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 !"

Industrial Modeling Language (IML)

IML© IMPL-DATA© / IDL© Deployment 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-IML-RMID-1-10*

Introduction

The IML (Industrial Modeling Language) file is our user readable import or input file to the proprietary (closed-source) and purpose-driven (fit-for-purpose) IMPL modeling and solving platform which is implemented in the IMPL Interfacer for “no-code” or “low-code” types of implementations. IMPL is an acronym for *Industrial Modeling & Programming Language* provided by *Industrial Algorithms Limited (IAL)* and is short for “IMPLementable” with perhaps one of its most important goals of helping its industrial users, modelers and analysts with being able to produce more products reliably of better quality consuming cheaper feeds in an economical, efficient, environmental, sustainable and smarter manner consistent with the well-known definition of optimization as the action of making the best, optimus or most effective use of a situation and/or resources. A more elaborate definition of optimization is for the purpose of satisfying lower, upper and target hard and soft bounds and constraints whilst simultaneously seeking the best objective function value(s).

The CSV-based IML file allows the user to configure the necessary data to model and solve large-scale and complex industrial optimization and estimation problems (IOP's and IEP’s) such as planning, scheduling, coordinating, control and data reconciliation and regression in either off- or on-line environments enabling the future of better of decision-making. The data configurable in the IML file are broken-down into several categories or classes where these data categories are used as further sections in this manual. Essentially, this is what we call static (non-time-varying or time-invariant) and dynamic (time-varying, time-dependent or transient) problem data (master and transactional) which are used to configure and control the IOP’s / IEP’s. It should also be clear from these data categories that all of these can be further classed into two higher-levels known as “configuration” and “cycle” data. Configuration data includes all data except for the cycle data found in the data categories of content (current) and command (control). Configuration data is typically static whereas cycle data is dynamic and explicitly has a time or temporal dimension attached to them to represent that the command, event, order, proviso or transaction has a defined beginning and end. The word cycle is similar to the concept of a case but hopefully provides the connotation that the IOP / IEP is executed, run or spawned on a regular / routine basis or interval most commonly referred to as the receding / moving horizon which helps to mitigate the omnipresent effects of uncertainty and variability. This is of course very well-known in the field of model predictive control (MPC) which can be likened to an on-line version off-line advanced planning and scheduling (APS) with measurement (parameter) feedback.

This manual is specific to IMPL-DATA©'s / IDL©'s industrial data language capability only.

One of the most important characters of IMPL is the "!" exclamation character which is IMPL's standard comment character and can be used for in-line comments as well i.e., comment character "!" not in column one (1). Any feature, row or line of the IML file starting with a "!" and any characters after the "!" will be ignored where any feature, row or line in the IML file not recognized by IMPL will also be considered as a comment feature, row or line. The "!" character is the primary comment symbol for IMPL where the backslash "\" character may only be used in the ILP / ILPet and INP / INPet foreign-files as a secondary comment character. It is helpful at this point to summarize and list the special input-related characters used within IMPL and they are: "," (comma*, universal and primary delimiter*), ";" (semi-colon*, secondary delimiter*), ":" (colon*, delimiter*), "`" (tick*, continuation*), "!" (exclamation mark*, comment*), "\" (backslash*, path and file name delimiter and ILP / INP comment*), "&" (ampersand), "@" (asperand, each, at), "$" (dollar sign), "?" (question mark), "|" (pipe*, delimiter*), "#" (hash, number), "~" (tilde) "=" (equal), "<" (less than), ">" (greater than), "( )" (parentheses, *data functions and grouping expressions in formulas or expressions*), "{ }" (curly braces, *grouping expressions in formulas or expressions*), "[ ]" (brackets, *data-vector-element and time-/trial-lags, -delays or shifts*), "^" (caret, circumflex, *exponential power*) and "." (period).

And furthermore out of an abundance of caution, it is important to note that for reliability reasons, it is strongly recommended that IML files and their include files (IMLet) should not contain any (horizontal or vertical) tab characters (ASCII codes 9 and 11) whatsoever as these may cause unpredictable results given their invisible-ness. As well, there may also be the existence of "non-breaking spaces" in (Unicode Transformation Code 8-bit) which also appear as blank spaces or whitespaces in text editors but are UTF-8 characters which will be lexed and parsed as actual non-blank characters in IMPL. Although there are warning checks in the IMPL file processing to detect and identify where these special characters may be present, it is prudent to not include them.

IMPL-DATA© / IDL© (Industrial Data Language) Overview

Notably, the calculation data, constant data, computation data and concatenation (companion) data described above are all collectively referred to as **IMPL-DATA© / IDL© (IMPL-DATA Language or**

Industrial Data Language) shown in Figure 3 below which is a "fit-for-purpose" or "purpose-driven" appliance of IMPL and is what we refer to as basic or primitive Industrial Data Programming / Processing (IDP) or an Industrial Data Scripting Language (IDSL). The primary and fundamental purpose of IMPL-DATA / IDL and IDP / IDSL is to minimize the occurrences or events of redundant data and to facilitate relatively simple, straightforward and spreadsheet-like data pre- / post-processing capabilities involving both calc-scalars, data-vectors, data-vector-groups, data-vector-elements and text-strings. As such, IMPL-DATA / IDL may be considered as an Industrial Internet of Things (IIoT) calc-data / data-calc computation engine and is based on the idea, notion or concept of functional and imperative programming with its data functions. Further to that, IMPL-DATA / IDL is purposely not interactive per se but more "interfaceable" in terms of reading / writing, importing / exporting, loading / unloading and inputting / outputting *independent and dependent data*. The popular and more powerful interactive environments such as Mircosoft's Excel, Mathworks' Matlab, Wolfram's Mathematica, GNU's Octave, etc. and the Python (with NumPy and SciPy), R and Julia computer scripting languages, are always encouraged to be used to design, develop, deploy and debug more advanced and innovative, intelligent, insightful, interactive, integratable and interpretable methods and techniques. Therefore, IMPL-DATA / IDL should be considered as a collaborative and complementary vehicle to aid in both the off-line and on-line deployment of these data analytics technologies and to ultimately industrialize and operationalize them in near-real-time and real-time environments**. That said, IMPL-DATA / IDL should be considered principally as "last mile", "last minute" or "just-in-time" computer programming and scripting so to speak whereby it can provide the *fine tuning, finishing touches and final touchups* of industrial data immediately before and after the industrial problem or sub-problem is modeled and solved i.e., optimized or estimized.**

As previously mentioned, more advanced and sophisticated data-processing may be programmed with IPL (Industrial Programming Library) and/or IMPC (Industrial Mathematical Programming Code) using computer programming and scripting languages by the developer user, modeler or analyst. As well, recursive, successive and iterative data calculation blocks (imperative and functional programming) may be sequentially computed for example in Microsoft's DOS or Powershell batch file scripting languages to run / execute dynamic, nonlinear and/or implicit calculations in a sequence, series or simulation by repeatedly invoking the IMPL.exe console program although this may be inefficient as separate instances of IMPL.exe are executed or run.

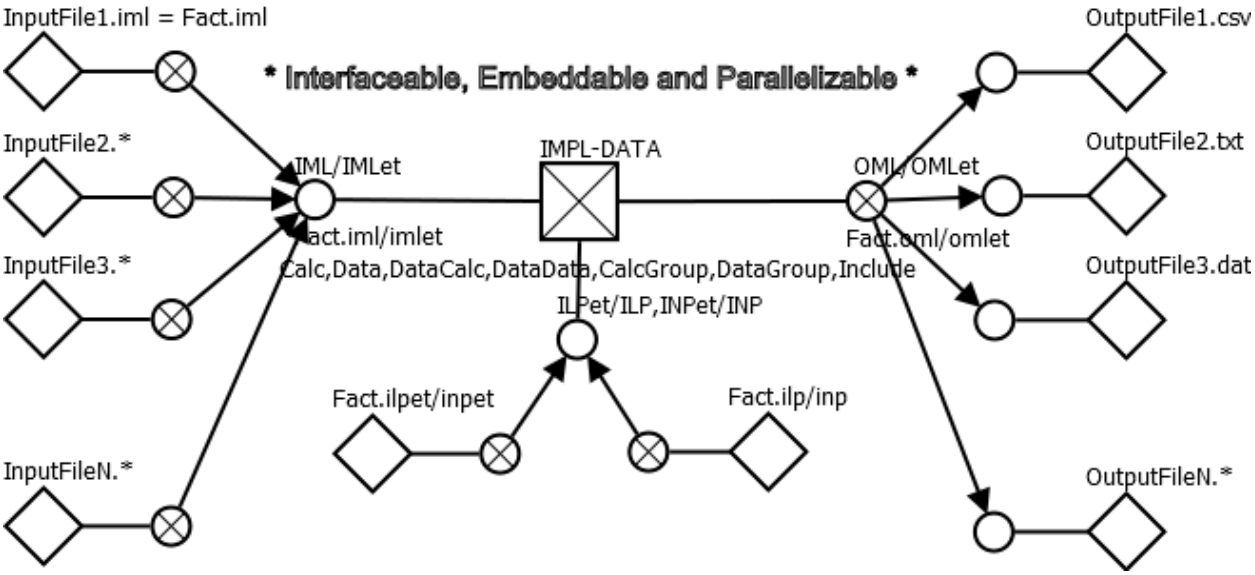

Figure 4. Flow Diagram of IMPL-DATA / IDL with various IML input and OML output files.

Further and more recently, integrated data studios (IDS's) such as Google Data Studio, Amazon QuickSight and SPICE (Super-fast, Parallel In-memory Computation Engine), SAP Lumira, Kilpfolio, DeepIQ's DataStudio, Uptake's / ShookIOT's Data Fusion, Element Analytics' Element Unify, etc. that can connect to virtually any data source and can visualize / view structured, unstructured and time-series/-ordered data are growing in popularity. Consequently, IMPL-DATA / IDL and IDP / IDSL may be easily called on as a basic data calculation and advanced data computation engine (DCE) to turn or convert independent / exogenous data into dependent / endogenous data or information via thoughtful and timely engineering, empirical and analytical ("engalytical") data calculations, computations, correlations, expressions, formulas and relations. And, IMPL-DATA / IDL is a standalone data calc / comp engine separate from the usually built-in real-time data historian calculation engines such as Aspen Technologies' Aspen-Calc or Aspen's CCF (DMCplus / DMC3 Controller Configuration File), AVEVA's (formerly OSIsoft's) PI-ACE and Honeywell's PHD Virtual Tags. In addition, IMPL-DATA / IDL also provides the necessary capability to perform both discrete, nonlinear and dynamic (DND) optimization and estimization via its ILP / INP foreign-files which sets IMPL-DATA / IDL apart from other types of calculation / computation engines.

For more insight into applying IMPL-DATA, we offer the following overview. All industrial applications input and output data from / to one or more data sources / sinks respectively as we show in the attached flow diagram or flowsheet entitled "Anatomy of an Industrial Application". Arguably, no

industrial application would be complete without converting input data to output data (information) via data calculations or computations which may or may not be model-based implying some form of estimation and/or optimization i.e., predictive and prescriptive analytics.

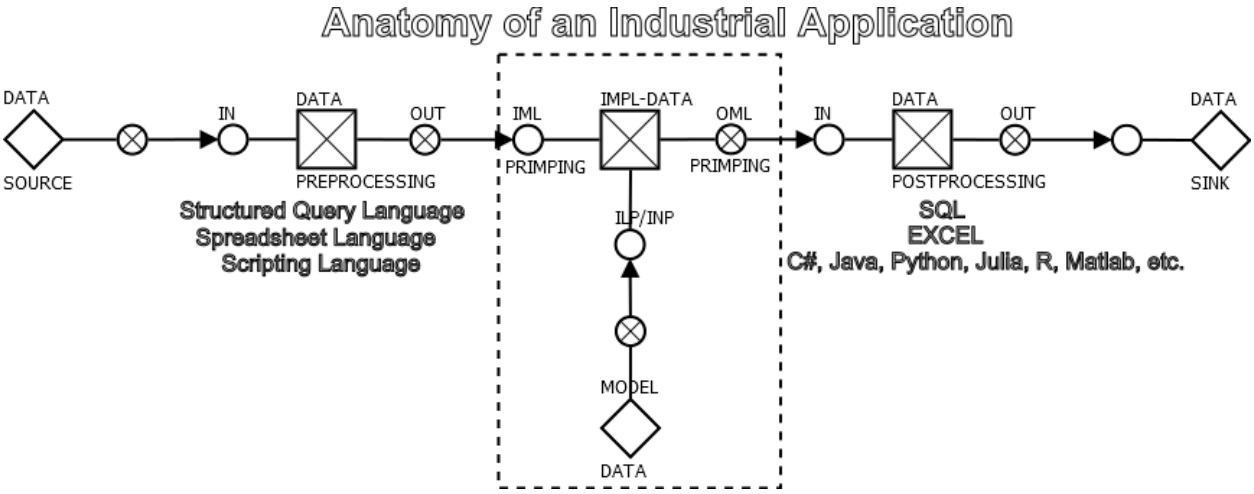

Figure 5. Flow Diagram of IMPL-DATA / IDL with various IML input and OML output files.

As the figure highlights, between the data preprocessing and postprocessing operations there is a data calculation and/or computation engine such as IMPL-DATA. If the industrial application involves estimization such as data reconciliation and regression or discrete, nonlinear and dynamic optimization, then some form of a model is required which includes sets, parameters, variables, constraints and derivatives. For those of you who develop and deploy industrial applications, unfortunately no matter how complete the data processing is before and after the data calculations / computations, which are usually coded in structured query, spreadsheet and/or scripting languages, inevitably there is always a need to perform a little more pre- and post-data processing which we call *data primping*. Industrial data primping is the final step before the input data retrieved from the data sources via the data preprocessing operation is processed and the initial step before the calculated or computed data is output from the engine and before being received in the data sinks via the data postprocessing operation.

Fortunately, IMPL-DATA's IML (Industrial Modeling Language) and OML (Output Modeling Language) have the built-in capability to perform the necessary data primping operations with its calc-scalar and data-vector data structures and data functions if the data transferred is formatted via CSV files. And, if the data is transferred in memory, then IMPL-DATA's application programming interface (API), software

developer kit (SDK) or computer programming / scripting link (CPL / CSL) can be invoked where IMPL-DATA is called from any computer programming / scripting language that can call dynamic link libraries or shared objects. Obviously some balance or tradeoff between the various levels or degrees of data processing and data primping programming is necessary as part of the industrial application's project, which can be somewhat demarcated in terms of Informational Technology (IT) and Operational Technology (OT) resources and skills, nevertheless, data primping is real and may be considered as a fine-tuning, tweaking, tinkering, trimming or touchup aspect of your industrial application's data processing needs.

Calculation Data

IMPL enables all IML number fields (integer and real) in all frames for capacity data, etc. to be entered as mathematical symbols, expressions, formulas or calculations / calc's via the IMPL compiler. These scalar calculations / calc-scalar's are useful to "pre-process" or "pre-program" the problem or sub-problem data before being used in any industrial optimization, estimization and simulation problem / sub-problem and is similar to the concept of scripting at least at a primitive level. A frequently used term when interfacing or interacting with tagname data for example from relational databases and real-time data historians (i.e., transactional, time-series / temporal databases) or other data sources is to "calcize", "symbolize", "tokenize" or "parameterize" or even "calcify" the IML file and this can be readily achieved using these IMPL calculations or what IMPL calls calc's or calc-scalars (cf. IMPL's console formulas and formulasfile flags). In essence, a calc-scalar is simply the binding of a name, key or symbol to any real value and may be altered, changed, modified or updated anywhere within the IML file.

Importantly, similar to the other character names allowed within IMPL's symbol naming convention, calculation / calc-scalar names must not include IMPL's primary delimiter the comma (",") and should not include characters recognized as a mathematical, arithmetic or bracketing operators nor IMPL's secondary delimiters such as semi-colons ";", colons ":", pipes "|", continuation ticks "`" and the comment exclamation "!" and backslash "\" characters. However, periods ".", underscores "_", tilde "~", ampersand or the and symbols "&", asperand, each or the at symbols "@", number signs "#", dollar signs "$", question marks "?", apostrophes "'",quotes """, percent "%", equal to "=", less than "<" and greater than ">" characters are all valid and acceptable naming or symbol characters for scalar calculations / calc-scalars and data-sets, -lists and -vectors as well as text-strings.

Calculation / calc-scalar expressions or formulas may include arithmetic operators: **"+", "-", "*", "/" and "^"** where each operator must be separated or grouped by an operand, parenthesis **"()"**, including curly braces **"{}"** and square brackets **"[]"** for data-set / -list / -vector element, row or point indexes and time- / trial- lag, -delay or -shift indices, intrinsic functions: **ABS, SQRT, LN** (natural loraithm, base e)**, LOG** (base 10)**, EXP, SIN, COS, TAN, DEG, RAD, ASIN, ACOS, ATAN** (A = Arc)**, SINH, COSH, TANH** (H = Hyperbolic)**, FACT** (factorial)**, INT, ROUND, FLOOR, CEILING** and **SIGN**, relational / logical functions: **MIN, MAX, IF, NOT, EQ, NE, LE, LT, GE** and **GT** as well as **AND, OR** and **XOR** and special functions: **MNL, MXL** (n-ary minimum and maximum of a list), **CIP, LIP, SIP** and **KIP** (constant, linear, monotonic spline and constrained spline interpolation), **URN, NRN** (uniformly [0.0,1.0] and Normally [-,+] distributed random noise with a default mean of 0.0 and a standard-deviation of 1.0) and **XFCN**'s (extrinsic or externally user-coded single-value functions). The calculation / calc frame can be added anywhere in the IML file as long as the calc-scalar is created before it is to be used in any calc-scalar expression or formula which is typical of a dynamically-typed interpreted language i.e., calc's are solved given its order / sequence of declaration or definition. Note that all trigonometry functions are in radians; to convert from degrees to radians use RAD. Also, although the hyperbolic trigonometric functions SINH, COSH and TANH are supported, they may be expressed as **(EXP(X) – EXP(-X))/2.0, (EXP(X) + EXP(-X))/2.0 and SINH(X) / COSH(X) = (EXP(2.0 * X) – 1.0) / (EXP(2.0 * X) + 1.0)= (EXP(X) – EXP(-X))/(EXP(X) + EXP(-X))** respectively if required. For interest, the TANH() function may be used to approximate the unit-step or Heaviside function as **[~0.0,~1.0] = 0.5 * (TANH(X/K) + 1.0)** where X is any positive (-ve) or negative (-ve) real number and K may be set as K = 0.001, 0.01, 0.1, etc. Another interesting and perhaps more precise approximation of the unit-step function is **MAX(0;X) / ABS(X) = (X + ABS(X)) / (2*ABS(X) + EPSIL) = -MIN(0;-X) / ABS(X) = -(-X + ABS(X)) / (2*ABS(X) + EPSIL)** where EPSIL regularizatizes the approximating function when X approaches zero (0d+0). And the MOD (modulos or remainder) function although not supported, may be formulated or expressed exactly as **MOD(X;Y) = X–Y*INT(X/Y) = X–Y*FLOOR(X/Y)**.

As an example of calc-scalar expression or formula, a well used instance is the straightforward gas or vapor flow compensation which involves density (D, DD), absolute temperature (T, TD), absolute pressure (P, PD) and compressibility (Z, ZD) and its flow meter's, sensor's or instrument's design values i.e., **CF = UCF * SQRT(D/DD * P/PD * TD/T * ZD/Z)** where CF and UCF are the compensated and uncompensated volume flows or flowrates respectively.

Both unary minus and unary plus are supported in any calculation expression although it is recommended not to prefix formulas with a "+" sign in front or before the formula as this is redundant, implied or implicit but is unavoidable for "-" sign. For the functions with two or more arguments i.e., binary and n-ary, IMPL uses semi-colons **(";")** to separate or delimit the arguments where colons **(":")** are used to separate the abscissa(s) for its special interpolation functions (i.e., CIP, LIP, SIP and KIP). These also have corresponding CIP2, LIP2, SIP2 and KIP2 which have two abscissas to calculate a difference between two interpolation curves and are commonly used to compute an incremental or individual composition from a cumulative or aggregated composition profile such as from distillation or distribution curves. The primary reason IMPL employs semi-colons (";") (and colons (":")) for delimiting and demarcating the calculation / calc functions mentioned is due to the fact that IML relies solely on commas (",") to lex or delimit its frame's feature fields and as such using commas in the functions would unfortunately separate the functions into multiple and erroneous fields or expression-parts. Thus semi-colons (and colons) prevent this whereby IMPL Server's internal compiler is fully cognizant of the semi-colon (and colon) demarcation requirement.

