freedom in the problem to “match” a known solution typically from an actual sample of the plant, production or process system. This is to ensure that the predictions from the cyber-physical system or the digital-twin / digital-threads are consistent with the actuals from the physical system or analog-twin.

A related method which is particularly applicable and effective for dynamic problems, is to incrementally increase or extend the future time-horizon / -profile / -perspective / -purview from some feasible shorter starting profile (if possible), to help pinpoint, isolate or narrow down the time-line for when the infeasibility starts to occur. Since industrial optimization and estimation problems are both temporal and structural in nature (i.e., are dynamic / time-dependent / time-varying / time-variant / transient and involve a network of inter-connections, paths, routes, transfers, etc.), locating a point in time (and space) when (and where) the inconsistency is manifested has proven to be a valuable aid in helping to analyze infeasibilities / inconsistencies for industrial problems.

Most Commonly Encountered Infeasibilities / Inconsistencies …

Here we list some of the most common samples, situations or scenarios of infeasibility or inconsistency root causes found in industrial scheduling control (“schedcon”) and optimization (“schedopt”) problems of which all of these are related to lower bounding issues. Note that this list will be updated regularly as other common infeasibility / inconsistency occurrences or events are encountered.

1. An opening quantity or holdup is lower than its configured lower bound after the tank, storage vessel or pool unit has returned from maintenance and its deadstock inventory (i.e., holdup that cannot be normally drawn from) is close to zero (0.0) and below its normal operating lower bound.

   Another cause is related to opening pool qualities (i.e., densities, components and properties) when a pool has switched material-services or operations from one grade to another where the previous grade's material is in fact off-specification for the next grade i.e., either violates the quality's lower or upper bound.

2. A noncontiguous order's lower bound holdup quantity on a unit-operation-port-state is in fact greater than its internal- or tee-stream's upper bound total flow quantity.

3. A unit-operation is forced to be setup for some time-plank (i.e., time-window, -interval, -range, -span or multiple contiguous / consecutive time-periods/-steps) using a logic setup order, transaction or command and, on one of its in- or out-port-states, a lower multi-use bound is configured to ensure that there is at least one or more flows-in or flows-out at the port-state. However, there is some upstream or downstream flow logic or logistics restriction that prevents this from occurring during the specified time-plank such as an upstream / downstream pool unit-operation with the filling-to-full / drawing-to-empty constriction, etc.

Composing Problems with Sub-Problems and Decomposing Problems

All industrial-scale optimization and estimation problems are composed of smaller sub-problems i.e., sub-balances, sub-blocks, sub-flowsheets, sub-formulations, sub-models, sub-envelopes, sub-systems, etc. IMPL supports two (2) relatively simple approaches to managing the complexity of their configurations. The first is to configure or build each smaller sub-problem separately / independently whereby the user, modeler or analyst ensures that each sub-problem provides feasible and expected results when modeled and solved. Then, common data may be used to link, connect or compose several / multiple sub-problems together into a single / mono problem by simply adding common / connecting / coupling / composing equality contraints. From an intelligent problem-solving perspective, if any of the sub-problems cannot be solved feasibly, then of course the overall, combined or composed problem is also insolvable. Hence, modeling and solving separate sub-problems is a useful way to help resolve infeasibility / inconsistency issues in the problem's model and/or the data (i.e., master / static and transactional / dynamic data).

The second is to configure the larger single / mono problem initially and then using the frame `Mask-&sUnit,&sOperation`, to filter, exclude, skip, passover, ignore or mask unit-operations that should not be contained or included into each sub-problem and this is the concept of decomposing a single problem into many smaller sub-problems. However, when decomposing into individual sub-problems it will be required to ignore the overall quantity and quality balances on certain interchange, exchange or boundary (inbound/front-edge, outbound/back-edge) unit-operations as these unit-operations will have one or more in- and out-port-states missing / ignored / passovered and hence no overall balance will be possible.

Steps to Invoke the Industrial/Iterative Decomposition Heuristic (IDH)

The Industrial / Interactive / Incremental / Inductive / Intuitive / Iterative Decomposition Heuristic (IDH), which may considered as a *constructive greedy search heuristic*, comes in two (2) flavours. The first is considered as a rolling-horizon heuristic (RHH) or temporal decomposition approach called the CDH from Kelly, "Chronological decomposition heuristic for scheduling: a divide and conquer method", *American Institute of Chemical Engineering Journal*, 2002. And the second, called the FDH from Kelly and Mann, "Flowsheet decomposition heuristic applied to scheduling: a relax and fix method, *Computers & Chemical Engineering*, 2004 which is a structural, spatial or flowsheet decomposition approach and employs the well-known relax-and-fix heuristic (RFH).