It is important to mention that the total size or length for any calculation string is fixed at *4096* characters even when the line continuation character tick ("`", grave accent and ASCII code 96) is used to spread the calculation expression across or over multiple features, lines or rows. That is, a calc-scalar formula or expression string may include multiple lines where the continuation tick must be the last character in the line but the total calculation string size or length including all continued or continuation lines must not exceed a total of *4096* characters. See also the datacalc, datadata, property macro and condition macro frames which also support continuation features, lines or rows to enhance the readability of the IML files.

```
&sCalc,@sValue
CALC1,CEXPRESSION1
CALC2,CEXPRESSION2
CALC3,CEXPRESSION3
…
CALCN,CEXPRESSIONN
&sCalc,@sValue
```

The philosophy or intent of using calculation data or scalar calc's is to minimize as much as possible the use of actual, literal or raw numbers found in the IML file and to reduce the amount of redundant configuration data by supporting user, modeler or analyst local scalar (and vector) memory in the IML

file. By using names (symbols, string indentifiers) instead of explicit numbers (scalars) it is easier to separate the underlying problem's model from the problem's data. In addition, it is also convenient for the user, modeler or analyst to configure calculation expressions where any IML field that expects a number may be replaced by a formula.

Coupled with the built-in and in-line conditional or logical functions or formulas such as IF(), NOT(), …, XOR(), relatively complex and sophisticated rules may be coded, scripted or programmed to provide the user with the necessary capability to pre- and post-program the data. It should be noted that these functions are similar to those found in spreadsheet software such as Microsoft Excel. However, more advanced pre- and post-programming should be relegated to IMPL's IPL (Industrial Programming Library / Language) where its code can be embedded into any computer programming or scripting language though IPL can also configure calculations / calc-scalars easily and conveniently (cf. IPL's IMPLreceiveCalc(), IMPLrunCalc() and IMPLretrieveCalc() routines).

Also note that the concept of rules are distinctly different from constraints in IMPL; only mathematical variables and constraints are known to IMPL during its modeling, presolving and solving process. A (pre-processing) rule can only be applied to the model data and cycle data (problem data) before IMPL models, presolves and solves the problem or sub-problem and not during its solution. A (post-processing) rule can be applied to the solution data after IMPL has been modeled, presolved and solved where rules can be employed to alter or modify the solution data and then IMPL can be re-run in a loop to iterativey or sequentially arrive at multiple good feasible solutions.

For clarity, we describe below the syntax of IMPL's in-line logical, conditional, relational or comparison functions which should always be read starting with the "**if**" prefix as shown below and **X** represents the left-hand-side (L.H.S.) proposition and **Y** represents the right-hand-side (R.H.S.) proposition i.e., **X ~ Y** and ~ represents {=, <=, >=} where the X or Y propositions evaluate to either **true** or **false**:

IF(X) **if X = 0,** then evaluates to **0 (zero, false)** else evaluates to **1 (one, true)**, or
if X /= 0, then evaluates to 1 else evaluates to 0.

NOT(X) **if X /= 0,** then evaluates to 0 else evaluates to 1; opposite/complement of **IF()**, or
if X = 0, then evaluates to 1 else evaluates to 0.

EQ(X;Y) **if X = Y,** then evaluates to 1 else evaluates to 0 when X /= Y
– see also IF(X).

NE(X;Y) **if X /= Y,** then evaluates to 1 else evaluates to 0 when X = Y; opposite/complement of **EQ()**
– see also NOT(X).

LE(X;Y) **if X <= Y,** then evaluates to 1 else evaluates to 0 when strictly X > Y.

LT(X;Y) **if X < Y,** then evaluates to 1 else evaluates to 0 when X >= Y.

GE(X;Y) **if X >= Y,** then evaluates to 1 else evaluates to 0 when strictly X < Y.

GT(X;Y) **if X > Y,** then evaluates to 1 else evaluates to 0 when X <= Y.

AND(X;Y) **if X = 1 and Y = 1,** then evaluates to 1 else evaluates to 0.

OR(X;Y) **if X = 1 or/and Y = 1,** then evaluates to 1 else evaluates to 0.

XOR(X;Y) **if X = 1 or Y = 1 (and not both),** then evaluates to 1 else evaluates to 0.

The "X" and "Y" arguments are the propositions where these logical, conditional, relational or comparison functions above return values that are all real numbers and AND(), OR() and XOR() are essentially Boolean comparison functions as one (1) is solely considered as true / TRUE. The absolute and relative tolerance settings used in the above equality and inequality logical functions or expressions can be found in the IMPL.set settings or options file (cf. **ABSLOGFUNEPS = 1D-5** and **RELLOGFUNEPS = 1D-5**) and may require the user, modeler or analyst to adjust or reset these logical tolerances for their particular circumstances or situations depending on the amount of precision or accuracy required. These tolerances may also be set to EPSIL (1D-13, default) or even zero (0D+0) if necessary to enforce stricter comparison requirements. When the logical functions above are evaluated, both the absolute and the relative logical or conditional expressions or formulas are checked and if one or both of these are met, then the logical expression is evaluated. For example, EQ(X;Y) equals zero (0.0) or false when ABS(X - Y) >= ABSLOGFUNEPS OR ABS(X - Y) >= (ABS(X) + EPSIL) * RELLOGFUNEPS and is one (1.0) or true otherwise. Please note that although these unary and binary logical and relational functions are completely nestable or embeddable e.g., IF(AND(X;Y)), etc., to implement n-ary conditional functions the external or extrinsic functions XFCN()'s may be easily employed. For instance, a n-ary AND2(X1; … ;XN) or OR2(Y1; … ;YM) XFCN() external / extrinsic function may be user-, modeler- or analyst machine-coded in C, C++ or Fortran whereby all or any of the multiple input or supplied arguments respectively need to be true or one (1) for the XFCN() coded AND2() or OR2() to be true (1).

For the random generating functions URN and NRN, the very first instance of the function call or invocation must contain the random seed which must follow the protocol or preferred setting that the seed to be at least 4 * n + 1 where n is the expected maximum number of instances of random numbers that will be generated i.e., URN(4*n+1) and NRN(4*n+1). All subsequent and following instances of URN and NRN should be configured as URN(INNON) and NRN(INNON) where these indicate to IMPL's random number generator not to reset or re-initialize the random seed and to retrieve the next random number in its internal series, signal or sequence. It is typical that random number generators incrementally or recursively, continuously update their random seed value during the random sequencing and hence the requirement that on subsequent calls to URN() / NRN() for the same series, sequene or signal, the random seed must be internally updated in order to compute at least pseudo-randomness. Technically, there is no issue with configuring or setting n to be some exceedingly large and arbitrary 32-bit integer number between one (1) and 2,147,483,648 (2^31) but 4*n+1 above was recommended which as of present is actually not required and is a legacy requirement from an earlier and now deprecated random number generator.

It bears noting that there are a few calculation/calc-scalar names or symbols that should not / never be used as calc-scalar names in IMPL and they are any character and case combination of "eps", "inf" and "nan" (Not-a-Number", NaN). Instead, use "EPSIL" (epsilon, 1D-13), "INFIN" (infinity, 1D+23) and "NNON" (Non-Naturally-Occurring-Number, -99999) respectively as examples. In addition, the naming a calc-scalar "VOID" is also not permitted or allowed as it conflicts with the "VOID" keyword to void or cancel certain UOPSS-QLQP / UQF openings and orders.

It is also worth further describing the use of the paired delimiters parentheses "()", curly braces "{}" and square brackets "[]" in IMPL. Both parentheses "()" and curly braces "{}" are used to primarily group or partition sub-expressions, sub-formulas or terms within an expression or formula involving literal numbers, calc-scalars and/or data-vector-elements. To distinguish a data-vector from a data-vector-element, square brackets are used where inside these brackets a single calc-scalar index expression or formula only may be grouped arbitrarily by parentheses and braces. Unfortunately index expressions / formulas involving other data-vector-elements (i.e., with "[]" square brackets) are not supported in IMPL as the multiple sets of square brackets are not managed nor supported. Finally, to group the various input and output arguments in all of the IMPL intrinsic and extrinsic functions (i.e., ABS(), XFCN(), DATAFCN(), MODELFCN(), WRITEFCN(), REPEAT(), etc., parentheses "()"

are employed exclusively and therefore to group sub-expressions in a right-hand-side (R.H.S.) calc-scalar argument, curly braces "{}" must be used to distinguish it from "()".

IMPL also supports the concept of calc-groups which currently are only useful for user-, modeler- or analyst-coded data functions (DATAFCN's) and outputting, writing or exporting multiple calc-scalars as a set or collection in the OML file given a single calc-group OML configuration statement as well as the user-, modeler- or analyst-programmed model and write functions (MODELFCN's and WRITEFCN's). A calc-group may also be considered conceptually as a data-set or -list i.e., multiple calc-scalars is a data-vector or a compound calc-scalar. A calc-scalar may be assigned to one of more calc-groups and the assignments, attachments, associations or memberships of calc-scalars to a calc-group may be configured over multiple calc-group frames. This makes incrementally assigning, attaching or associating many calc-scalars to the same calc-group using multiple calc-group frames convenient. In addition, calc-groups are also useful to provide what are referred to as index-sets, index-lists or index-vectors i.e., a collection, set, list or vector of indexes, indices or iterators. When calc-groups are outputted, written or exported from OML, they can provide the necessary mapping or indexing of data that can be incorported into for example Microsoft Excel's LOOKUP(), INDEX() and MATCH() formulas when post-mapping the output data into spreadsheets for trending, visualizing, publishing, reporting, etc.

```
&sCalc,&sGroup
CALC1,CALCGROUP1
…      ,CALCGROUP1
CALCN,CALCGROUP1
CALC1,CALCGROUP2
…      ,CALCGROUP2
CALCN,CALCGROUP2
&sCalc,&sGroup
```

To input, read or import text-strings and to output, write, print or export these text-strings as well as configuring them collectively into a text-group, the following two (2) frames are offered. There is no text-processing performed such as lexing and parsing on or with these text-strings and they may be simply received and retained by IML and then retrieved by OML and are considered as constant string data only. The text-string values do not require single- or double-quotes (' ' or " ") before and after the string where both leading and trailing blanks / spaces are removed whereas blanks or spaces between the text characters are retained and are of course considered as part of the text string value. If single-

or double-quotes are placed, then these are simply considered as other text characters where these quotes will be provided or passed through as-is on output. The length, size or number of characters supported for both the text-string names and values is **sixty-four (64)** which is stored in IMPL's catalog resource-entity (cf. **`basestringlen`**). These text-strings are useful to document unit-of-measures, engineering-units or physical number symbols / names, tagnames, other types of labels, indicators, date and time strings of any format, name mapping for system integration, indicating unit-operation-group membership or inclusion, etc.

```
&sText,@sValue
TEXT1,TEXTSTRING1
TEXT2,TEXTSTRING2
…
TEXTN,TEXTSTRINGN
&sText,@sValue
```

```
&sText,&sGroup
TEXT1,TEXTGROUP1
…      ,TEXTGROUP1
TEXTN,TEXTGROUP1
TEXT1,TEXTGROUP2
…      ,TEXTGROUP2
TEXTN,TEXTGROUP2
&sText,&sGroup
```

The calc-list and text-list frames below support or allow duplicate, non-unique and repeating entries or members to be retained in the lists as opposed to calc-groups and text-groups which do not permit non-unique, duplicate or redundant records similar to the concept of a set where a set is a collection of unordered, unchangeable and unindexed items or records. Technically, the calc- / text-lists are internally stored in IMPL's global, common or shared memry as list resource-entities where the roster-enumerator has three (3) keys and opposed to two (2) keys for the calc- / text-groups. The third key is an internally incremented index or indice ensuring that each record-entry or reference-event is unique even if the calc-scalar / text-string name and list name found in the features, rows or lines of the frame are identical to those previously added, inserted or populated calc-scalar / text-string and their respective list names.

```
&sCalc,&sList
CALC1,CALCLIST1
…      ,CALCLIST1
```

```
CALC1,CALCLIST1
CALCN,CALCLIST2
…       ,CALCLIST2
CALCN,CALCLIST2
&sCalc,&sList

&sText,&sList
TEXT1,TEXTLIST1
…       ,TEXTLIST1
TEXT1,TEXTLIST1
TEXTN,TEXTLIST2
…       ,TEXTLIST2
TEXTN,TEXTLIST2
&sText,&sList
```

To configure or register the location of the XFCN user-, modeler- or analyst-coded extrinsic / external (third-party) functions referred to previously, configure the XFCN frame below where if the path name is absent, missing or not preset, then the path name of the IML file is assumed. This frame can be inserted multiple times into the IML file to dynamically re-configure different user-, modeler- or analyst-coded external / extrinsic functions assigned to the same XFCN mnemonic if required. In addition, the XFCN functions can also be used or referenced in any IMPL formula-expression including ILP and INP foreign-files and optimized or estimized but it cannot be dynamically re-configured in the same way calculations can be. That is, for formulas which use variables as its endogenous arguments, XFCN is essentially statically typed whereby its details below may be configured only once.

```
XFCN-@sPath_Name,@sLibrary_Name,@sFunction_Name
DRIVE:\DIRECTORY\,LIBNAME.DLL,FUNCNAME,@
,LIBNAME.DLL,FUNCNAME,@
XFCN-@sPath_Name,@sLibrary_Name,@sFunction_Name
```

IMPL supports XFC1 (9) through XFCN (23) to XFCZ (35) user-coded functions by appending an integer number type from zero (0) to thirty-five (35) for the fourth field as indicated by the “@” or asperand symbol. Index 0 (and 23) is reserved for XFCN and index 1 is for XFC1 and so on where any number outside of this range will default to XFCN with a warning message. This enables thirty-five (35) separate, individual and “state-less” or “memory-less” (i.e., no Fortran “save” attributes or statements for example are allowed) user, adhoc, bespoke or custom functions to be called by IMPL to model complex and even confidential, private and proprietary expressions in machine-code and does not require specifically Intel Fortran for the source code compiling where C and C++ computer programming

languages may also be employed. **Please be aware that the path name field if non-blank, must have as its last character a forward or backslash "/" or "\" indicating that it is a path name prefix. If the path field is blank (""), then the fact's / subject's path name will be substituted accordingly.**

To code this simple external / extrinsic user function XFCN, the single-value return number must be double precision complex (i.e., both its real- and imaginary-parts are double precision) where there are two (2) obligatory input arguments of type long integer (32-bits) ("n") and a vector ("x") of double precision complex of length "n" and starting at index one (1) as follows.

```
complex function XFCN(n: integer,
                      x: complex*n)
```

```
      function xfunc(n,x)
#if stdcalling == 1
cDEC$ ATTRIBUTES DLLEXPORT, STDCALL, REFERENCE, DECORATE, ALIAS : "xfunc" :: XFUNC
#else
cDEC$ ATTRIBUTES DLLEXPORT, ALIAS : "xfunc" :: XFUNC
#endif

        implicit none

        complex(8) :: xfunc

        integer(4), intent(in) :: n
cDEC$ ATTRIBUTES VALUE :: n
        complex(8), intent(in) :: x(1:n)
cDEC$ ATTRIBUTES REFERENCE :: x

c ... Insert developer user, modeler or analyst code here ...
c ...

c ...
c ... Insert developer user, modeler or analyst code here ...

c ... Insert developer user, modeler or analyst code here ...
c ...

c ...
c ... Insert developer user, modeler or analyst code here ...

      end function xfunc
```

IMPL will systematically determine the number of input arguments (both exogenous and endogenous) in the IML file and the ILP and INP foreign-files for each instance of XFCN and pass this number "n" as well as their values "x" to the user-, modeler- or analyst-written external function code. The variables or endogenous arguments in the argument list of the XFCN essentially configure the external / extrinsic formula's or expression's sparsity-pattern.

It should be mentioned that the requirement to use complex numbers especially for formulas allows IMPL to compute first-order derivatives or gradients of near-analytical quality numerically and automatically by the Complex-Step Method (CSM) which is similar to the concept of Dual Numbers Differentiation (DND). In addition, please be aware that all intermediate calculations inside the XFCN source code involving any x(1:n) or x(1..n) endogenous variable elements must also be statically typed as double precision complex in order to retain the propagation of their real- and imaginary-parts during IMPL's first-order derivative computations using the CSM. As a matter of insight and from the perspective of modeling, intermediate calculations are modeled explicitly or in closed-form equations whereas equality and inequality constraints are modeled implicitly in open-form. Formulation-wise, closed-form is when the dependent or output variable is on the left-hand-side only and open-form is when the dependent / output can be on both the left- and right-hand-sides of the equation assignment.

Although IMPL supports upto thirty-five (35) multiple-input single-output XFCN's, it is straightforward to extend this internally inside the XFCN source code by including if-then-else- or case- / select- / switch-statements with specific input arguments as flags, switches or selections. In this way, the same XFCN may be invoked computing a different type of output based on its multiple inputs. For example, one XFCN may calculate both the research and motor octane (RON and MON) bonuses or blending-values of gasoline material but not at the same time, instance or invocation during the modeling and solving operations / processes of IMPL.

Constant Data

IMPL allows dense one-dimensional (1D) arrays or simple data-sets, -lists, -arrays, -sequences, -serieses, -signals or -vectors to be configured which can be used in any scalar calculation and any formula-expression (cf. properties-property, unit-operation conditions and coefficients, etc.). Similar to calc-scalars, a data-set, -list or -vector is simply the binding or mapping of a name, key or symbol to multiple real values indexed or cursored by an integer number indice surrounded by square brackets "[]" and starting or beginning at one (1) and finishing or ending at some four-byte integer number greater than or equal to one (1). They are referred to as constant data as usually these data-vectors represent *independent data* from which related *dependent data*-vectors may be created and computed via calc-scalar expressions or formulas, datacalc and datadata functions described further below. These data-vectors are stored in IMPL's simple-set resource-entity under the fixed and system reserved roster-

enumerator name “ss_DATA” i.e., ss_DATA = 1 with each data-set, -list or -vector representing a separate record-entry or reference-event with a fixed, static or immutable length, size, arity, cardinality or 1D dimenisional rank always starting or beginning at index number one (1) (cf. the simple-set data dump file *.dts). A data-vector is homogenous containing all real values, is expected to have two (2) or more dta-vector-element, row or point values (but having only one (1) is also supported) and is essentially a compound or collection of calc-scalars as the data-vector’s elements, points or rows are considered the same or equivalent. Compared to a spreadsheet, a data-vector is a single column with multiple rows or a single row with multiple columns of real / double precision data. This simple-set resource-entity roster-enumerator, ss_DATA, is defined or declared in IMPL to have a single integer key converted from is data-vector name and multiple / many real values where once a reference-event’s or record-entry’s range-exhibit is registered, defined, declared, configured, specified, etc., it cannot be decreased / shortened nor increased / lengthened i.e., its meta-data or structure is constant or immutable. This is sometimes referred to as an immutablely sized or lengthed data-vector where the smallest or minimalist sized or lengthed data-vector may contain one (1) data-vector element, point or row only i.e., essentially a calc-scalar or trivial data-vector. Fortunately however, the real values of the data-vector’s data-elements, -points or -rows may of course be modified, altered, changed, revised or updated via the data functions found further on in this manual such as the SWAP(), SWAP2() and SUBSTITUTE() data functions.

The individual data-elements / -points / -rows may be indexed and accessed or retrieved using its named column as DNAME[nnn] for example referenceable or indexable in IML by “[nnn]” which is the index or cursor number in the data-vector always starting from one (1) and surrounded by square brackets ([]). This is similar to the C and C++ language arrays also using the square brackets except that IMPL’s data-vectors or 1D-arrays always begin, start or initialize at element, row or point number one (1). Index-expressions or index-formulas are allowed and supported with the requirement that the compiled and computed result of the index-expression/formula must resolve to a valid index number in the proper range of the data-set / -list / -vector. **Unfortunately, no data-vector elements referenced or referred to by square brackets nested or embedded within or inside the other square brackets are supported i.e., DNAME2[DNAME1[99]] is prohibited, forbidden and unsupported.**

The data expressions or formulas (see DEXPRESSION) may contain real numbers, calc-scalar / calculation data symbols / identifiers and any valid calculation expression operator described previously for calc-

scalars in the calculation data. The data-vector names must follow the same naming convention as the calc-scalar names. **Please be advised that although a data-set, -series, -sequence, -signal, -list or -vector 1D-array may be contain only one (1) data-vector-element, -point or -row, to configure a data-vector with the constant data frame below, there must be at least two (2) or more data-points, -rows or -elements per data-vector provided, supplied or specified. Else, the data-vectors will not be properly lexed, parsed and compiled by the IMPL Interfacer() routine which processes the IML files.**

```
&sData,@sValue
DNAME1,DEXPRESSION11
,DEXPRESSION12
,DEXPRESSION13
…
,DEXPRESSION1N
DNAME2,DEXPRESSION21
,DEXPRESSION22
,DEXPRESSION23
…
,DEXPRESSION2N
DNAME3,DEXPRESSION31
,DEXPRESSION32
,DEXPRESSION33
…
,DEXPRESSION3N
```

… OR …

```
DNAME1
,DEXPRESSION11
,DEXPRESSION12
,DEXPRESSION13
…
, DEXPRESSION1N
DNAME2
,DEXPRESSION21
,DEXPRESSION22
,DEXPRESSION23
…
,DEXPRESSION2N
DNAME3
,DEXPRESSION31
,DEXPRESSION32
,DEXPRESSION33
…
,DEXPRESSION3N
&sData,@sValue
```

The constant data frame, with its above two (2) similar data,value tuple formats shown, can be configured to effectively reduce the occurrences or events of redundant input or import independent data in the IML problem or sub-problem data configuration as this data may be accessed anywhere in the rest of the IML file and of course may be referenced and utilized during IMPL's optimization and estimization modeling and solving execution. Also, if any of the DEXPRESSION's are found to be a known data-set / -list / -vector, then all of the contents of the found data-vector are simply inserted into the DNAME data-vector at that point or location. This is useful to splice, join, combine or concatenate together multiple or many known data-vectors consecutively into a single and larger data-vector if required.

It should be noted for interest only that the first data-vector format in the constant data frame above is mimicked after the calculation frame for calc-scalars i.e., CALC1,CEXPRESSION1 is identical to the format of DNAME1,DEXPRESSION11. Hence, the rationale for the constant data frame's formatting which is a multi-value extension of the calc frame without the data-vector name for the second, third, etc. elements, rows or points.

In addition, the constant data frame also accepts or supports in-line multi-column data-vector element / point / row configuration format with multiple or many data expressions being delimited after the comma using the pipe character or symbol ("|", ASCII code 124) i.e., ,DEXPRESSION11 | DEXPRESSION12 | DEXPRESSION13 | … | DEXPRESSION1N where a leading and/or a trailing "|" are optional for the very first or premier and last or ultimate DEXPRESSION. The in-line multi-column delimited data-set / -list / -vector, using the pipe-separated value (PSV) format is convenient for the user, modeler or analyst to read, input, load or import data for each data-vector name as essentially a transpose of the multi-column and multi-data-vector tabular data import or input described immediately below. It should be underscored also that the in-line multi-column data-elements / -points / -rows may also be configured with multiple lines, rows or features as well i.e., ,DEXPRESSION21 | DEXPRESSION22 | DEXPRESSION23 | … | DEXPRESSION2N etc. on separate lines, rows or features. And, the PSV format is useful to easily convert multiple calc-scalars into a single data-set, -list or -vector.

```
&sData,@sValue
DNAME1,DEXPRESSION11 | DEXPRESSION12 | DEXPRESSION13 |
        ,DEXPRESSION14 | DEXPRESSION15 | DEXPRESSION16 |
```

```
        ,DEXPRESSION17 | DEXPRESSION18 | DEXPRESSION19 | … | DEXPRESSION1N |
DNAME2,| DEXPRESSION21 | DEXPRESSION22 | DEXPRESSION23 |
        , DEXPRESSION24 | DEXPRESSION25 | DEXPRESSION26 |
        , DEXPRESSION27 | DEXPRESSION28 | DEXPRESSION29 | … | DEXPRESSION2N
DNAME3,DEXPRESSION31 | DEXPRESSION32 | DEXPRESSION33 |
        ,DEXPRESSION34 | DEXPRESSION35 | DEXPRESSION36 |
        ,DEXPRESSION37 | DEXPRESSION38 | DEXPRESSION39 | … | DEXPRESSION3N |
&sData,@sValue
```

Moreover, if the second field (excluding comments) on the first feature, line, record or row, e.g., DEXPRESSION11, is detected not to be a literal numeric, expression or formula as it cannot be properly compiled, computed and converted to a valid real number, then IMPL automatically assumes that the first line, record, row or feature contains a header feature, row or line of a comma separated set, list, row vector or enumeration of data-set, -list or -vector names and allows for two-dimensional (2D) array, matrix, block, grid, tabular or multi-column input (i.e., multiple data-vectors) of real numbers and/or data formulas / expressions. The rest of the records, lines, rows or features within the constant data frame are assumed to contain the comma separated values (CSV) or data expressions / formulas corresponding to each data-set / -list / -vector / -sequence / -series name found in the first line, row or feature. This is very useful to input spreadsheet-like or table-oriented data after it has been converted to a CSV format given that IMPL does not read, load, input or import data directly or explicitly from any type of spreadsheet or workbook software such as Microsoft Excel nor any relational database table or view using ODBC (Open DataBase Connectivity), OLE (Object Linking and Embedding) or COM (Component Object Model). To integrate or interact with data directly to/from internal memory, IMPL's Industrial Programming Library / Language (IPL) of functions and subroutines may be called from any computer programming or scripting language that can link to dynamic / shared libraries.

```
&sData,@sValue
      DNAME1,      DNAME2,      DNAME3, …
DEXPRESSION11, DEXPRESSION21, DEXPRESSION31, …
DEXPRESSION12, DEXPRESSION22, DEXPRESSION32, …
DEXPRESSION13, DEXPRESSION23, DEXPRESSION33, …
…
DEXPRESSION1N, DEXPRESSION2N, DEXPRESSION3N, …
&sData,@sValue
```

One should note that these multiple column (2D) and multiple data-set / -list / -vector input instances must each be in a separate or individual constant data frame. That is, if there is a requirement for say three (3) instances of 2D multi-data-set / -list / -vector input, then the user, modeler or analyst is

required to configure three (3) separate data frames with its `&sData,@sValue` **leader and trailer features. This is due to the fact that IMPL processes one and only one multi-column 2D data block, grid, matrix or table per constant data frame.**

When a large 2D-array, block, grid, table or matrix of data is to be imported, read, loaded or inputted into IMPL as fast as possible and the data has already been pre-processed i.e., there are no occurrences or events of embedded calc-scalar / data-vector-element related expressions or formulas found in any data frame feature and field, then the IMPL setting or option **IMPORTDATATYPE = 0** may be set. This setting may be changed in the *.set file or modified at any location or place in the IML file using the setting frame defined below. If IMPORTDATATYPE equals one (1, default), then IMPL recognizes and processes each data field, cell or element as a formula-expression whereas when the setting is equal to zero (0), then the data field, cell or element values are read or inputted as literal and real numerical values bypassing or circumventing the compiling and computing overhead of the expression; typically, speedups of two (2) to five (5) times are expected.

Furthermore and similar to the above multi-column and multi-data-vector format, a sparse version of the single-column and multi-data-vector format is also supported where each feeder feature, line, row or record must have the data-vector name as shown below in the very first field, position or column of every feature, row or line. This data frame format can read, input, import or load multiple data-vectors where the size or length of each data-set / -list / -vector are all equal to the total number of non-comment feeder features, rows or lines in between the data frame's leader and trailer features. As mentioned, this is a sparse data import or input where only one (1) non-zero value over all data-vectors is received or inserted per feature, row or line and RNNON is the default for all data-vector-element, -point or -row values i.e., IMPL's missing, absent, not-available or non-existent data value.

It should be highlighted that trivally, if only one data-vector name is encountered during the load, then it simply reads, imports or inputs a single data-vector similar to the aforementioned single-column and single-data-vector forms with the exception of not being able to recognize any embedded data-vectors and only one data-vector is read, imported or loaded within each constant data frame. That is, identical to the multi-column and multi-data-vector form above, each single-column and multi-data-vector formatted instance must also be in a separate constant data frame.

```
&sData,@sValue
```
DNAME1,DEXPRESSION1
DNAME3,DEXPRESSION2
DNAME2,DEXPRESSION3
DNAME2,DEXPRESSION4
DNAME3,DEXPRESSION5
DNAME1,DEXPRESSION6
DNAME1,DEXPRESSION7
DNAME3,DEXPRESSION8
DNAME2,DEXPRESSION9
```
&sData,@sValue
```

DNAME1,DEXPR1|RNNON|RNNON|RNNON|RNNON|DEXPR6|DEXPR7|RNNON|RNNON
DNAME2,RNNON|RNNON|DEXPR3|DEXPR4|RNNON|RNNON|RNNON|RNNON|DEXPR9
DNAME3,RNNON|DEXPR2|RNNON|RNNON|DEXPR5|RNNON|RNNON|DEXPR8|RNNON

After the read or load in this example, the three (3) data-vectors have the above nine (9) values. These data-vectors may then be data processed using for instance the COUNTRANGE() and SCRAPERANGE() datacalc and datadata functions which respect the RNNON as missing-data as do all of the other IMPL intrinsic / internal data functions described in the computation data.

A dense version is also supported and asserts or assumes that each data-set / -list / -vector has the same number of elements, points or rows as the others. The dense version may only be read, imported or inputted using an include file with the keyword DATAFRAME where field six (6) of the include frame's feeder feature configures the number of columns or data-vectors to be loaded – see further below for a description of fields one (1) to five (5) related to the include-file frame (cf. concatentation / companion data). For example, if there are five (5) data-vectors, the number of feeder features in the include file must be a multiple of five (5) given the number of columns or data-vectors specified. The purpose of this dense version is to support situations or cases where an appended OML output file (APPENDFILE = 1) is generated for instance from the frequency / footingsfile flags or a Microsoft DOS batch file and by definition has the same number of data-vector-elements or each data-vector to be imported, inputted or loaded. The dense version format therefore makes reading, importing, inputting or loading the multiple data-vectors in a single file with a single column seamless and straightforward.

In summary for the constant data frames, there are essentially three (3) types of data input formats or forms described above which are: a) multiple single-column and single-data-vector (MSCSDV) form, b) multi-column and multi-data-vector (MCMDV) form and c) sparse and dense single-column and multi-

data-vector (SCMDV) form. The use of the word column represents the data value structure on input and the data-set, -list or -vector is where and how its multiple data values are represented, retained, recorded and stored. The MSCSDV form has three (3) possible format options for import and the MCMDV / SCMDV forms have only one reading or loading format option**. It should also be re-emphasized that for the multi-data-vector forms, their multi- and single-column data values must all be contained inside one constant data frame instance i.e., &sData,@sValue and the dense version of SCMDV must be an include file with keyword DATAFRAME and its sixth (6th) field set to the number of data-sets / -list / -vectors supplied or provided for dense read, import or input. That is, multiple MCMDV and SCMDV cannot be input from the same data frame whereas MSCSDV of course can have multiple data-vectors imported from the same constant data frame.**

At this point it is worthwhile to discuss how to represent date- and time-stamps in IMPL as a data-set, -list or -vector where text-strings may also be used of course except that a text-string is informational only i.e., no data processing / parsing performed. Date- and time-stamps can be represented as a data-vector with seven (7) data-vector-elements i.e**., *yyyy, mm, dd, hh, mm, ss, mss*** which are all real numbers and the hours (hh) are in 24-hour clock-time as there is no element to represent the AM or PM required in 12-hour clock-time. The output, export or writing of the temporal-related data-vector in OML may be provided by COLUMNS = -1 to print the date- / -time-stamp on one line, row, record or feature in the output file. Importing, reading or inputting the temporal-stamp or -tag in IML may use the PSV (pipe separated values) format as the data-vector can be entered on a single feature, row, record or line for convenience and readability.

Other data considered as constants in IMPL are the IMPL.set settings (cf. IMPLreceiveSETTING() routine) which may be modified, updated or changed using the following IML frames. If the IMPL setting symbol specified is not known to IMPL, then a warning message will be output in either the log file or console and the setting of course will not be changed. Any IMPL setting found within the feeder lines, rows or features of the frame will be immediately changed in the IMPL setting's data memory structure allowing the settings frame to be located or placed anywhere in the IML file. Configuring IMPL.set settings into the IML file is useful for commonly used settings such as the global excursion- or penalty-weights i.e., QUANTITYEXCURSIONWEIGHT, LOGICEXCURSIONWEIGHT and QUALITYEXCURSIONWEIGHT which are applied to the lower and upper bounds of variables only (i.e., the key logic / logistics excursions must be set locally in their respective frames). **Please note that it is the responsibility of the user, modeler or**

analyst to properly set / configure these setting parameters in terms of their validity or expected range. Also note that it is not recommended to receive, modify or alter RNNON (and hence INNON and SNNON) as it will cause spurious results and most likely terminate with an exception or crash. RNNON may be changed using the IMPL.set or fact.set / subject.set files or by calling the IMPLreceiveSETTING() routine prior to the IMPLreserve() routine.

```
&sSetting,@rValue
```
SETTING,SEXPRESSION
```
&sSetting,@rValue
```

Solver settings may also be input or set in the IML file with the solver's name and the underscore character ("_") prefixed to the settings found in their respective IMPL.solver file i.e., GUROBI_METHOD. These values will override or overwrite any of the solver settings found in their respective IMPL.solver files.

```
&sSolverSetting,@rValue
```
SOLVER_SETTINGS,SSEXPRESSION
```
&sSolverSetting,@rValue
```

Memory sizes or dimensions defined or declared in the IMPL.mem file or provided in a fact.mem / subject.mem file may also be configured in the IML file to re-size or re-dimension the resource-entities excluding the trivial series-set memory where the memory frame may be placed or located anywhere in the IML file. Also refer to the IMPL setting WRITEMEMORYFILE which can output, print, export or write a scaled, multiplied or factored version of the current or existing memory settings configured for the problem or sub-problem. For the simple-set, symbol-set, catalog, list, parameter and formula resource-entities (data-related) the IMPL Server routine IMPLresize() is called which properly and conveniently retains existing meta-data and data for these resource-entities. For the variables, constraints, derivatives and expressions (model-related) the IMPLrelease() followed immediately by IMPLreserve() are invoked instead as there is no meta-data and data to be retained since the optimization or estimization problem has not been modeled nor solved yet when the IML file is read, imported, inputted or loaded. For example, to increase the dynamic memory allocation for the total number of data-set-, list- or vector-element, -row or -point values, configure a feature, line or row in the memory frame as LENSSVAL,9000000 which is the simple-set (SS) resource-entity's total number of value row-elements.

It is very salient to highlight that IMPL pre-allocates or reserves memory for several internal temporary work 1D-arrays wiv(), wiv2(), wiv3() (4-byte integers), wrv(), wrv2(), wrv3() (8-byte reals of double precision) and wzv(), wzv2(), wzv3() (8-byte real and 8-byte imaginary complexes of double precision) of substantial size which are dimensioned in the appropriate IMPL Server's reserve() routine using the range-exhibit lengths or sizes from the following resource entities i.e., 1 to RANGESS (simple-set) + RANGEL (list) + RANGEP (parameter) + RANGEV (variable) + RANGEC (constraint) + RANGEF (formula). These temporary and state-less work arrays are used extensively in the IMPL data processing whereby to re-size, re-length or re-dimension these scratch vectors, the IMPL.mem and fact. / subject.mem files must be modified accordingly. If the wiv(), …, wzv3() work arrays are too small for the problem or sub-problem at hand, then IMPL will raise or throw a memory access violation exception and crash without any bounds-checking, etc. unless the debug version of IMPL is run.

```
&sMemory,@rValue
```
MEMORY,MEXPRESSION
```
&sMemory,@rValue
```

Please note that in these three setting and memory frames above we use the leader and trailer field `@rValue` instead of `@sValue` to basically distinguish the settings from the calculation and constant data where it is acceptable to assign the setting value using a calc-scalar, a data-vector-element / -point / -row or any expression / formula thereof.

Computation Data

IMPL enables multi-value computation or calculation for any dense, one-dimensional (1D) data-set, -list, -array, -series, -sequence, -signal or -vector of well-known and self-explanatory data aggregation, accumulation, accruing, summarizing and totalizing operations / operators (i.e., a vector to a scalar) as listed which are referred to in IMPL as datacalc's: **LENGTH(), SUM(), SUMABS(), SUMSQ(), PRODUCT(), MINIMUM(), MINIMUMI(), MAXIMUM(), MAXIMUMI(), MEAN()**, **VARIANCE(), MEDIAN(), MODE()** (first found single mode only)**, NORM1(), NORM2(), NORMOO(), POSITIVES()** and **NEGATIVES()** where for interest another well-known aggregation, the total sum-of-squares (TSS, SST), equals the variance times the length or size of the data-set, -list or -vector minus one (1) i.e., TSS, SST = VARIANCE(x) * (LENGTH(x) - 1) where x is any known data-set / -list / -vector.

By IMPL's definition, a datacalc transforms or converts one or more data-vectors into a calc-scalar or a data-vector-element, -row or -point. The data-vector's upper bound (tail, stop) is found as its length, size, arity, cardinality, rank or dimension (and internally known as its range-exhibit) where as mentioned its lower bound (head, start) is always unity or one (1). The data-vector's numerical range is easly calculated as the difference between its maximum and minimum values. The data-set / -list / -vector string name is the only entry permitted within the opening (left) and closing (right) parentheses besides any blank characters i.e., no data-set / -list / -vector expressions must be found and no calc-scalar expression of any kind are supported involving the returned or retrieved resultant value of the datacalc function or operation and the data-set / -list / -vector. If further calculations are required with the datacalc function results, then these must be performed using other calculations / calc-scalar's configured in either a calc or datacalc frame and not involve any of the datacalc functions or operations.

It is important to be aware that all of the aggregation / accumulation functions or operators listed below recognize absent-, nonexistent-, not-available- or missing-data as indicated by IMPL's RNNON (i.e., default RNNON = -99999.0) where it is well established that these aggregation / accumulation calculations ignore, skip or passover these RNNON data values when they occur, are found or encountered in accordance with IMPL's standard missing-data / missing-value protocol.

```
&sDataCalc,@sValue
```
CALC,LENGTH(DATA)
CALC,LENGTH(DATA-GROUP) ***Note that if a data-group, then return the number of data-vectors.**

CALC,SUM(DATA)
CALC,SUMABS(DATA)
CALC,SUMSQ(DATA)
CALC,PRODUCT(DATA)

CALC,MINIMUM(DATA)
CALC,MINIMUM(DATA-GROUP) ***Note that if a data-group, then return the group / global minimum.**
CALC,MINIMUMI(DATA)
CALC,MAXIMUM(DATA)
CALC,MAXIMUM(DATA-GROUP) ***Note that if a data-group, then return the group / global maximum.**
CALC,MAXIMUMI(DATA)

CALC,MEAN(DATA)
CALC,VARIANCE(DATA)
CALC,MEDIAN(DATA)

CALC,MODE(DATA)

CALC,NORM1(DATA) ***See also NORMOE() and NORMIME().**
CALC,NORM2(DATA)
CALC,NORMOO(DATA)

CALC,POSITIVES(DATA) ***Note that zero (0.0) is considered as a positive (+ve) number.**
CALC,NEGATIVES(DATA) ***Note that zero (0.0) is not considered as a negative (-ve) number.**
`&sDataCalc,@sValue`

Note that IMPL also accepts the datacalc frame leader and trailer features to be specified as `&sCalcData,@sValue` as both will be properly recognized in IML though the preference is to use `&sDataCalc,@sValue`. And, similar to the calc frame and the property and condition macro frames, both the datacalc and datadata frames support the tick ("`") continuation special character (grave accent, ASCII code 96) though the total string length is limited to *4096* characters.