In order to invoke the CDH flavour of the IDH where for illustrative purposes only we decompose the time-horizon, -profile, -perspective or -purview into only two (2) time-chunks or sub-time-profiles, there is a preliminary step one (1) and a few IMPL Console executable flags that need to be specified or configured as follows:

1. Configure the first time-chunk or temporal partition to be some appropriately chosen shorter future sub-time-profile or smaller number of time-periods into the future and run a logistics

(MIP) sub-solve to find logistics-/-integer-feasible sub-solutions. This should take less computational time as the time-profile has been shortened.

2. Choose a logistics- / integer-feasible sub-solution which may or may not be the "best" logistics MIP sub-solution found and set the **feed flag** and the **figure flag** (i.e., a 64-bit or 8-byte integer) accordingly.
3. Set the **feasibility flag** to `logics` which will allow the logistics sub-problem to be re-optimized.
4. Set the 64-bit or 8-byte integer **folio flag** to a number distinct from the figure flag e.g., `100` so that the IDH found logistics-feasible sub-solutions do not overwrite the previously found logistics-feasible sub-solutions. See also the **folioprefix** and **figureprefix flags** which are text string character prefixes.
5. Set the **fade flag** to be either the same duration as the shortened sub-time-profile or something even shorter which increases the "crossover" or temporal overlap as described by Kelly (2002). The fade (or otherwise known as the fadeafter) flag ignores, skips, passes-over or forgets any unit-operation (UO) and unit-operation-port-state-unit-operation-port-state (UOPSUOPS, UOPS2 or UOPSPSUO) tee- / external-stream setup logic orders, commands or provisos (loaded from the figure and feed flags' EXL file) with start-times greater than the fade flag.
6. Set the time-profile back to the original time-horizon duration and re-run the logistics (MIP) solver to hopefully find logistics- / MIP-feasible solutions for the full time-profile into the future. *If the full logistics problem is infeasible or inferior, dependent on the previously chosen logistics-feasible sub-solution, then choose a different preliminary logistics sub-solution and/or increase the crossover / overlap amount of time.*
7. See also the **fix, flip** and **fadebefore flags**.

In order to invoke the FDH flavour of the IDH, this requires more knowledge of the UOPSS flowsheet as the structural or spatial decomposition is based on subjectively and hypothetically separating the superstructure into conceptually "most important" (complicated, difficult, etc.) and "least important" sections, sectors or sub-superstructures if you will. The "most important" section has its discrete logics declared, as usual, as binary bounded by zero (0) and one (1) and the "least important" sector has its discrete logics declared as continuous but of course still bounded by zero (0.0) and one (1.0).

1. Configure the "least important" section of the flowsheet by relaxing their UO and UOPSUOPS setup logics to have negative (-ve) lower bounds i.e., less than the lower bound threshold set by

the IMPL setting or option RELAXLOGICTHRESHOLD (-1000.0, default). IMPL will automatically declare these binary variables to be of type non-discrete (continuous) and lie between zero (0.0) and one (1.0) i.e., within the continuous range or domain [0.0,1.0] where the "most important" setup logics, as usual and as mentioned previouslty, are declared to be either at zero (0) or one (1) i.e., they lie within the discrete set {0,1}. Run a logistics (MIP) sub-solve to find logistics- / -integer-feasible sub-solutions where this should take less computational time as there are less explicitly declared discrete binary variables.

2. Choose a logistics- / MIP-feasible sub-solution which may or may not be the "best" logistics MIP sub-solution found and set the **feed flag** and the **figure flag** accordingly.
3. Set the **feasibility flag** to `logics` which will allow the logistics sub-problem to be re-optimized.
4. Set the **folio flag** to a integer number distinct from the figure flag e.g., `100` so that the IDH found logistics-feasible sub-solutions do not overwrite the previously found logistics-feasible sub-solutions. See also the **folioprefix** and **figureprefix flags** where are text string character prefixes.
5. Before re-running the logistics MIP solver, re-configure the "least important" setup / startup logics back to have zero (0) lower bounds (i.e., lower bounds greater than or equal to RELAXLOGICTHRESHOLD) and re-run the logistics (MIP) solver to hopefully find logistics / integer-feasible solutions for the full logistics problem. *If the full logistics problem is infeasible or inferior, dependent on the previously chosen logistics-feasible sub-solution, then choose a different preliminary logistics sub-solution else consider changing the demarcation between the most and least important UO and UOPSUOPS setup logics.*
6. See also the **fix** and **flip flags**.