The **COUNTRANGE()** and **SUMRANGE()** datacalc functions have three (3) arguments where data-rows / -points / -elements not missing, absent, not-available or non-existent, greater than the real lower bound (user-defined minimum) and less than or equal to the real upper bound (user-, modeler- or analyst-defined maximum) are simply counted or summed respectively and returned. Note that COUNTRANGE() may be used to determine the number of instances, hits, events or occurrences of the data-vector's mode returned by MODE() where mode is the range with the largest count, arity or cardinality of a single value i.e., referring to the real number in the data-set, -list or -vector that is found or encountered most often.

```
&sDataCalc,@sValue
CALC,COUNTRANGE(DATA;LOWER;UPPER)
CALC,SUMRANGE(DATA;LOWER;UPPER)
&sDataCalc,@sValue
```

IMPL also supports data aggregation / accumulation operations or functions involving more than one (1) dense data-vector separated by the semi-colon (";") delimited such as the **DOTPRODUCT()**. The dot-product operator or function is useful to multiply two dense data-vectors together of the same length or size and then to sum, totalize or accumulate their result where any missing, absent, not-available or non-existent data values (RNNON) will tacitly add a zero (0.0) contribution to the dot-product sum. In linear algebra terms, the dot-product operation is identical to x' * y where x' is the transpose of the column vector x multiplied by the column vector y.

```
&sDataCalc,@sValue
```
CALC,DOTPRODUCT(DATA;DATA2)
```
&sDataCalc,@sValue
```

Given a data-vector of initial-values, starting-points, default-results or more appropriately variable-points and an equation, formula or function string, the **EVALUATE()** data calculation functon first compiles then second computes, simulates or evaluates the formula / expression and returns it as a calc-scalar value. The formula string has a maximum character limit of **4096** and IMPL recognizes the variable operands by the upper case "X" with an immediately adjacent 4-byte integer number index having a range or domain of 1..2,147,483,648 (2^31) to the right of the "X" which matches the variable-point's data-vector data-element, -row or -point – see also the foreign-variables. Operands that are not identified as formula variabes ("Xnnn") may be literal real number constants or any recognized calc-scalar or data-vector-element / -point / -row. If for any reason IMPL is unable to evaluate the formula string, then EVALUATE() will return RNNON.

```
&sDataCalc,@sValue
```
CALC,EVALUATE(DATA),FSTRING ***Note that the FSTRING is outside the parentheses.**
```
&sDataCalc,@sValue
```

Interpolating ordinate values or y calc-scalars from ascendingly sorted XX and YY data-vectors is supported using the **SPLINE()** data calculation function and returns RNNON if the type of interpolation is not known or an error occurred when interpolating at the supplied x calc-scalar or abscissa value. The range of values for the spline type are: 0 = constant (CIP, zero-order), 1 = linear (LIP, first-order), 2 = monotonic cubic (SIP) and 3 = constrained (KIP).

```
&sDataCalc,@sValue
```
CALC,SPLINE(DATAXX;DATAYY;TYPE;X)
```
&sDataCalc,@sValue
```

The **NORMOE()** and **NORMIME()** are primarily intended for single-loop PID performance measures of 1-(Manhattan), 2- (Euclidean) and oo- (maximum) norms for its output-error (oe,t = r,t - y,t = ysp,t - y,t) and its input-move-error (or delta-input-error, input usage) (ime,t = du,t = u,t – u,t-1) where the 1- and 2-norms are normalized or standardized by simply dividing by the total number of time- / trial-periods found in the time-series, sequences or signals r,t and u,t and ,t of course represents the time- / trial-period index or indice. However, the NORMOE() and NORMIME() may also be applied to any multiple-

input and multiple-output (MIMO) system output, dependent or controlled variable series, sequence or signal (CV,t) that has a setpoint, target or reference and an input, independent or manipulated variable (MV,t) that is used to control, command, influence or govern one or more CV,t's. The normalization / standardization is essentially to make them comparable across multiple time-series data-sets similar to an average or mean and to be consistent with the oo- / infinity-norm which is not aggregated nor accumulated like the other two (2) making the normalization / standardization essentially average values across the time-series – see also the RETUNEPID() data function. Note that the norm type right-hand-side (R.H.S.) argument is three (3) for the infinity-, maximum- or oo-norm and any of the data-vector-elements, -points or -rows that are missing, non-existent or absent as indicated by RNNON are ignored, skipped or passed-over as usual i.e., simply excluded from the calculations.

```
&sDataCalc,@sValue
CALC,NORMOE(DATAR;DATAY;NORMTYPE)
CALC,NORMIME(DATAU;NORMTYPE)
&sDataCalc,@sValue
```

The **LBQ()** datacalc function computes the well-known Ljung-Box portmanteau Q statistic which sums across the range of auto-correlations starting from one (1) to the number of lags excluding lag zero (0) and calculated from the right-hand-side (R.H.S.) supplied data-set, -list or -vector which returns the L.H.S. calc-scalar Q value. The Q statistic should be less than the Chi-squared (X2) critical or threshold value with degrees-of-freedom (DOF) equal to the number of lags to be considered as pure or pseudo white noise series, signal or innovation sequence. Please refer to the SISOARX(), MLR() and RR() which returns the Q and the X2 statistics at the end of the coefficient or parameter data-set, -list or -vector as well as the CHISQUAREX(ALPHA;DOF) which can be called to compute X2 at a suitable significance-level i.e., alpha = 0.05 for a 95% confidence-interval. Also refer to the XCORRELATION() datadata function which can be used to compute the individual auto-correlations for any time-series data-vector with any number of lags including cross-correlation if the two (2) supplied data-vectors are different. **Please note that if the number of lags argument is negative (-ve), then the mean or average of the data-set, -list or -vector will not be calculated and the data will not be mean-centered internally in the datacalc function.**

And of special note, it should be mentioned that if the auto- / serial-correlation function (ACF) is replaced by the partial auto- / serial-correlation function (PACF), then from Monti, A.C., "A Proposal for Residual Autocorrelation Test in Linear Models", *Biometrika*, (1994), the **MTQ()** data function also

follows a Chi-Squared distribution with the same scaling and degrees-of-freedom as the Ljung-Box portmanteau Q statistic.

```
&sDataCalc,@sValue
```
CALC,LBQ(DATA;LAGS) *** Note that if LAGS is negative (-ve), then data is not mean-centered.**
CALC,MTQ(DATA;LAGS) *** Note that if LAGS is negative (-ve), then data is not mean-centered.**
```
&sDataCalc,@sValue
```

It must be pointed out that although these data aggregation calculations are convenient inside IML, they are intentionally left rather simplistic and limited. For more sophisticated and general types of data calculations, computations, etc., it is recommended to code or script these in a computer programming or scripting language (CPL / CSL) such as C, C++, Fortran, VB, C#, Python, Julia, Matlab, R, Octave, etc. invoking IPL and/or in Intel Fortran using IMPC or even using a spreadsheet calculation environment such as Microsoft Excel and then interfacing or integrating the data i.e., inputting, importing or reading from the IML data frame. Ultimately and as previously mentioned, the data processing capabilities found here are primarily designed for data pre-processing (IML) and data post-processing (OML) and as such have relatively limited capability compared to a CPL / CSL combined with IPL / IMPC (Intel Fortran only) - see also the DATAFCN(), MODELFCN() and the WRITEFCN() external / extrinsic data functions codable in C, C++ or Fortran.

Although there are no data-vector arguments in the three (3) datacalc functions **STUDENTT()**, **CHISQUAREX()** and **FDISTF()**, these datacalc functions compute the well-known Student-t, Chi-Square and F distribution critical or threshold values also known as the upper control limit (UCL) by specifying the significance-level ALPHA as a fraction (i.e., 0.05 for 95% confidence and 0.01 for 99%) and the required number of degrees-of-freedom (DOF and DOF2). For the Student-t critical value, the significance-level fraction may be adjusted according to the Sidak (or Bonferroni) correction or inequality where the corrected or adjusted ALPHA equals 1.0 – (1.0 - ALPHA)^(1.0/DOF2) or approximated as ALPHA / DOF2. If the return values for these functions is RNNON, NaN or Infinity, then an error has occurred in terms of their iterative and direct substitution procedures to derive / invert their statistical distribution values, which is rare, but may occur if ALPHA and/or the DOF's are extreme. These datacalc functions should match or compare well to their respective statistical table values. For example, for the FDISTF() at ALPHA = 0.05 the table values with DOF = 1..20 and DOF2 = oo are: (1) 3.8415 (2) 2.9957 (3) 2.6049 (4) 2.3719 (5) 2.2141 (6) 2.0986 (7) 2.0096 (8) 1.9384 (9) 1.8799 (10) 1.8307 (12) 1.7522 (15) 1.6664 (20) and the computed values from FDISTF(0.05;1..20;500) are: (1) 3.8502 (2)

3.0039 (3) 2.6128 (4) 2.3798 (5) 2.2221 (6) 2.1068 (7) 2.0096 (8) 1.9469 (9) 1.8886 (10) 1.8397 (12) 1.7615 (15) 1.6764 (20) 1.6013.

```
&sDataCalc,@sValue
CALC,STUDENTT(ALPHA;DOF)
CALC,CHISQUAREX(ALPHA;DOF)
CALC,FDISTF(ALPHA;DOF;DOF2)
&sDataCalc,@sValue
```

An interesting shortcut or approximation to the F distribution is X2(DOF)/DOF / (X2(DOF2)/DOF2) which is a normalized ratio of two Chi-Square (X2) statistics where X2(DOF) is the Chi-Square statistic with DOF degrees-of-freedom at the same significance-level ALPHA.

The four (4) **SCAN…()** datacalc functions below return, retrieve or result in a calc-scalar which computes the probability of occurrence for any of the data-vector's rows, points or elements being below or less than or equal to the argumented value with **SCANBELOW()** and being above or greater than or equal to the supplied value via **SCANABOVE()** where these are known as simple range checks. **SCANFREEZE()** and **SCANJUMP()** assume that the data-vector is time- / trial-ordered, seriesed or sequenced as it is in real-time and compute the probability of the rate-of-change (ROC) of the data are frozen i.e., less then or equal to the argumented value and whether the rate-of-change of the data jump are greater than or equal to the supplied value respectively. The definition of probability is simply the counted number of data-elements that are below, above, frozen or jumped outside of the value configured divided by the total number of data-elements minus any missing-, absent-, not-available- or non-existent-data. SCAN…()'s flag argument, if not zero (0), will flag and replace or modify each data-vector's element, point or row with RNNON, conventionally indicating or signaling missing, absent, unavailable or non-existent data, when any of the scan checks are violated. If flag is zero (0), then no data-vector elements will be modified with RNNON. This also implies that the right-hand-side (R.H.S.) data-vector is altered, changed or modified on its return if flag is non-zero.

```
&sDataCalc,@sValue
CALC,SCANBELOW(DATA;VALUE;FLAG)
CALC,SCANABOVE(DATA;VALUE;FLAG)
CALC,SCANFREEZE(DATA;VALUE;FLAG)
CALC,SCANJUMP(DATA;VALUE;FLAG)
&sDataCalc,@sValue
```

The **TEXT2CALC()** datacalc function returns the true argument value if the text-string is known or exists and the text-string value matches or equals the string argument, else the false argument value is returned i.e., if TEXT argument = STRING argument, then return the TRUE argument, else return the FALSE argument. For example, if we have a text-string named or called ACTIVE which has a text-string value of "on" and we have the datacalc statement IS_ACTIVE,TEXT2CALC(ACTIVE;on;1;0), then if the text-string value of ACTIVE equals the TEXT2CALC() string argument value, TEXT2CALC() will receive into the existing / known or non-existing / unknown IS_ACTIVE calc-scalar the real number value of one (1) because the ACTIVE text-string value and the TEXT2CALC() string argument value are equal (true), else if false, then zero (0) would be received into IS_ACTIVE. The TEXT2CALC() datacalc function is useful to convert a text-string value into a calc-scalar value whereby the calc-scalar value can be used in other calc-scalar and data-vector formulas or expressions.

```
&sDataCalc,@sValue
CALC,TEXT2CALC(TEXT;STRING;TRUE;FALSE)
&sDataCalc,@sValue
```

Finally, if no recognized datacalc function is encountered or found on the right-hand-side (R.H.S.), then IMPL will simply compile and compute the R.H.S. as a usual, basic or simple calc-scalar / data-vector-element expression, formula or statement even if the calc-scalar is new or non-existent i.e., not previously known to IMPL. This is convenient for the user, modeler or analyst as they do not need to configure a separate calc frame to process any encountered calc-scalar / data-vector-element expression, formula or statement in the datacalc frame.

And for added convenience available with the datacalc frames, it is also possible to assign, set, modify, update or swap a left-hand-side (L.H.S.) data-vector-element with square brackets and inside its calc-scalar index expression with a right-hand-side (R.H.S.) calc-scalar / data-vector-element formula – see also the datadata frame's SWAP(), SWAP2() and SUBSTITUTE() data functions. In conjunction or coupled with the SPIN"" data formulary discussed below, single data-vector-element / -row / -point recursive equations, recurrence relations or rolling calculations may also be computed within datacalc frames. For instance, the feature, row or line in a datacalc frame of "x[99],x[99-1]+x[99-2]" is supported where the L.H.S. individual data-vector-element with index number 99 is replaced by the R.H.S. expression or formula resulting value which is the sum of its 98th and 97th elements, points or rows; for interest, this recursive equation is known as the Fibonacci series or

sequence. And, refer to the datadata frame's RECURSE() data function to perform recursive or iterative formula calculatons.

There are several basic or rudimentary lookup (find, locate) datacalc functions that convert, transform or map the UOPSS-QLPQ© / UQF© name keys to their internal positive (+ve) index number keys as follows. These are primarily required to configure user-, modeler- or analyst-defined adhoc, bespoke, custom or non-standard formulations configurable in IMPL's scalar- and somewhat set-based and sparsely formed ILP / INP foreign-files. For interest, these number key indexes or indices are created or generated in the IMPL Interfacer() routine via the IML frames and/or their corresponding or matching IPL functions when adding, inserting or populating the units (U's), unit-operations (UO's), unit-operation-port-states (UOPS's), unit-operation-port-state-unit-operation-port-states (UOPSUOPS's), etc.

It should be stated that these UOPSS-QLPQ© / UQF© related lookup datacalc functions along with IMPL's foreign-modeling capability i.e., xX() and fF() to return or retrieve named foreign-variable and -constraint number indexes or indices, can be used to integrate or intermingle standard / non-standard and non-foreign / foreign model interoperability. For example, any standard / non-foreign variable may be included into any non-standard / foreign-constraint but not visa versa i.e., a non-standard / foreign-variable may not be involved in a standard / non-foreign-constraint. However, it is possible to hide, ignore, passover, remove, delete, exclude or not include any standard / non-foreign-constraint and then to re-create, re-generate or re-configure it with standard / non-standard and non-foreign / foreign-variables. It is also possible to hide, etc. any user linear, logic, logical or logistics constraint as these constraints are considered to be IMPL standard / non-foreign-constraints – see the uolUOL() datacalc function. For interest, the concept of hiding a constraint is to simply hide, ignore or passover the constraint from being be known or transferred to the solver but it is always known to the modeler and is found in other algebraic modeling languages (AML's) such as MOSEL (XPRESS).

```
&sDataCalc,@sValue
CALC,uU(UNAME)
CALC,oO(ONAME)
CALC,pP(PNAME)
CALC,sS(SNAME)

CALC,mUO(UNAME;ONAME)
CALC,kPS(PNAME;SNAME)
```

CALC,ijUOPS(UNAME;ONAME;PNAME;SNAME)
CALC,jiUOPSUOPS(UNAME;ONAME;PNAME;SNAME;UNAME2;ONAME2;PNAME2;SNAME2)

CALC,rgUOG(RGNAME)
CALC,sgUOG(UNAME;OGNAME)
CALC,wUOGOG(UNAME;OGNAME;OGNAME2)
CALC,qgUOPSG(QGNAME)
CALC,pgUOPSUOPSG(PGNAME)

CALC,hH(HNAME)
CALC,gG(GNAME)
CALC,eE(ENAME)
CALC,dD(DNAME)
CALC,cC(CNAME)
CALC,bB(BNAME)
CALC,aA(ANAME)
CALC,lL(LNAME)

CALC,uolUOL(UNAME;ONAME;UOLNAME)

CALC,xX(XSUBNAME;NUMBER;SUPREMUM) *** Note SUPREMUM is required for leading-zero spacing**
CALC,fF(FSUBNAME;NUMBER;SUPREMUM) *** Note if NUMBER=0, no NUMBER is appended/suffixed**
`&sDataCalc,@sValue`

While there are no data-vector arguments in the datacalc function **PRIMEN()** below, this datacalc function simply returns the first prime number found within the lower and upper bound range provided or supplied by its arguments. If a prime number cannot be found given the specified range, then RNNON is returned instead.

```
&sDataCalc,@sValue
CALC,PRIMEN(LOWER;UPPER)
&sDataCalc,@sValue
```

Before we proceed to the datadata functions, which return a data-set, -list or -vector (i.e., a 1D-array) of left-hand-side (L.H.S.) multiple data-elements, -points or -rows, the **SETTING()**, **SIGNAL()** and **STATISTIC()** datacalc functions simply return or retrieve the IMPL setting, signal real value such as USELOGFILE, RNNON, EPSIL, INIFIN, NUMLAGS, etc. when the setting / signal or statistic name is known (cf. IMPL.set, the IML setting frame and the IMPL.hdr file). Else if the R.H.S. setting / signal argument name is unknown, then RNNON is returned and an IML error is raised. These two (2) datacalc functions are useful to retrieve or get any IMPL settings, signals and statistics inside the IML file.

```
&sDataCalc,@sValue
```
CALC,SETTING(NAME)
CALC,SIGNAL(NAME)
CALC,STATISTIC(NAME)
```
&sDataCalc,@sValue
```

Further to the above data frame (constant data) and datacalc frame (scalar data computation), IMPL supports the capability to populate, manipulate and compute dense data-sets, -lists, -signals, -series, -sequences or -vectors as non-recursive, non-recurring or non-rolling functions, relations, formulas or expressions of other data-vectors and/or calc-scalars using essentially vector arithmetic, whole array processing, vector processing or implicit / internal for- / do-loops. **Please be aware that all data-vectors found, involved or included in the data expression or formula, must be of equal size / length or greater else an error / exception will occur.** It is salient to highlight that absent-, nonexistent-, not-available- or missing-data are also conveniently handled here whereby any expression involving a data-element / -point / -row value with a missing-, absent-, unavailable- or non-existent-value, as indicated by the IMPL setting RNNON for a real non-naturally occurring number, will simply have its left-hand-side (L.H.S.) 1D data-array element value also set or imputed as a missing- / not-available- / incomplete- / absent-value (RNNON). These data-set, -list or -vector expressions simply apply the IMPL lexed, parsed and compiled calc-scalar calculation expression multiple or many times over the number of data-elements / -points / -rows found in the involved or included data-vectors**.**

From an arithmetic sequencing perspective, the data formulas supported here must be in closed, explicit, non-recurring or non-recursive form as opposed to open, implicit, recurring, rolling or recursive form. The primary reason only non-recursive / non-recurrent / non-rolling data formulas are allowed is due to the fact that no data-vector-elements can be referenced using the square brackets [] containing the element index numbers or element index expressions. However, please refer to the RECURSE() data function which does supports recursive, recurring or rolling calculations as well as square brackets for any data-vector-elements, -points or -rows.

And somewhat similar to the datacalc frame, if a recognized known, existing or previously configured left-hand-side (L.H.S.) calc-scalar is encountered or found in the datadata frame, then IMPL will compile and compute the R.H.S. simply as a usual calc-scalar / data-vector-element expression or formula and not as a data-vector. It is extremely convenient for both the datacalc and the datadata frames to support calc-scalar computing when it is necessary for instance to incrementally update a

calc-scalar value in a datacalc or datadata frame without having to explicitly configure a separate calc frame each time there needs to be a modification or update to a calc-scalar inside a series or sequence of datacalc and datadata frame statements or features.

```
&sDataData,@sValue
DATA,DEXPRESSION1
DATA,DEXPRESSION2
…
DATA,DEXPRESSIONN
&sDataData,@sValue
```

In addition, IMPL supports several handy and useful data (datadata) functions or data operators described further below. These data functions are essentially keywords with accompanying parentheses ("(" and ")") to indicate and contain its calc-scalar and/or data-vector arguments. Only calc-scalar and data-vector-element (cf ."[" and "]") arguments will be compiled and computed as valid argument expressions or formulas. Inside these expressions or formulas, it is important to use curly braces "{" and "}" to group or partition any expression / formula terms as the parentheses "(" and ")" are reserved for the data function's opening and closing delimiters. **Furthermore, data-vector expression arguments are not directly compilable and computable and the data functions themselves cannot be nested, embedded nor involved in any formula-expression in anyway. Hence, these data functions in both the datacalc and datadata frames are really data subroutines and should be considered as such given that they cannot be included or nested in data formulas or expressions where the left-hand-side (L.H.S.) and right-hand-side (R.H.S.) of the statement is really a subroutine call statement re-organized as a function invocation or call.**

The **DATETIMENOW(), DATENOW()** and **TIMENOW()** functions have no arguments and simply return the current system date and time as data-sets / -lists / -vectors with the following real number data-elements / -points / -rows of [1] = yyyy, [2] = mm, [3] = dd, [4] = hh, [5] = mm, [6] = ss and [7] = mss representing the underlying date and time two (2), three (3) and four (4) digit integer numbers.

The **SCATTER()** function simply scatters, spreads, copies or repeats any real value across the length, size or dimension configured for a new / unknown or existing / known data-set, -list or -vector and is useful to initialize, create or generate a data-vector when required. **However, if the left-hand-side (L.H.S.) data-vector exists or is known and the length argument is negative (-ve), then simply ignore, passover**

or skip scattering or spreading the value across its elements, points or rows. This is helpful when first initializing, starting or beginning a data-set, -list or -vector with the SCATTER() data function but subsequently it is initialized via a data frame most likely from some subsequent and iterative run or execution. The negative (-ve) length argument signals or indicates to not scatter, spread, etc. any values.

The **SYNTHESIZE()** function synthesizes, creates or generates multiple data-set / -list / -vectors using the left-hand-side data name as the root, base or prefix character string starting from the lower integer number (>= 0) and stopping at the upper integer number (<= 2^31 = 2,147,483,648) arguments as suffixes and by default incrementing by one (1). This provides an incremental and contiguous / consecutive number of data-vectors synthesized or generated with their initial, starting or default value and length arguments identical to SCATTER()'s. And, if the stride or step optional argument is present and not INNON, then SYNTHESIZE() generates or creates data-vectors with a user, modeler or analyst index incrementing and with our leading-zeros index numbering convention identical to the SPIN"" data formulary and the ILP / INP foreign-files' REPEAT() and REPLICATE"" model function and formulary. **As well and similar to the data-group and the SPIN"" data formulary, if the upper (stop, tail) is less than (<) the lower (start, head) sub-iterator, then the data-vector numbering is skipped, ignored or passed-over.**

The **SEQUENT()** function multiplies the data-element's / data-point's / data-row's internal series or sequential index number, always starting from IMPL's standard of one (1), by a specified or configured real value (i.e., a coefficient, factor or multiplier). Obviously if the real value specified is one (1.0), then SEQUENT() simply returns its internal sequence or series index number which is useful to externalize a sequenced or serially index number data-set, -list or -vector. And, if the right-hand-side (R.H.S.) size or length argument is negative (-ve), then the sequence arrangement or ordering is simply reversed i.e., in decreasing or descending order instead of the default (+ve) ascending or increasing order.

The **SLIDE()** data function simply slides the data-vector to the very left, top or front by moving the data forward as specified by the number of data-vector-elements, -rows, or -points i.e., SLIDE() returns the rest or remaining data starting at index or indice number one (1). If the number is zero (0), then SLIDE() trivially returns the same right-hand-side data-vector into the left-hand-side (L.H.S.) data-set, -list or -vector i.e., a copy or duplicate. Currently only a positive (+ve) number argument is supported to slide

the data to the very left, top or front, but in the future it is possible for the data to slide to the very right, bottom or back of the data-vector if the number argument is negative (-ve).

The **SLICE()** data function slices, splits, cuts or extracts the data-set / -list / -vector into a contiguous or consecutive subset / sublist / subvector / subarray using the lower and upper index bound arguments specified between IML's standard semi-colon ";" argument delimiter as mentioned previously, and as expected, the minimum number of data-elements, -rows or -points in the sliced data-vector subset is one (1) i.e., a data-vector with only one (1) element, row or point is supported. SLICE() may also be used for example to time- or trial-shift a time-series / trial-sequence of data up or down where the lower and upper bound arguments provide the required positive (+ve) or negative (-ve) lagging or shifting. See also the SHIFT() data function and the SEQUENCE() data function which can effectively slice any data-vector into an arbitrary arrangement, order, sequence or series. **Please be advised that if the lower index / indice bound exceeds the upper bound of the right-hand-side (R.H.S.) data-set, -list or -vector, the SLICE() command simply defaults the returned or retrieved L.H.S. data-vector to that of the R.H.S. data-vector.**

The **SPLICE()** function combines, joins, augments, merges or concatenates only two (2) data-sets, -lists or -vectors together to form a larger dense data-vector where the `&sData,@sValue` data frame may also be configured to do the same except that the data frame allows for more than two (2) data-vectors to be concatentated, joined or spliced with different data-field formats (cf. "," and "|"). As well, refer to the STACK() data function which combines or stacks together multiple data-vectors found or contained in a known data-group in their defined membership arrangement, order or sequence. The SPLICE() datadata function is useful when there is a need to top or bottom append (prefix or suffix) one data-set, -list or -vector with or to another.

The **SHED()** function sheds, replaces, modifies, updates, swaps, switches or substitutes any data-vector-elements in its R.H.S. data-vector with a specified or configured default imputation value if any of its values are less than the lower input argument or greater than upper argument, and returns or retrieves the entire data-vector back into the L.H.S.'s data-set / -list / -vector. The SHED() function is handy to swap, substitute or switch any of its values that are too small or too large and alter, modify or update them with the user, modeler or analyst supplied input argument for the imputation value which of course could be RNNON if required. Another possible use for the SHED() function is to shed data-vector-

element values with a supplied imputation value by specifying the appropriate lower and upper bounds to saturate, clip or clamp certain values to lie at some minimum or maximum limit. The SHED() data function may also be called successively to update a single L.H.S. data-vector multiple times as the R.H.S. and L.H.S. can be the same data-vector**. Please note that SHED() by default, checks for NaN and +/- Infinity (i.e., inf, Inf) and if detected, it then sheds, replaces, modifies, etc. its NaN / Infinity value with the SHED() imputation value R.H.S. argument. This approach assumes that any NaN or Infinity value is by default outside the supplied lower and upper bounds. According to IEEE 754, NaN and Infinity are floating point numbers after calculations such as 1.0 / 0.0, LOG(0.0) and SQRT(-1.0) which are undefined and cannot be represented accurately.**

The **SKIP()** function skips, ignores or passes over a number of data-points, -rows or -elements within a data-vector where if the number is zero (0), then SKIP() simply returns an exact copy of the right-hand-side data-vector. Skipping data-vector rows, points or elements is more suited to time-series, time-sequenced, time-sampled or trial-spaced data-sets, -lists or -vectors so as to ignore or passover in between data elements and typically used to increase the sampling time-step, -instant, -interval or -period for example. More specifically, this happens when a data-signal, -sequence, -string or -stream is over-sampled with a higher than required sampling frequency and the sampling-interval / -instant or cycle-time can be increased or lengthened by skipping uniformly over one or more data-points, -rows or -elements which may be considered as a simple re-sampling or re-sequencing procedure, operation or function. The SKIP() data function may also be employed when a data-vector contains data-points, -rows or -elements from essentially multiple data-vectors that have been combined together in some way such as being stretched or even stacked similar to an unfolded or linearized matrix (2D-array) transformed into a vector (1D-array). This can occur for example when an OML output file is appended by multiple successive or consecutive runs of the IMPL Console program.

The **SHIFT()** data function time- or trial-shifts any data-vector by lagging, backward-shifting or shifting down the right-hand-side data-vector which replaces or imputes the top or head data-vector-elements with a value specified by the user, modeler or analyst. However, if the shift index is negative (-ve), then the bottom or tail data-vector-elements will be replaced or imputed by the supplied value and this is a lead instead of a lag i.e., shifting up for a lead and shifting down for a lag. For example, if the lag is five (5), then there will be 5 imputed values found at the top of the data-vector i.e., data-vector-elements 1..5 and if the lag is -5, or equivalently the lead is 5, then the very last 5 data-elements at the bottom

will be the imputed value. The primary purpose or intent of the SHIFT() data function is to perform dynamic linear model estimation using multiple linear regression (MLR()) and principal component analysis and regression (PCA() then MLR()) which is based on time-lagging or backward-shifting independent regressor, explanatory or predictor variables.

The **STACK()** and **STRETCH()** data functions simply repeats or replicates the entire right-hand-side's (R.H.S.'s) data-set, -list or -vector multiple times (STACK()) as specified by the length argument and repeats or replicates each element, row or point of the R.H.S. data-vector (STRETCH()) respectively and returns the left-hand-side data-vector which is of course expanded or elongated. And, if the R.H.S. data-vector argument is found to be a known data-group, then its length argument is ignored and each data-set / -list / -vector member is unfolded, linearized, heaped, piled or stacked one after the other in the exact order or sequence encountered in the data-group starting or beginning membership from the first data-vector discovered i.e., the very first data-vector member assigned, associated or attached to the data-group has its data-vector-elements, -rows or -points at the very top of the stacked data-vector. This is especially helpful when interfacing, interacting or integrating data with the external / extrinsic data functions (DATAFCN) and many data-vectors are required as input where the output or to-be returned data-vector can also be stacked and then easily destacked using the SLICE() data function when required.

The **SWAP()** data function simply swaps, switches, modifies, replaces, revises, updates, substitutes, etc. a single specified value at a specified integer index or indice into or inside the existing data-vector. The X,SWAP(xv;ix) is very efficient as it surgically receives a singleton real value into the specified data-vector's data-element, data-row or data-point using the index number argument i.e., X(ix) = xv where both xv and ix are the right-hand-side arguments respectively and can be calc-scalar/data-vector-element formulas. Note that the lower case xv and ix are used to indicate that these are calc-scalars or data-vector-elements and not data-vectors; see the SWAP2() below.

The **SWAP2()** data function suffixes or appends as many data-set, -list or -vector-elements / -rows / -points as possible starting, beginning or opening at the swap or suffix index specified relating to the left-hand-side data-set, -list or -vector. SWAP2(XV;ix), where XV is a data-vector and ix is a calc-scalar index. SWAP2() is similar of course to the SWAP() data function which swaps, modifies or replaces a single value at the index or indice supplied whereas SWAP2() suffixes or appends multiple values (XV) starting

from the swap or suffix index (ix) configured. **However, it is relevant to be aware that the left-hand-side (L.H.S.) data-set, -list- or -vector must be first initialized to the total, complete or entire size or length expected as the SWAP2() data function only modifies, replaces or swaps the right-hand-side (R.H.S.) data-vector-elements / -rows / -points starting, opening or beginning from the swap or suffix index or indice and does not literally suffix or append new or extra data elements, rows or points to the L.H.S. data-set, -list or -vector as the user, modeler or analyst may recall that a data-vector's size, length, arity or cardinality is immutable, fixed or constant.**

The **SUBSTITUTE()** data function returns the substituted, replaced, revised, adapted, modified or updated real values for the left-hand-side (L.H.S.) known / existing data-vector with the real values found in the first right-hand-side (R.H.S.) data-vector argument using the index numbers, indices or iterators supplied by the second R.H.S. data-vector argument where these integer indexes reference the L.H.S. data-vector. The X,SUBSTITUTE(XV;IX) statement is also a multi-value version of SWAP() similar to SWAP2() and simply substitutes any data-vector's element, row or point according to the following statement X(IX) = XV. IX is the second argument data-vector containing the proper indexes or indices in X truncated internally to be integer whole numbers, XV is the first argument data-vector of real values with a size or length greater than or equal to the length of XI and less than or equal to the size of X where X is the known L.H.S. data-vector with its selected real values substituted, swapped, replaced, revised, etc. with XV. **And take note, if any of the R.H.S. index numbers (IX) are missing, absent, not-available or non-existent as indicated or signaled by IMPL's Non-Naturally-Occurring-Number (e.g., INNON), then these indices are simply ignored, skipped or passed-over consistent with IMPL's missing-data handling protocol.**

The **SHUFFLE()** data function simply shuffles or permutates the right-hand-side (R.H.S.) data-set's, -list's or -vector's rows, points or elements randomly given a random seed and returns the results into the left-hand-side (L.H.S.) data-vector. This is the same random rearranging, re-ordering or re-sequencing method / algorithm found in the KMS() datadata function and the IMPL Solvers' IMPLshuffleability() routine. Deterministically speaking, if you shuffle or permutate the same data-vector with the same seed, SHUFFLE() will return the same L.H.S. data-vector result. This implies that you can permute multiple (parallel) data-vectors together with the same seed separately i.e., using multiple SHUFFLE() calls, invocations or instances, and they will all be rearranged consistently as if they were shuffled together or en bloc.

The **SCRAPERANGE()** data function returns or retrieves a data-vector of index numbers, indices, iterators or cursor values given an input data-vector and its real lower and upper bound values defining a specific data range where missing, absent, unavailable or non-existent data-vector-elements, rows or points, as indicated by RNNON, are ignored, skipped or passed-over as usual. With SCRAPERANGE(), a subset of data-vector-element row indexes or indices is retrieved from the input data-vector within the real lower and upper bound range i.e., **lower bound < data-vector-element value <= upper bound** where the COUNTRANGE() datacalc function returns the count, arity or cardinality of indices or index numbers for the subset data-vector. If there are no data-vector-element values within the specified range, then IMPL will return RNNON indicating a missing, absent, not-available or non-existent index number value. SCRAPERANGE() may be called or invoked to find for example outliers, anomalies, aberrants, etc. in an input data-vector and return or retrieve the subset of indexes or indices whose data-vector-element / -point / -row values are within or between the **lower (greater than, >) and upper (less than or equal, <=) bounds** of the range or domain configured. *There is an optional flag argument that if zero (0, default) will retrieve or return the indexes, indices or cursors with values found with inside the range, else if the flag is one (1), then indexes, indices or cursors correspond to the values outside the lower and upper range specified or supplied.*

The **STARTRANGE(), SPANRANGE()** and **STOPRANGE()** data functions each retrieve / return a data-vector of index numbers, indices, iterators or cursor values given an input or right-hand-side data-vector and its real lower and upper bound arguments defining a specific data range where missing, absent, unavailable or non-existent data are skipped, passed-over or ignored identical to the SCRAPERANGE() function. The STARTRANGE() function returns the index number, indice or cursor value of the start, begin or initial of each span interval, window or time- / trial-plank where a time-/trial-plank spans a certain number of contiguous / continuous / consecutive data-vector-elements, -rows or -points.. The SPANRANGE() returns the number, count or cardinality of index numbers for each span interval, window or time- / trial-plank including the start and stop events or occurrences similar to a campaign, run-length or uptime of setup logics for a continuous-process and the STOPRANGE() provides the index / indice number of the stop, end or final index number. If no span intervals, windows or time-/trial-planks are found, then a single element, point or row data-vector is returned with value RNNON. Technically, one (1) of the three (3) are redundant or dependent on the other two (2) data-vector results i.e., **dvstart[i] =**

dvstop[i] - dvspan[i] + 1, dvspan[i] = dvstop[i] - dvstart[i] + 1 and **dvstop[i] = dvstart[i] + dvspan[i] - 1** where the **dvstart** data-vector is returned by the STARTRANGE() data function for example.

The **SORT()** and SEQEUNCE() functions typically operate in tandem where SORT() may be utilized for instance to find the median of the data-vector and SEQUENCE() is used to pick, choose or select particular data-rows / -points / -elements. SORT() sorts, orders or ranks the argumented data-vector in ascending, increasing or non-decreasing order using a comparison quick sorting method (cf. sortrx() in the IMPC Manual) but only returrns the sorted permutation index numbers, indices or iterators (ranking) where the argumented data-vector is unchanged / unaltered / unmodified. To sort in descending, decreasing or non-increasing order, the user, modeler or analyst must negate the data-vector row, element or point values by pre-multiplying by minus one (-1.0).

The data function **SEQUENCE()** sequences, permutates or re-indexes the data-set, -list or -vector of real values in the first data-vector argument using the second data-vector argument which contains the sorted or permutation index numbers typically returned or retrieved from SORT() but not necessarily (cf. SCAPRERANGE(), etc.). The returned, retrieved or resulting or left-hand-side (L.H.S.) data-vector from SEQUENCE() contains the sequenced (or sorted) data-vector of real values in ascending order of at least the size or length of the right-hand-side (R.H.S.) permutation index-set / -list / -vector. For example, if X is the data-vector of 1 to N numbers to be sorted and IX is the returned permutation integer index data-vector from SORT(), then the IML statement XS,SEQUENCE(X;IX) is equivalent to XS = X(IX) in any computer programming or scripting language that supports vector or whole-array processing where XS(1) or X(IX(1)) is the smallest value of X and XS(N) or X(IX(N)) is the largest value of X(1:N), X(1..N) or {X(1),…,X(N)}.

Of particular note, the SEQUENCE() data function is also helpful to return and retrieve as output a subset, sub-list or sub-vector of its first data-vector argument where its second data-vector argument provides the sub-index-set, sub-index-list or sub-index-vector which can be likened to a noncontiguous or non-consecutive slice, series or sequence. For instance, if X is the first data-vector of length N and IX is the second data-vector of smaller size n where n < N, then SEQUENCE() outputs, retrieves or returns X(IX(1:n)), X(IX(1..n)) or {X(IX(1),…,X(IX(n))}. This means or implies that the SEQUENCE() data function also acts or performs like a SUBSET() data function.

The **DESORT()** data function returns the reverse-sorted, inverse-sorted, unsorted or desorted integer index data-set / -list / -vector given the sorted permutation index data-vector which enables the sorted real data-vector to be reverse-sorted, inverse-sorted, unsorted or desorted back to its original unsorted data-vector. The DESORT() function simply desorts the sorted permutation index-vector according to the following statement KX(IX) = IX(KX) = JX where IX is the sorted permutation index data-vector of X after calling SORT(X), JX = 1..N = 1:N = {1,…,N} which is internally generated and KX is the returned or resulting data-vector of index numbers i.e., X = XS(KX) = X(IX(KX)) = X(KX(IX)) where XS = X(IX).

The **SPLINES()** datadata function is identical to the datacalc function SPLINE() except that SPLINES() returns a data-set, -list or -vector of dependent unsorted Y values on the left-hand-side given the data-vector of independent unsorted X values as the last argument on the right-hand-side. The SPLINES() function is simply a multi-value version of SPLINE() when it is required to interpolate using the same abscissa and ordinate sorted X and Y values multiple dependent (ordinate) values from multiple independent (abscissa) values. This is useful to distribute, discretize or digitize an ASTM distillation curve into a pre-defined set, list or vector of distillation temperatures to compute its cumulative evaporation cut or component volume- or weight-based compositions. This is also helpful to transform temporally ordered or sorted transactional data such as orders, commands, provisos, etc. into time-series time-ordered data i.e., transient, time-varying, time-variant or dynamic target / setpoint changes. When the interpolation or spline type is set to zero (0), constant or zero-order interpolation is performed producing a zero-order-hold, step or stairs type of time-series, -sequence or -signal.