It is worth mentioning that the CDH and the FDH flavours of the IDH may also be combined together in that order or sequence to both reduce the time-profile / -horizon and relax the least important UO and UOPS2 setup logics in order to apply a double decomposition in both time and space. For instance, MIP solve the logistics sub-problem with the shorter time-horizon with the least important setup binaries relaxed. Next, choose a logistics sub-solution assuming that one or more are found incidentally and unrelax the least important setups and MIP solve again to find hopefully multiple logistics-feasible sub-solution with no relaxed setup binaries but with the shorter number of time-periods. Then, repeat the above for the full time-profile which also requires two (2) more MIP solves i.e., one with relaxed least important setup logics and another with unrelaxed least important setups.

It is also interesting to advise that a convenient demarcation or division of the most and least important unit-operations (UO) and unit-operation-port-state-unit-operation-port-state (UOPSUOPS, UOPS2 or UOPSPSUO) tee- / external-streams can be managed via the IML aliases. For example, we have found that arbitrarily dividing the flowsheet into say an upstream part and a downstream part whereby the upstream or downstream can be labeled as the most important and the downstream or upstream can be labeled as the least. Another segmentation may be based on prioritizing particular operations, tasks, modes, materials or grades such as the higher volume of product(s) being the most important and the lower volume of products(s) as the least important i.e., a stock decomposition heuristic (SDH).

Steps to Invoke the Industrial/Incumbent Diversification Search (IDS)

In order to invoke the Industrial / Incremental / Incumbent Diversification Search (IDS) which can be used similar to a "MIP solution-pool" found in most commercial MIP solvers and may be considered as a *local improvement heuristic search*, there is a prelinminary step one (1) and a few IMPL Console executable flags as well as the IMPL setting INCLUDEMIPSTARTS that need to be specified or configured as follows:

1. Choose the logistics- / integer-feasible solution that is to be the "incumbent" from some previously run logistics (MIP) sub-solve and set the **feed flag** and the **figure flag** accordingly. This incumbent may also be a logistics-feasible solution that is either a quality-feasible, -infeasible or -inferior solution.
2. Set the **feasibility flag** to `logics` which will allow the logistics problem or sub-problem to re-optimized influenced or guided by the (sub-)solution and referenced by the figure and feed flags.
3. Set the **folio flag** to a integer number distinct from the figure flag e.g., `100` so that all IDS found logistics-feasible solutions do not overwrite the existing, original or previously found logistics-feasible solutions or sub-solutions. See also the **folioprefix** and **figureprefix flags** which are text string character prefixes.
4. Set the **fish flag** to a relatively small non-zero positive (+ve) or negative (-ve) real number such as `0.1D+0` or `0.1D+0`. The fish flag represents an objective function weight to maximize the incumbent diversification variable constrained based on the incumbent's "active" ("open", "on",

"setup", etc.) unit-operation (UO) and unit-operation-port-state-unit-operation-port-state (UOPSUOPS, UOPS2 or UOPSPSUO) tee- / -external-stream setup logics. That is, those UO and UOPS2 setups that are at or near one (1).

5. Set the **INCLUDEMIPSTARTS setting** to either one (1, default) or zero (0). If one (1), then IMPL adds the well-known "MIP starts" capability to the commercial solvers CPLEX, COPT, GUROBI and XPRESS as well as the community solver HIGHS for those UO and UOPSUOPS setup (or startups) logic binaries that are at or near one (1). This setting, if not zero (0), will help to speed up the IDS as it has a starting, initial or incumbent solution but will not necessarily find more or better diverse logistics-feasible solutions. And, if INCLUDEMIPSTARTS equals zero (0), then instead of storing the setup (or startup) logics in the target soft bound field or attribute of the orders, commands, provisos or transactions required by INCLUDEMIPSTARTS = 1 and the IDS, they are stored in the lower and upper hard bound fields and if the logic value is near-one or at one (1), then IMPL will fix only those setups (or startups) that are near-one or at one (1) allowing the rest of the logic binary values that are zero (0) or not one (1) to be free. As such with INCLUDEMIPSTARTS = 0, this will of course reduce the number of discrete binary or logic variables significantly at the expense of the problem of sub-problem being over-specified and possibly infeasible and for IMPL's Industrial / Incumbent Decomposition Heuristic (IDH), INCLUDEMIPSTARTS = 0 is required.
6. See also the **fix** and **flip flags**.