It is important to highlight that with time-ordered, time-indexed, time-sorted, time-signal and time-series data-sets, -lists or -vectors, the precedence, sequence, rank or sort-order of data must be from older (top) to newer (bottom) data where data[1] is the oldest (i.e., at the top of the list) and data[nnn] is the newest (i.e., at the bottom of the list). This time- / trial-ordering or sorting is consistent and standard with IMPL's past, present and future time-horizon or -profile where older data is the past / present and newer data is in the future i.e., -NTP+1, …, 0, 1, …, NTF and NTP is the number of time-periods in the past / present and NTF is the number of time-periods in the future. This convention of older then newer data is also consistent with appending or suffixing a data file incrementally when new data becomes available in for example when data logging or recording using flat-files. However, if the data-vector is in reverse or inverse order / sequence i.e., newer preceding older data, then the SORT() data function may be called or invoked accordingly.

The **ACCUMULATE()** function performs a straightforward cumulative sum or incremental aggregation of the data-set / - list / -vector and returns the resulting data-vector sometimes referred to as a rolling, cumulative or moving sum or total. For instance, this data function can be used to convert or transform a finite impulse-response (FIR) to a finite step-response (FSR) i.e., FSR[t] = FSR[t-1] + FIR[t] where t is the data-vector-element integer index number in the range, domain, spread or span of 1..NT. The ACCUMULATE() data function is a simple form of a recursive function or recurrence relation as it adds the previously accumulated data-vector-element value (FSR[t-1]) to the current data-vector-element value (FIR[t]) and so on.

The **DIFFERENCE()** function performs the well-known one (1) or two (2) degree of differencing found in time-series analysis i.e., data[t] – data[t-1] or data[t] – 2 * data[t-1] + data[t-2] returning the same length data-set as the argument data-set where the first and second data-points are zero (0D+0) respectively and any missing-, absent-, not-available- or non-existent-data also returns zero (0D+0) for the differenced value. Adequate differencing is generally determined when the auto- and cross-correlation functions dampen out quickly. If the degree of differencing is set to zero (0), then DIFFERENCE() will simply retrieve or result in the data-vector being returned as-is. Although the backwards differencing operation is not recursive nor recurrent, it does reference individual data-vector-elements, rows or points via the backward shift operator, as found in the square brackets [], and therefore requires an explicit data function. For interest, the first- and second-order difference operator may also be computed by the RECURSE() data function or the with the datacalc from and its SPIN”” data formulary.

The **XCORRELATION()** function computes the cross-correlation coefficients for the first data-set argument (output) with respect to the second data-set argument (input) given the specified number of lags in the past and respecting missing- / absent- / unavailable /- non-existent-data. The XCORRELATION() will also compute the auto-correlation coefficients when both of the argument data-sets are the same. It is well-known (Bartlett’s test) that the 95% confidence-interval of a correlation coefficient is approximated as 2.0 * 1.0 / (n – k)^0.5 where 2.0 is the nominal Student-t 95% threshold or critical value, n is the number of data-points, k is the number of lags and (n – k) is sometimes called the number of degrees-of-freedom. Please take note that the returned data-set / -list / -vector contains 1..k+1 data-elements where the first (1st) data-element is defined as the cross-correlation coefficient at

lag zero (0) and the last cross-correlation coefficient is at lag k. One should also note that although the partial auto-correlation function is available in IMPL cf. PCORRELATION(), our intent is generally not to explicitly identify the stochastic noise model i.e., the auto-regressive integrated moving average (ARIMA) structure, but to include some useful form (over-parameterization) of the stochastic noise disturbance sub-model when estimating the deterministic process transfer function sub-models i.e., first-order and second-order plus dead-time (FOPDT, SOPDT and the general TFPDT). **Please note that if the number of lags argument is negative (-ve), then the mean or average of the data-set, -list or -vector will not be calculated and the data will not be mean-centered internally in the datadata function similar to the LBQ() and MTQ() datacalc functions.**

The partial auto-correlation **PCORRELATION()** data function is an identification technique to help determine the parsimonious or canonical number of lags in the auto-regressive (AR) part of the stochastic noise transfer function. PCORRELATION() is somewhat similar to XCORRELATION() to compute the partial auto-correlation (PACF) coefficients using the Durbin-Levinson algorithm which vary according to the 95% confidence-interval of +/- 2.0 * 1 / n^0.5 where 1 / n is the variance of each PACF element. As mentioned, the PACF of size or length 1..k is primarily helpful to aid in the identification of the denominator degree of the auto-regressive (AR) stochastic noise model. **Please note that if the number of lags argument is negative (-ve), then the mean or average of the data-set, -list or -vector will not be calculated and the data will not be mean-centered internally in the datadata function similar to the LBQ() and MTQ() datacalc functions.**

In order to compensate for significant auto-correlation or serial-dependency (i.e., signal de-serialization) contained in an input / independent time-series and an output / dependent time-series, the **PREWHITEN()** and **PREFILTER()** functions may be called before XCORRELATION() and PCORRELATION() which is a well-known pre-processing technique found in systems identification and time-series analysis. The returned left-hand-side (L.H.S.) data-vector contains the prewhitened or prefiltered data where missing-, absent-, unavailable- or non-existent-data values (RNNON) may be present. The number of lags argument associated with the PREWHITEN() operation specifies the number of coefficients in the typically “over-parameterized” auto-regressive (AR) stochastic noise sub-model which is an established approach to prewhiten an input-series before cross-correlating it with an output-series. The polynomial form of the auto-regressive (AR) prewhitening transfer function is *AR(z^-1) = 1 + ar1 * z^-1 + ar2 * z^-2 + … + ark * z^-k* where *k* is the number of lags and z^-1 is the z-transform operator which is also known as

the backward-shift operator. However and worthy of note, the PREWHITEN() function does not return the one (1) in the AR(z^-1) polynomial and the PREFILTER() function implies the one (1) in its prefiltering calculations by internally including the one (1) in the prefiltering calculations. Prewhitening is essentially a multiple linear regression (MLR) algorithm which is a robust, reliable and effective method to de-serialize input and output signals, sequences or serieses which may or may not exhibit correlative and causative natures. The AR stochastic coefficients for each lag (excluding the one (1) in the monoic polynomial as mentioned) are stored, persisted or retained in the last data-set / -list / -vector argument which is required by the PREFILTER() operation or function to prefilter the output-series. Ultimately, prewhitening the input and prefiltering the output sharpens or enhances the cross-correlation in order to better expose the dead-time, time-delay, transport-delay or process-lag and the input-output deterministic process sub-model order. However, this applies only to open-loop data absent of feedback, intervention or counteraction control i.e., an input is some function of the output as computed by the controller. If the data is closed-loop with feedback, then it is known that the cross-correlation will return the inverse of the feedback controller and there will be a significant cross-correlation coefficient at time lag zero (0).

Univariate pseudo-random binary sequences (**PRBS()**), generalized binary noise signals (**GBNS()**) and white noise (**WN()**) functions may be specified to generate randomized, serially independent input-series or “dither” signals, suitable for time-series analysis, identification and estimation of actual or physical dynamical systems. These randomized series or sequences may also be used to perturb both the setpoint, target or reference signals of any univariate (SISO) or multivariate (MIMO) controlled variables (CV’s) as well as their manipulated variables (MV’s) in order to break the feedback control correlation structure when estimating dynamic process (and noise) transfer function models from closed-loop data. The first R.H.S. argument configures the length of the randomized signal, the second sets the size or amplitude of the signal, the third is the either the switch-time (PRBS) or switch-probability (GBNS) and the last is the random seed configured as any positive (+ve) integer number greater than zero (0) and less than or equal to 2^31 = 2,147,483,648 where each different random seed generates a different random sequence, series or signal. There is also an optional argument for each PRBS(), GBNS() and WN() data function which if present after the non-optional arguments configures the mean, average, opening or steady-state real value for the generated data-sequence, -series or -signal where if not present the default of zero (0.0) is applied.

Discrete-time and deterministic (process) transfer function plus dead-time (**TFPDT()**) dynamics and the stochastic (noise) auto-regressive integrated moving-average (**ARIMA()**) dynamics may also be configured based on the time-based z-tranform (z^-1) nomenclature and structure of Box and Jenkins, Time-Series Analysis: Forecasting and Control, *Holden-Day*, 1976 to dynamically simulate single-input and single-output (SISO) deterministic and stochastic linear time-invariant dynamic processes. For TFPDT(), the first data-vector is the exogenous or external input or independent time-series in deviation or perturbation variable form that can be generated by either PRBS() or GBNS() (cf. DATAU) and for ARIMA(), the first data-vector is the exogenous or external white noise or innovation shock time-series that can be generated via WN() (cf. DATAA) or supplied as data regression residuals which are inherently in deviation / perturbation variable form. The other arguments for TFPDT() are the dead-time or whole-periods of time-delay which includes the zero-order-hold (ZOH) sampling delay, degrees of the numerator and denominator and the data-vectors for their respective coefficients known as the Box and Jenkins (BXJK) numerator omega (w) and denominator delta (d) discrete polynomials i.e., y,t = (w0 + w1*z^-1 + w2*z^-2 + …) / (1 + d1 * z^-1 + d2 * z^-2 + …) * u,t-td where td is the time-delay or dead-time with a minimum value of one (1) for the inherent one time-period of sampling-delay due to the assumed zero-order hold (ZOH) sampling. *The initial, opening, starting or beginning time-period, -step, -instant or -interval index for the deterministic output time-series is MAX(dead-time + degree of omega; degree of delta) plus one (1).* The other arguments for ARIMA() are the degree of differencing, degrees of the numerator and denominator and data-vectors for their respective coefficients known as the Box and Jenkins (BXJK) numerator theta (t) and denominator phi (p) i.e., z,t = (1 + t1*z^-1 + t2*z^-2 + …) / (1 + p1 * z^-1 + p2 * z^-2 + …) / (1 – z^-1)^dd * a,t where z,t and a,t are the coloured- and white noise sequences respectively and dd is the degree of differencing. *The initial, opening, starting or beginning time-period, -step, -instant or -interval index for the stochastic output or colored-noise time-series is MAX(degree of theta; degree of differencing + degree of phi) plus one (1).*

The steady-state or stationarity (drift, trend, etc.) detection **SSD()** function is supported also with missing- / absent- / not-available- / non-existent-data handling and uses the algorithm published in Kelly and Hedengren, "A steady-state detection algorithm to detect non-stationary drifts in processes", *Journal of Process Control*, 2013. The state-state window size or length of the SSD() data function must of course be smaller than the size or length of the input data-set, -list, -vector, -1D-array, -signal, -series or -sequence and larger than or equal to three (3) for the number of non-missing-data-points. The critical / threshold value or upper control limit (UCL) is configurable and typically between three (3.0)

and four (4.0) relating to the Student-t statistic; please refer to the STUDENTT() datacalc function. The last argument is optional for the user, modeler or analyst and can supply or provide exogenously a standard-deviation value which is useful when the standard-deviation is known or assumed and should not be estimated using the data-window and is a tunable hyperparameter. The returned data-vector represents the probability of the input data-vector being steady or stationary (the null hypothesis of zero or no drift, trend or slope) where the steady-state or stationarity probability is simply computed as the number of times, instances, occurrences, events or excursions, divided by the window size, that a data-element, data-point or data-row contained in the data-window or time- / trial-plank lies inside or within the configured UCL, critical or threshold value. For the starting or initial number of probabilities equal to the size of the window length and relating to the number of priming time- / trial-periods, these are initialized to zero (0.0) probability implying and inferring unsteady-ness although they are really unknown, undefined or unavailable. It is the responsibility of the user, modeler or analyst to decide on and supply the probability tolerance of whether the data-signal is steady / unsteady or stationary / non-stationary and requires some amount of tuning with respect to the statistical threshold and eventually the probability threshold values especially when multiple data-sequences (i.e., other key sensors, instruments or meters) are required to reasonably assess or ascertain steady-state-ness for a system or envelope such as a process unit. The SSD() also supports the data-window length or size to be negative (-ve) on input, this will only call the IMPL Solvers' SSD algorithm known as IMPLsteadyability() once when the past number of data-vector-elements, -rows or -points have been reached or encountered and is a batch type of analysis. When the window argument is positive (+ve) on input, SSD() calls the IMPLsteadyability() routine for each or every time- / trial-period in the right-hand-side data-signal and is a continuous type of analysis. Fundamentally, when the returned SSD() probability is above a certain user, modeler or analyst defined UCL, threshold or critical value, then this indicates or signals the opportunity that the past data of length data-window is deemed to be steady and can trigger say an estimization and/or optimization application to be run or executed incorporating the past window size of data to compute aggregates such as averages or means and/or accumulations or totals. For systems or envelopes with more than one data-signal, the user, modeler or analyst must use the Bonferroni / Sidak adjusted significance-level for their UCL's with degrees-of-freedom equal to the number of data-vectors. Hence, the SSD() is considered as a multiple univariate or multi-univariate technique as opposed to a truly multivariate analysis. Please refer to the STARTRANGE(), SPANRANGE() and STOPRANGE() data functions to determine the start and stop index numbers for the returned SSD probability data-vector as well as the number, count or cardinality of index numbers within the time-

/trial-plank or -interval spanned contiguously or consecutively within the same lower and upper bound range. In addition, it is worthwhile mentioning that missing data elements, points or rows within the right-hand-side (R.H.S.) data-vector are considered as being unsteady or non-stationary by default i.e., their probability of being steady or stationary equals zero (0.0).

The **SYNCHRONIZE()** data function computes the well-known "dynamic time warping" (DTW) method to align, map, match, shift or synchronize a sample / test time-series signal to a known reference / template time-series signal where the first data-vector is the template signal and the second data-vector is the test signal and these two (2) signals may of course be of different sizes or lengths but with the same discrete time-period durations. The user, modeler or analyst is required to specify the norm type i.e., either one (1, absolute, Manhattan) or two (2, squared, Euclidean) only, and the retrieved or returned data-vector is the synchronized data-vector with the same size or length as the template or reference signal. The SYNCHRONIZE() algorithm is useful to temporally synchronize a test / sample time-ordered data-vector according to a 1-norm or 2-norm deviation metric from a known template / reference signal such as a startup, switchover or shutdown event sequence for any type of quantity, logic or quality phenomenological variable such as a flow, holdup (level, inventory, stock), temperature, pressure, etc.

First-order partial derivatives or gradients, gains, deltas, intensities or slopes may be easily computed using the **DERIVATIVES()** data function where the data-vector argument contains the initial-values, starting-points, default-results or more appropriately variable-points for each variable considered and the numerical perturbation size is the IMPL setting PERTURBPLUS (default = 1D-12). The DERIVATIVES() datadata function returns a data-vector containing the first-order partial derivatives with the same size or length as the variable-points' data-vector; please see the complementary EVALUATE() datacalc function. Immediately after the DERIVATIVES keyword and its closing paranthesis and closing comma delimiter i.e., ")," is the formula or equation expression in infix notation with upto **4096** characters. Each variable must be represented by a prefixed upper case "X" only with an immediately suffixed or appended positive (+ve) integer number index i.e., X99 or X999 in the domain or range of 1.. 2,147,483,648 (2^31). The integer numbers must match or map to the exact data-elements in the data-vector argument. If the formula / equation expression is invalid and thus cannot be lexed and parsed or compiled into postfix notation, then the resulting data-vector will contain all RNNON's. In addition and

similar to the foreign-model *.ilp and *.inp foreign-files, the DERIVATIVES() data function allows known calc-scalars and data-vector-elements to be recognized within the formula-expression.

Single-input-single-output auto-regressive exogenous (**SISOARX()**) function, also known as the infinite impulse response (IIR), performs multiple linear regression with one input and one output only with selected lags to directly estimate discrete-time dynamic process (combined deterministic and stochastic) transfer functions of the form *y,t = a1 * y,t-1 + … + am * y,t-m + b0 * u,t-k-0 + b1 * u,t-k-1 + … + bn * u,t-k-n + e,t* where *m* is the degree of the output denominator part of the transfer function, *n* is the degree of the input numerator part of the transfer function and *k* is the process time-delay or dead-time (cf. Ljung, L., System Identification: Theory for the User, *Prentice-Hall*, 1987 and King, M., Process Control: A Practical Approach, *John Wiley & Sons*, 2016). Of note in terms of the formulation, we have purposefully formulated the SISO ARX in multiple linear regression (MLR) model form (i.e., y = X * b) so as to allow seamless interchange of their coefficient or parameter results between the different IMPL-DATA's regression methods i.e., SISOARX(), MLR(), RR() and SSR(). The time-serieses *y,t* and *u,t* are the mean-centered process output and process input (external or extra stimulus signal input) respectively where SISOARX() always assumes that the supplied or provided data (i.e., *y,t* and *u,t*) to SISOARX() are in deviation or perturbation form (i.e., subtracting its mean, steady-state, opening, initial, starting or beginning value) before estimating the ARX coefficients or parameters (*a1, …, am*, *b0, …, bn*) and the degrees and delay tuple (*m,n,k*) must be supplied by the user, modeler or analyst exogenously. Importantly and by definition, we tacitly assume that u,t-k causes (i.e., affects, influences, maps to, relates to, etc.) y,t and therefore causality / causation between the independent process input (u,t-k) and dependent process output (y,t) is implied or inherent. There are two (2) returned data-sets / -lists / -vectors from the SISOARX() data function, method or operation with the first left-hand-side data-vector containing the multiple linear regression residuals or prediction errors (i.e., actual / measured minus predicted / modeled) otherwise known as the pseudo-white noise series or innovation shocks (i.e., see *e,t* in the above equation) and is expected to be identically and independently distributed (i.i.d.) i.e., random with no significant serial or auto-correlation. The second returned data-vector (right-hand-side) contains the tuples of the estimated ARX coefficients followed by their variances, the appended model structure (degrees and delay) and diagnostic statistics specifically (*a1, …, am, b0, …, bn, va1, …, vam, vb0, …, vbn, m,n,k, TSS,ESS,R2,Q,X2,RNNON*) where *va1,* etc. are the estimated variances for each parameter, TSS is the total sum-of-squares, ESS is the error or residual sum-of-squares where the regression sum-of-squares RSS = TSS – ESS, R2 is the unadjusted coefficient of determination, Q is the

Ljung-Box portmanteau statistic using a default of thirty lags (cf. IMPL setting NUMLAGS = 30) for its residual auto-correlations relevant when the data are dynamic time-serieses, X2 is its (Q's))Chi-Square statistic threshold or critical value at a default significance-level of 0.05 or 5% and finally the appended RNNON is to indicate the end of the data-vector. To compute the mean-square (prediction) error (MSE / MSPE) simply divide the ESS or SSE by the number of trials minus one (1) where it is interesting to note that the MSE / MSPE is theoretically composed of three (3) terms: the variance, the bias squared and the irreducible error squared. The root mean square (prediction) error (RMSE / RMSPE) is simply the square-root of the MSE also known as the regression's standard error. Take note that the input and output means are not returned but can be easily calculated using the MEAN() function. The estimated or expected random shocks, a.k.a. white noise load disturbance time-series, or the sequence of regression residuals / prediction errors that are minimized in the ARX multiple linear regression manifests everything unknown or not accounted for in the estimated ARX dynamic model including both unmeasured and measured disturbances (i.e., potential or possible feedforward disturbance, disruption or distraction variables). As the SISOARX() routine is a direct parametric model estimation technique, either open-loop or closed-loop data (including manual feedback / operator intervention / counteraction) may be used where closed-loop data implies that the process input is a function of the process output via a controller transfer function such as a PID control algorithm (proportional-integral-derivative) or the like. SISOARX() also properly handles and manages missing-, absent- or non-existent data in both the process output and input whereby instead of the ARX estimation being linear regression, it automatically becomes nonlinear regression as the missing-data data-elements / -points / -rows for an output and/or input become themselves estimation variables and hence we have coefficient or parameter variable times an output or input variable which is bilinear or second-order and thus nonlinear.

As a matter of insight, although the ARX model structure of SISOARX() is parametric and effective at describing the behavior of closed-loop dynamic systems under feedback control (with or without setpoint and process input dither signals), it should be highlighted that the more parsimonious, separated and general Box and Jenkins (BXJK) deterministic (process) and stochastic (noise) transfer function model structures, found in their book Time-Series Analysis: Forecasting and Control, *Holden-Day*, 1976, have been shown by several researchers to be more accurate than ARX, ARMAX and Output-Error (OE) including sub-space or state-space (SS) models. However, parametric ARX models are simpler to identify and estimate and do not require more complicated identification methodology and nonlinear

regression methods such as the Levenberg-Marquardt regularization heuristic (unless missing-data are present) and ARX models recommended for PID controller design by Rivera and Gaikwad, "Digital PID controller design using ARX estimation", *Computers and Chemical Engineering*, 1996 and Guzman, Rivera, Berenguel and Dormido, "*i-pIDtune*: An interative tool for integrated system identification and PID control", *IFAC PID'12*, 2012. Further to that, it should be mentioned that the underlying IMPL Solvers' routine that computes the SISOARX() results called IMPLslopeability() is actually implemented and invokes IMPLseriesability() which does implement the more general nonlinear least squares algorithm (Levenberg-Marquardt method) discussed in Box and Jenkins (1976) but of course is more problematic in terms of convergence stability due to non-convexity and initial- or starting-values. And, it is also interesting to mention that it is straightforward to duplicate the PREWHITEN() data function with the SISOARX() by essentially auto-regressing any time-series, time-indexes or time-ordered data-vector with itself instead of having a different process output and input. The user, modeler or analyst simply configures that the SISOARX() output is the same as the input time-series with the appropriately chosen lag structure for the output and input. Please note however that the ARX coefficients from SISOARX() must be multiplied by minus or negative one (-1.0) in order to properly match the AR coefficient data-vector values from PREWHITEN() which has no exogenous input excitation. *And 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.*

Single-input and single-output (SISO) proportional-integral-derivative (PID) control is a very important industrial sub-algorithm found in all modern SCADA, PLC and DCS Operational Technology (OT) computer systems with its formalization dating back to 1922 and used for setpoint tracking (servo-control) and disturbance rejection or suppression (regulatory control). Tuning for installed performance characteristics can be difficult and time-consuming and IMPL's **RETUNEPID()** data function is unique given that closed-loop feedback data of installed or existing PID controllers may be used to identify and

estimate the process dynamic parametric model via SISOARX(). RETUNEPID() is essentially an "indirect" model-based method where no open-loop or manual step testing is required although the PRBS(), GBNS() and WN() may be used to provide process input dithering and/or a setpoint dithering, deviaition or perturbation signals which can be employed to better identify the process delay or dead-time, etc. when applied to the real or physical process control loop. The first and second input data-vectors to RETUNEPID() are the setpoint and white noise load disturbances in deviation / perturbation variable form (i.e., steady-state- or mean-centered). Refer to SISOARX()'s e,t variable representing the regression residuals or prediction errors which are hopefully independent and identically distributed (i.i.d.) random shocks but is not a requirement. The white noise load disturbance may be obtained directly from SISOARX()'s left-hand-side data-vector of multiple linear regression residuals and the third data-vector is the last right-hand-side SISOARX() argument of ARX coefficients, coefficient variances, degrees plus delay tuple and diagnostics i.e., (*a1, …, am*, *b0, …, bn, va1, …, vam*, v*b0, …, vbn, m,n,k, TSS,ESS,R2,Q,X2,RNNON*). The fourth and fifth calc-scalar arguments are the degree of differencing (0, 1 or 2) and the (normalized or standarized) input-move-error or rate-of-change norm upper bound, cut or limit respectively. The input-move-error norm upper bound is considered as RETUNEPID()'s single retuning hyperparameter which influences the amount of input-move (rate-of-change) variability. *If the upper bound argument is negative (-ve), then RETUNEPID() minimizes the input-move-error norms instead of the output-error norms and is subject to or constrained by the upper bound on the output-error norms instead of the input-move-error norms. This is required for what is known as "level-flow smoothing" (LFS) or otherwise known as "averaging-level control" (ALC); for more details see Kelly, "Tuning digital PI controllers for minimal variance in manipulated input moves applied to imbalanced systems", Canadian Journal of Chemical Engineering, 1998 and Horton, Foley and Kwok, "Performance assessment of level controllers", International Journal of Adaptive Control and Signal Processing, 2003.* The last right-hand-side input argument must be a seventeen (17+) element sized data-vector containing in order the installed proportional-gain (Kp, 1), its lower bound (2) and its step-size, -increment or -bound (3), its upper bound (4) and its step-size, -increment or -bound (5), the same for both the integral-time (Ti, 6..10) and the derivative-time (Td, 11..15) terms, time-period / time-step / time-instant duration, execution cycle, cycle-time or sampling interval or instant (Ts, Tc or tpd, 16) with the very last data-element, -point or -row (17) indicating the well-known PID equation type configured as zero (0, default), one (1) or two (2) for the A, B and C PID equation types respectively where for proportional control only, set Ti to a large real number and Td to zero (0.0). **Please note that if the Kp, Ti or Td upper bounds are less than or equal to their default setting values, then only one range is calculated instead**

of two spanning from the lower bound to its upper bound only. In addition, data-vector-elements (18), (19), (20) and (21) may contain the process input and output lower and upper bounds also in deviation / perturbation variable form (i.e., minus its mean, steady-state or opening, initial, starting or beginning value) and these are incrementally optional defaults as -INFIN and INFIN respectively. The controller output or process input (u) bounds are implemented as clamping limits whereas the process output (y) bounds determine whether the output for each time-period, -step or interval is feasible. Take note that there is also integral-gain (Ki) and derivative-gain (Kd) which must be converted to integral-time and derivative-time i.e., Ki = Kp / Ti and Kd = Kp * Td where conceptually the Kp relates to the present, Ti to the past and Td to the future. The returned L.H.S. data-vector contains seventeen (17) elements starting with the "best" 1-norm (sum-of-absolute errors) Kp, Ti and Td (1..3) settings in terms of minimizing the process output-errors (setpoint minus process output) subject to the supplied or specified process input-move-error's upper bound limit where the fourth (4) and fifth (5) data-vector-elements are the "best" normalized / standardized 1-norm output-error and input-move-error norms respectively. Data-vector-elements (6..10) contain the normalized 2-norm (sum-of-squared errors) and (11..15) contain the infinity-norm (maximum absolute error) PID settings and error performance metrics respectively; the 1-norm and 2-norm are normalized or standardized by simply dividing by the number of trials, samples or observations in the setpoint / white noise load disturbances which make them more comparable or consistent to the maximum- / oo-norm and consistent across varying numbers of time- / trial-periods. The penultimate and ultimate data-vector-elements (16..17) contain the total number of possible simulation runs and the actual number of simulation runs that are feasible (i.e., within their clamping and limiting bounds) and are not unstable; the difference between these two is the number of infeasible / unstable simulation runs. It is important to highlight that since the (normalized) input-move-error norm upper limit is essentially a constraint or cut applied to the brute-force grid (or exhaustive) search when finding the lowest or minimal (normalized) output-error norm, our PID retuning optimization by exhaustive search may be unattainable, infeasible or inconsistent where no valid PID settings may be found if the input-move-error upper limit cannot be achieved as indicated by the existence of RNNON's, therefore the user, modeler or analyst may need to increase the (normalized) input-move-error norm upper bound, cut or limit accordingly.

A salient comment on the non-setpoint disturbance in RETUNEPID(), to accommodate retuning PID process control loops which may not have white noise load disturbances, as there may be measured feedforward disturbances and/or interacting measured disturbances from other PID loops in a multi-

loop / coupled system, the user, modeler or analyst can “prefilter” their collected or combined load disturbance using the PREFILTER() data function employing the auto-regression (AR) coefficients estimated, fit or regressed from SISOARX(). The prefiltering calculations will simply cancel out RETUNEPID()’s internal conversion or transformation of the white noise load disturbance to a colored-noise load disturbance inherently computed by the ARX dynamic model. By preprocessing with PREFILTER(), any arbitrary load disturbance data-vector may be configured into the white noise load disturbance argument provided it is prefiltered accordingly as described.

It is worth mentioning that any of the data functions, methods or operations found immediately listed below must not be included in any type of algebraic or formula-expression which is identical to a subroutine call statement versus a function call found in other computer programming languages. Only its scalar arguments between the data functions’s semi-colons and right parentheses may be simple calc-scalars / data-vector-element expressions i.e., the vector data function arguments must not be involved or included in any calculation / data expression and unfortunately no left and right parentheses are allowed in the simple calc-scalar / data-vector-element expression as it conflicts with the data function’s parentheses “(“ and “)”. Instead, if complicated calc-scalar / data-vector-element expressions or formulas are required for the right-hand-side (R.H.S.) arguments, then use “{“ and “}”. Once the left-hand-side (L.H.S.) data-vector is created or modified cleanly, using these data functions or operations, it may only then be involved or included in any data expression, formula or algebra similar to the calc-scalar / data-vector-element expressions.

```
&sDataData,@sValue
```
DATA,**DATETIMENOW**() ***Note: [1] = yyyy, [2] = mm, [3] = dd, [4] = hh, [5] = mm, [6] = ss and [7] = mss**
DATA,**DATENOW**()
DATA,**TIMENOW**()

DATA,**SCATTER**(VALUE;LENGTH) ***Note that if DATA exists and “LENGTH” is negative (-ve) then skip**
DATA,**SYNTHESIZE**(VALUE;LENGTH;LOWER;UPPER**;STRIDE**) ***Note that “;STRIDE” is optional**
DATA,**SEQUENT**(VALUE;LENGTH) ***Note that if “LENGTH” is negative (-ve) then sequence is descending**

DATA,**SLIDE**(DATA2;NUMBER)
DATA,**SLICE**(DATA2;LOWER;UPPER) ***Note that the “LOWER” and “UPPER” are indexes or indices**
DATA,**SPLICE**(DATA2;DATA3) ***Note see also the data frame to perform two or more multiple splices**

DATA,**SHED**(DATA2;LOWER;UPPER;VALUE) ***Note that VALUE is “imputed” and can be RNNON**
DATA,**SKIP**(DATA2;NUMBER) ***Note that if “NUMBER” is zero (0), then DATA = DATA2**
DATA,**SHIFT**(DATA2;LAG;VALUE) ***Note the if the “LAG” is negative(-ve), then a lead**

DATA,**STACK**(DATA2;LENGTH)
DATA,**STACK**(DATAGROUP;**LENGTH**) ***Note that "LENGTH" is ignored with data-groups**
DATA,**STRETCH**(DATA2;LENGTH)

DATA,**SWAP**(VALUE;INDEX)
DATA,**SWAP2**(DATA2;INDEX) ***Note that "INDEX" is the starting indice to suffix or append "DATA2"**
DATA,**SUBSTITUTE**(DATA2;DATA3) ***Note that "DATA2" are the values and "DATA3" are the indexes**

DATA,**SHUFFLE**(DATA2;SEED)

DATA,**SCRAPERANGE**(DATA2;LOWER;UPPER;**FLAG) *Note that ";FLAG" is optional (0, default or 1).**
DATA,**STARTRANGE**(DATA2;LOWER;UPPER)
DATA,**SPANRANGE**(DATA2;LOWER;UPPER)
DATA,**STOPRANGE**(DATA2;LOWER;UPPER)

DATA,**SORT**(DATA2)
DATA,**SEQUENCE**(DATA2;DATA3) ***Note that "DATA2" are the values and "DATA3" are the indexes**
DATA,**DESORT**(DATA2)

DATA,**SPLINES**(DATAXX;DATAYY;SPLINETYPE;DATAX)

DATA,**ACCUMULATE**(DATA2)
DATA,**DIFFERENCE**(DATA2;DEGREED) ***Note that if "DEGREED" is zero (0), then DATA = DATA2**

DATA,**XCORRELATION**(DATA2;DATA3;**LAGS**) ***Note that if LAGS is negative (-ve), no mean-centering**
DATA,**PCORRELATION**(DATA2;**LAGS**) ***Note that if LAGS is negative (-ve), no mean-centering**

DATA,**PREWHITEN**(DATA2;LAGS;DATA3)
DATA,**PREFILTER**(DATA2;DATA3)

DATA,**PRBS**(LENGTH;SIZE;SWITCH;SEED**;MEAN**) ***Note that ";MEAN" is optional**
DATA,**GBNS**(LENGTH;SIZE;SWITCH;SEED**;MEAN**) ***Note that ";MEAN" is optional**
DATA,**WN**(LENGTH;SIZE or STDEV;SEED**;MEAN**) ***Note that ";MEAN" is optional**

DATAZ,**ARIMA**(DATAA;DEGREED;DEGREET;DEGREEP;DATAT;DATAP)
DATAY,**TFPDT**(DATAU;DELAY;DEGREEW;DEGREED;DATAW;DATAD)

DATA,**SSD**(DATA2;WINDOW;CRITICAL**;STDEV**) ***Note that ";STDEV" is optional**

DATA,**SYNCHRONIZE**(DATA2;DATA3;NORMTYPE) ***Note that DATA2 is the reference/template signal**

DATA,**DERIVATIVES**(DATA2**),FSTRING *Note that the FSTRING is outside the parentheses**

DATAWNLD,**SISOARX**(DATAY;DATAU;DEGREEM;DEGREEN;DELAYK;DATAARX)
DATA,**RETUNEPID**(DATASP;*DATAWNLD*;DATAARX;DEGREED;UPPERNORMLIMIT;DATAPID)

DATA,**xX**(XSUBNAME;START;STOP**;SUPREMUM) *Note that ";SUPREMUM" is optional**

DATA,**fF**(FSUBNAME;START;STOP**;SUPREMUM)**
`&sDataData,@sValue`

Due to the relatively simplistic lexing, parsing and compiling of the datadata frame right-hand-sides (R.H.S.'s), only one set of left and right matching parentheses i.e., "(" and ")" are allowed in any feature, line or row of the datadata frame. This also applies to the data function's semi-colon (";") argument delimiter whereby calculation / calc-scalar expressions involving ";" must not be found embedded as data function or operation arguments. If more complicated value, length, degree, number, etc. arguments or parameters are required, then simply replace these with explicit calc-scalars avoiding the use of calc-scalar expressions / formulas with parentheses and semi-colons for the value, length, degree, number, etc. arguments. However, curly braces "{}" may be used as equivalent substitutes for parentheses "()" when calc-scalar formulas are required to group certain terms in the expression or formula.

To model with data-blocks, data-grids, data-tables, data-columns or data-matrices (i.e., 2D-arrays), IMPL provides the capability to simply group together one or multiple data-sets, data-lists, data-series, data-columns or data-vectors where each group must be entered or encountered consecutively or contiguously as shown below i.e., one data-group after the other in series, sequence or order. Data-groups are a simple and straightforward way for IMPL to form two-dimensional (2D) arrays with multiple columns of data i.e., multivariable or multivariate data structures, composites, collections or containers. In IMPL, a data-vector contains multiple calc-scalars and a data-group contains multiple data-vectors also known of course as a matrix, block, grid, table or 2D-array with rows and columns. We use the word group instead of matrix as IMPL uses the term group for multiple constructs such as unit-groups, unit-operation-groups, unit-operation-port-state-groups, etc. and thus data-group (calc-group and text-group) is a consistent and standard notion within IMPL**. Please be advised that the data-group frame supports the SPIN"", SPIN2"" and SPIN3"" data formulary's also provided with the datacalc and datadata frames to allow both the data-sets, -lists or -vectors and data-groups to have appended or suffixed leading-zeros integer number strings via the special characters "$", "&" and "@" also available with the ILP / INP foreign-model's REPEAT() model functions.**

`&sData,&sGroup`
DATA1,DATAGROUP1
… ,DATAGROUP1
DATAN,DATAGROUP1

DATA1,DATAGROUP2
… ,DATAGROUP2
DATAN,DATAGROUP2

DATA,DATAGROUP,@,@ or
DATA,DATAGROUP,@,@,INNON

DATA,DATAGROUP,@,@,@ ***Note the start,stop,step/stride tuples for 1, 2, and 3 dimensions**
DATA,DATAGROUP,@,@,@, @,@,@
DATA,DATAGROUP,@,@,@, @,@,@, @,@,@

DATA,DATAGROUP,@,@,@, @,@,@, @,@,@ ***Note that the @'s may be duplicated w/ the SPIN""'s**
`&sData,&sGroup`

The two (2), three (3), etc. asperands "@" shown in the data-group frame indicate the hidden fields that for instance if two (2) are present specify the lower (begin) and upper (end) integer index number bounds that can systematically configure a contiguous / consecutive block, series or sequence of data-vectors into the same data-group using the naming convention found in the SYNTHESIZE() function. If three (3) hidden fields are detected, then they represent the start (lower, head >= 0), stop (upper, tail <= 2^31 = 2,147,483,648) and step (stride, spacing, increment, etc.) integer index numbers which also includes our leading-zeros index numbering convention identical to the SPIN"" data formulary and the REPEAT() and REPLICATE"" model function / formulary. If INNON is detected in the third (3rd) hidden field, then no leading-zeros index numbering is applied and is identical to the having just two (2) hidden fields. Effectively, the other hidden integer fields enable automatic configuration of the following: DATALOWER, DATAGROUP, …, DATAUPPER, DATAGROUP and is helpful to automatically, systematically or programmatically configure the assignment, association or attachment of multitudes of data-vectors to a single data-group or block, grid, table, 2D-array or matrix. And likewise, if six (6) or nine (9) hidden fields are configured, then their start, stop, step tuples automatically generate two (2) or three (3) dimensions of many data-sets, -lists or -vectors to one (1) data-group. **Please note that there is no implied ordering or sequencing in terms of the arrangement, location or placement of the data-vectors with their respective data-group as both the data-vector and its data-group are combined together to form the unique key duple (or duplex tuple). As well and similar to the SYNTHESIZE() datadata function and the SPIN"" data formulary, if the stop (upper, tail) is less than (<) the start (lower, head) sub-iterator, then the data-vector numbering is skipped, ignored or passed-over as the iterator tuple is considered to be a null set or list.**

The first data function to use the data-group is the **MTIMES()** and supports missing-, absent-, unavailable- or non-existent-data handling by ignoring the update of the matrix product incremental sum when either or both data-vector-elements are found to be RNNON. The target, returned, retrieved or resulting data-group, first data-group and second data-group arguments all need to be defined, declared, configured or specified before MTIMES() is invoked or called. This is a simple dense matrix multiplication of the form Z = X * Y where X has dimensions of NRX x NCX, Y is NRY x NCY and Z is NRZ x NCZ for rows and columns respectively. The retrieved data-group (Z) has the row dimension of the first data-group argument (NRX) and the column dimension of the second data-group argument (NCY) i.e., Z has dimensions of NRX x NCY where NRZ = NRX and NCZ = NCY.

Similar to the MTIMES(), IMPL also supports the **MTRANSPOSE()** of any data-group where the rows become columns and the columns become rows i.e., XT = X' = MTRANSPOSE(X) where XT has dimension NCX x NRX and no missing-data handling is of course required or necessary. As with the MTIMES(), the MTRANSPOSE() requires that the left-hand-side data-group and its data-vector members must all be defined, declared or configured before or prior to calling or invoking the transpose data manipulation operation.

Please be aware that the left-hand-side (L.H.S.), dependent, retrieved or returned data-groups (and hence their data-vector members or assignees) must always be configured explicitly before calling these data functions as IMPL expects that these grouped data-vectors be properly lengthed, sized or dimensioned via the SCATTER() or SYNTHESIZE() data functions for example.

```
&sDataData,@sValue
DATAGROUP,MTIMES(DATAGROUP1;DATAGROUP2)
DATAGROUP,MTRANSPOSE(DATAGROUP1)
&sDataData,@sValue
```

Data-block, data-table, data-grid and data-matrix multivariable functions such as the Latin Hypercube Sampling (**LHS()**) require the above data-groups to be configured in order to properly insert, set or populate the resultant multivariate data structures into the data-group's individual data-set, data-list or data-vector members. LHS is a well-known segregated random sampling technique for efficient and effective design of experiments (DoE), runs, simulations, trials, etc. where each data-vector in the data-group will vary from zero (0.0) to one (1.0) randomly and clustering of the variable moves or perturbations will be minimized accordingly to the design of the LHS technique. More specifically, for

each data-vector, the number of samples (n) chosen, selected or sampled are randomly distributed with one from each interval (0.0,1.0/n), (1.0/n,2.0/n), …, (1.0 – 1.0/n,1.0) and randomly permuted. The LENGTH argument configures the number of samples, runs, experiments, surveys, simulations, trials, etc. and the SEED argument is the random number generator seed and must be greater that zero (0). The final or end result of the LHS is known as a sample matrix, grid, block, sheet or table. Take note that if the data-group contains only one (1) data-vector which is valid, then the LHS will simply return a random series or sequence of uniformly distributed deviates, shocks or innovations within the open interval (0.0,1.0).

```
&sDataData,@sValue
DATAGROUP,LHS(LENGTH;SEED)
&sDataData,@sValue
```

Similar to the LHS() but a discrete version, the "K from / out of M shuffling or selecting" (**KMS()**) data function returns or retrieves a data-group of data-vectors with randomly selected or shuffled K values out of M for a specified number of length or size denoted as N i.e., selections, shuffles, samples, speculations, scenarios, situations, etc. The returned values are zero (0) or one (1) where the sum over M for each N exactly equals K and is adeptly suitable for example to exogenously generate, provide or supply N rows of M coefficient-setups or parameter-switches in data regression or sensor-switches or -setups in data reconciliation. A previous and simple algorithm taken from Green, B.F., "Fortran subroutines for random sampling without replacement", *Behavior Research Methods & Instrumentation*, (1977) is replaced by the *Partial Fisher-Yates Shuffle* found in the works of D.E. Knuth which is also called the *Simple Random Sampling WithOut Replacement* (SRSWOR) algorithm. The other input arguments are the random number integer seed and the K count, arity or cardinality of the subset of {0,1} numbers to choose, select, shuffle or permutate from the M number of data-sets / -lists / -vectors contained in the L.H.S. data-group. It is worth mentioning that some amount of duplication will exist i.e., in the N selections or shuffles configured, not all will be unique or different due to the randomness of the randomization as there is no mechanism to detect and eliminate repeats or revisits of the same solution in the algorithm. Therefore, the user, modeler or analyst should be wise in terms of configuring the number of selections or shuffles given K and M i.e., the number of distinct combinations N' is computed as N' = M! / ((M-K)! * K!).

Please note that for convenience, there is also an optional exclusion / inclusion set (cf. DATAZ below) that if any of its elements are zero (0), then the corresponding index is not selected or shuffled i.e., it is excluded from the search set. And, if the optional DATAZ last argument is present or exists, then if the corresponding indice is to be included it must have a value of one (1). The default when DATAZ is missing or absent is to include all indexes or indices in the selecting or shuffling i.e., DATAZ = [1,…,1].

In addition, the optional infix formula or expression string (and similar to the EVAULATE(), DERIVATIVES() and RECURSE() functions) representing a discrete validation constraint, cut or rule may be included as shown below where its computed or evaluated value must equal integer zero (0), else the randomly generated discrete coefficient-setups or parameter-switches are re-randomized until the formula eventually resolves to zero (0). We have chosen the word "validation" as these discrete constraints, cuts or rules provide a similar notion of causality as cross-validation via sample splitting with the inclusion of both training / in-sample / estimation / calibatrion and testing / out-of-sample / validation / confirmation data sets. The formula string may contain any known calc-scalars, data-vector-elements and the X1..XN with the upper case or capital "X" also used for ILP / INP foreign-variables and the index numbers of indices 1..N must correspond one-for-one to the ordering of the coefficient-setsups / parameters-switches where N equals the R.H.S. length argument. A useful expression or formula term to include is the MIN(MAX(F(X1,…,XN);0);1) where X1..XN correspond to the coefficient-setups or parameter-switches and not the coefficient or parameter values themselves and F(X1..XN) is some arbitrary infix function or formula. Essentially, the infix formula is intended to represent multiple A * X – b <= 0 types of terms in its expression and may also be thought of as a discrete acceptance, fitness or utility function for the coefficient-setups / parameter-switches.

And, it is important to be aware that presently there is no "watchdog" timer, counter or iterator for the number of cycles, repeats or trys when the discrete validation constraint, cut ot rule is invalid or violated which can cause excessive computation especially if tightly constrained, and if infeasible, must be aborted, exited, stopped or terminated by the user, modeler or analyst.

```
&sDataData,@sValue
DATAGROUP,KMS(LENGTH;SEED;NUMBER)
DATAGROUP,KMS(LENGTH;SEED;DATAZ) *Note that ";DATAZ" is optional
DATAGROUP,KMS(LENGTH;SEED;DATAZ),FSTRING *Note that ",FSTRING" is optional
&sDataData,@sValue
```

Principal Component Analysis (PCA) with missing-, absent-, not-available or non-existent-data handling (cf. RNNON) is another multivariable data function callable in IML as **PCAP()** and **PCAT()** which require the above data-groups as the input is a data-table, -grid, -sheet, -matrix or -block of independent and dependent variables of arbitrary correlation, covariance, colinearity or inter-relationships. PCA is a very well-known dimensionality reduction technique similar to Eigen Value Analysis (EVA) and Singular Value Decomposition (SVD). The DATAGROUPX argument below configures the X-block of data i.e., number of runs, samples, experiments, surveys, simulations, trials, etc. by the number of X data-vectors and the SCALE argument varies between zero (0) and three (3) for no scaling (0), mean-center and standard-deviation scaling (1), mean-centering only (2) and standard-deviation scaling only (3). DATAGROUPP contains the PCA loadings (P) when PCAP() is called with the dimension of number of X data-vectors by the number of principal components and DATAGROUPT is the returned or retrieved PCA scores (T = X * P) when PCAT() is called and has the dimension of number trials, observations or samples by the number of latents, factors or principal components (in order of largest to smallest co-variation / covariance explained). The variances of the PCA scores (T) or principal components are easily computed using the VARIANCE() data function, sometimes referred to as the eigenvalues of the X-block, which are required when computing the well-known Hotelling (T2) statistic. The total sum over all of the score variances for all of the X variables i.e., the number of principal components equals the number of X variables, computes the total variation in the X block.

```
&sDataData,@sValue
DATAGROUPP,PCAP(DATAGROUPX;SCALE) *Note that P are the loadings
DATAGROUPT,PCAT(DATAGROUPX;SCALE) *Note that T are the scores
&sDataData,@sValue
```

It is important to highlight that the number of latents, dominant dimensions, primary factors or principal components is determined or specified by the number of data-sets/ / -lists / vectors contained in DATAGROUPP (loadings) or DATAGROUPT (scores) and for interest, the internally computed sample covariance or information matrix (X' * X) analyzed is divided by the number of trial-periods, samples, observations, experiments, etc. minus one (1).

A multiple linear regression (**MLR()**) function with missing- / -absent- / unavailable- / non-existent-data handling is provided which is the well-known application of least-squares regression with a single output y variable and multiple input X variables i.e., multiple-input and single-output (MISO). The first argument in the MLR() function is the dependent y data-vector and the second argument is the

independent data-group with one or more X data-vectors also referred to as regressors, predictors or explanatory variables where the number of X's is determined or specified by the number of data-vectors contained in the data-group identical to PCA(). The third data-vector contains the coefficient or parameter estimates immediately followed by their variance estimates and diagnostic statistics identical to SISOARX() (i.e., TSS,ESS,R2,Q,X2,RNNON) with the last data-element valued at RNNON to indicate the end of the coefficient, coefficient variance and diagnostics data similar to SISOARX(). The standard equation for this data regression is y = X * b where y and b are column vectors of size n x 1 and p x 1 respectively and X is a matrix (or group of column vectors) of size n x p with n being the number of observations, samples, experiments, trials, etc. and p being the number of X variables and equivalently the number of coefficients or parameters. The y and X data-vectors are not mean-centered nor scaled (i.e., non-standardized) nor is any constant, intercept, base, balance or bias coefficient / parameter included. The returned data-vector returns the multiple linear regression residuals, errors or deviations of the measured or actual y data from its model predictions in the exact order or sequence as found in the first data-vector. For interest sake, it is possible to perform MISO ARX regression coupled with the SHIFT() data function to time-lag or -shift the X and y data accordingly and call MLR() to estimate the coefficients or parameters in the MISO ARX dynamic model. In addition, polynomial first-, second-, third-order, etc. nonlinear ARX (NARX) dynamic models may also be regressed, fit or estimated using MLR() since these types of models are also linear-in-the-parameters. As well, it is also possible to perform Principal Component Regression (PCR) where the X group of data-vectors is replaced by the PCA scores (T) group.

Please note that for convenience, the optional and exogenous coefficient-setup/-switch data-vector (i.e., DATAZ below) is supported which if zero (0), then the coefficient corresponding to its respective input X regressor, predictor or explanatory variable is excluded from the data regression and its coefficient is fixed / forced and defaulted to zero (0.0). If the coefficient-setup is one (1), then the coefficient is free / finite and will be fit, estimated or regressed. This allows the user, modeler or analyst to configure many possible independent variables into the data-group argument and then to systematically include (one, 1) and exclude (zero, 0) any X variable as required by specifying the optional and exogenous coefficient-setup data-set / -list / -vector.

```
&sDataData,@sValue
```
DATA,**MLR**(DATAY;DATAGROUPX;DATAB)
DATA,**MLR**(DATAY;DATAGROUPX;DATAB**;DATAZ) *Note that ";DATAZ" is optional**

```
&sDataData,@sValue
```

It is also important to note that unless the dependent, output, predicted or regressand variable (y) is mean-centered in either the MLR() above or the RR() below, the TSS and R2 retrieved in the coefficient data-set, -list or -vector returned is not correct as the mean or average of y[1..nt] is not included in these summary statistics. To compute the proper TSS' and R2' when no output mean-centering exists, simply compute the mean of y i.e., ym = MEAN(y) and adjust TSS from the MLR() or RR() data functions i.e., TSS' = TSS – sum over t = 1..nt {y,t} * ym + nt * ym^2.0 where nt is the number of trials, samples, observations, experiments, etc. and SUM(y) = sum over t = 1..nt {y,t} * ym. The coefficient of determination is then properly calculated as R2' = 1.0 – ESS / TSS' = (TSS' – ESS) / TSS' where ESS is unchanged as it is not a function of the output y mean (ym).

The ridge or regularized regression (**RR()**) data function is available and identical to MLR() for linear regression except for the regularization or ridge hyperparameter commonly referred to as "lambda" to regularize coefficients or parameters when significant multi-collinearity exists amongst the independent X data-vectors i.e., RR() works best in situations where the MLR() coefficients or parameters have high variance. If lambda equals zero (0.0), then RR() is equivalent to MLR(). An insightful paper on the practical use of ridge regression is Marquardt and Snee, Ridge Regression in Practice, *The American Statistician*, 1975. The well-known information matrix X' * X is simply replaced by X' * X + lambda * I where I is an identity matrix equal in size to the number of X's. Ridge regression requires the user, modeler or analyst to exogenously configure the lambda value where the primary goal of the regularization is to minimize the mean square prediction error (MSPE or MSE) of the testing, out-of-sample, confirmation or validation data (versus the training, in-sample, calibration or estimation data), known as cross-validation via sample splitting, by its adjustment, trimming, tweaking, tinkering or tuning whilst simultaneously reducing the variance or standard error in the estimated coefficients. Multi-collinearity occurs especially when dynamic data are regressed with significant auto- and cross-correlation but also when static apparent / structural ill-conditioning exists in the X-block data either incidentally or intentionally when quadratic and/or interaction terms are included. Unlike multiple linear regression, ridge regression is not scale invariant due to the objective function tradeoff of minimizing the regression or residual sum-of-squares together with the sum-of-squares of the coefficients and as such, it is recommended to standardize or mean-center and scale the X data-vectors prior to the regularization.

It should be mentioned that unfortunately if any of the independent X data-vectors contain missing, absent, not-available or non-existent data, as indicated by RNNON, then the RR() automatically reverts to MLR(), as mentioned when lambda equals zero (0.0), and nonlinear regression is employed to simultaneously estimate both the coefficients and any missing X data. Furthermore, the returned coefficient or parameter variances are properly computed as (X' * X + lambda * I)^-1 * X' * X * (X' * X + lambda * I)^-1 * MSE where MSE (or MSPE) is the calculated mean square (prediction) error of the ridge regression.

```
&sDataData,@sValue
DATA,RR(DATAY;DATAGROUPX;LAMBDA;DATAB) *Note, if LAMBDA = 0.0, then MLR() results
DATA,RR(DATAY;DATAGROUPX;LAMBDA;DATAB;DATAZ) *Note that ";DATAZ" is optional
&sDataData,@sValue
```

The (brute-force randomized variable) subset / surrogate selection or sub-system regression (**SSR()**) data function (also known as soft-sensor regression) is a multiple-input and single-output (MISO) sparse (versus dense) regression technique and in spirit is identical to MLR() and RR() (also MISO only) except that SSR() performs an internal and randomized brute-force variable subset selection search or more precisely a randomized enumeration of regressor or predictor candidates to find the "best" multiple linear / ridge regression model or the set, list or vector of non-zero coefficients / parameters that best fits the testing / out-of-sample data given the training / in-sample data. The left-hand-side receives the prediction error / residual or error sum-of-squares (ESS / SSE) for the testing data i.e., actual, measured or observed dependent y output variable (y2, testing data) minus its prediction for all of the selections, shuffles, scenarios, samples, etc. where the model prediction is a function of the linear coefficients or parameters estimated, fit or regressed from the training data. Appended to the very end of the L.H.S. data-vector is the best, least or minimum ESS / SSE value found, its location, index or indice and a terminating RNNON. The user, modeler or analyst must provide the number, size or length of the subset in terms of the number of independent X regressors, predictors, explanatory, feature, input, etc. variables (or basis functions) to be included, the number of (randomized) selections / shuffles (see also the KMS() data function) and the randomization seed integer value. Both the training data-vector for the dependent y1 variable, the data-vector-group containing the data-vectors for the independent X1 variables and the testing data-vector y2 and its data-vector-group X2 are required. If testing data is not available then simply use the training data as this defaults back to the usual training data only regression or estimation. The lambda regularization or ridge parameter may be given with a default value of zero

(0.0) which performs multiple linear regression (MLR) else regularized regression (RR) is provided. The lambda argument can be set to non-zero when there is significant mutual or cross-correlation amongst the independent X variables leading to ill-conditioning (i.e., near-zero singular values) and is added to the diagonal elements of the kernel matrix in the well-known Normal equations i.e., b = (X1 ' * X1 + lambda * I)^-1 * X1' * y1 where X1' * X1 is factorized via Cholesky decomposition. The coefficient (b) results for all of the independent X variables may be retrieved from the vector of parameters equal in size or length to the number of independent X variables found in the training and testing data sets. The last argument on the right-hand-side (R.H.S.) is an obligatory or mandatory data-vector of length or size equal to the number of independent X variables and must contain minus one (-1), zero (0) or one (1). Zero means do not include or exclude the particular X variable, one (1) means that the X variable may be considered as a valid candidate for inclusion into the subset for any selection and negative one (-1) implies that the corresponding X variable must always be included in the subset such as the intercept / constant or certain independent X variables that must be present in the subset or surrogate model.

Please note that the subset size, length or number must exclude the minus one (-1) variables. For example, if the configured subset number is set to five (5) and there are two (2) minus one variables, then the total subset size, length or number is seven (7) meaning that there will be seven (7) non-zero coefficient values in the returned parameter data-vector whereby the two (2) minus one independent X variables will be found in all randomized selections or shuffles. And, if the right-hand-side (R.H.S.) argument for the number of selections or shuffles is negative (-ve), then the 1-norm sum-of-absolute errors or residuals (SAE, ESA) for the testing data is returned in the L.H.S. data vector instead of the 2-norm sum-of-squares of errors or residuals (SSE, ESS). Furthermore, if the R.H.S. seed is negative (-ve), then compute the Ljung-Box portmanteau statistic Q, see also SISOARX(), MLR() and RR(), which is an indication of pure (i.e., independent with no serial or auto-correlation), near-pure or pseudo white noise for the regression residuals or prediction errors and is useful for dynamic, transient, time-series or time-indexed models.

A suggested application of the SSR() is to incrementally, sequentially or successively, starting from the smallest to the largest, set, specify, supply or configure the number of independent X variables in the subset, surrogate or soft-sensor model and to select a modest size for the number of selections to be randomly explored to circumscribe or determine an approximate number of independent X regressors, predictors, features or explanatory variables i.e., the smallest subsets with the smallest SSE or SAE.

Then, refine the search with the smaller subset domain or range and to increase the number of selections or shuffles with varying random seeds to better establish the most important, influential and informative subset of input X variables that characterize, explain or predict the testing data given the training data. This intelligent problem solving approach is also akin to the use of other dimension reducing algorithms such as principal component regression (PCR) and partial least squares / projection onto latent strucutures (PLS), etc. which also require the number of principal components, factors or latent variables to be circumscribed based on the amount of variance explained in the dependent y variable.

If the user, modeler or analyst requires the variance (or parameteric sensitivity information) for the parameters or coefficients similar to the SISOARX(), MLR() and RR(), then the user, modeler or analyst must first call the SCRAPERANGE(…; FLAG) with FLAG = 1 to retrieve only those sparse non-zero coefficient data-vector indexes, indices or cursors from the best subset / surrogate selected returned from SSR(). Once the non-zero data-vector-element indexes are known, then set their corresponding coefficient-setups or paratemter-switches (cf. DATAZ) to one (1) and the zero coefficients / parameters to zero (0) by calling the SUBSTITUTE() data function which will populate DATAZ or the inclusion / exclusion data-set, list or vector necessary to run MLR() or RR(). MLR() or RR() will then properly return the parameter variances for the sparse non-zero coefficients. The reason the supplemental or post analysis data is not automatically computed by SSR() from the training data is to reduce any auxiliary or extra calculating / computing load when the best sub-model or sub-equation is circumscribed or searched from the training and testing data.

Finally, it is appropriate here to highlight that both the Akaike and Bayesian Information Criteria (AIC and BIC) can be employed when selecting a suitable model structure from a number of candidate models. For our purposes, involving least squares regression methods and assuming Normally distributed regression residuals or errors, the normalized or standardized versions of the **AIC = 2 * k / n + LN(ESS/n), AIC,corrected = 2 * k / (n – k – 1) + LN(ESS/n)** and the **BIC = k * LN(n) / n** + **LN(ESS/n)** are found in the well-known book by Box, Jenkins, Reinsel and Ljung (2016) where ESS / SSE is the usual error / residual sum-of-squares, n is the sample, experiment, observation or trial size or length and k is the number of coefficients or parameters (i.e., the number of independent, input, predictor, explanatory or regressor variables) including the intercept, constant, bias, base or balance term in the model. If all of the models have the same k, then selecting the model with the minimum AIC, AIC,corrected or BIC is

precisely the same as selecting the model with the lowest, smallest or minimum ESS / SSE and the well-known **Mallow's Cp** metric is the same as the AIC in the case of linear regression or least squares. As such with the SSR() which requires the user, model or analyst to specify or configure the subset size or length, the SSE, SAE or the Ljung-Box Q statistic is appropriate to determine the best subset of independent / input variables. However, when comparing model structures across a varying number of input variables in the subset, this is where the AIC, BIC and Mallow's Cp metrics come into play to assist in making the tradeoff between the number of coefficients / parameters and the amount or level of variance / variability explained in the training / in-sample data and/or the testing / out-of-sample data. And for interest and completeness, the normalized / standardized Hannan-Quinn Information Criterion **HQIC = 2 * k * LN(LN(n)) / n + LN(ESS/n).**

```
&sDataData,@sValue
DATASSE,SSR(NSUBSET;NSELECTS;SEED;DATAY1;DATAGROUPX1;DATAY2;DATAGROUPX2;
            LAMBDA;DATAB;DATAZ) *Note that ";DATAZ" is obligatory and
                                if NSELECTS < 0, compute 1-norm SAE v. 2-norm SSE or
                                if SEED < 0, compute the Ljung-Box Q statistic v. SAE or SSE
DATASSE,SSR(NSUBSET;NSELECTS;SEED;DATAY1;DATAGROUPX1;DATAY2;DATAGROUPX2;
            LAMBDA;DATAB;DATAZ),FSTRING *Note that ",FSTRING" is optional – see KMS()
&sDataData,@sValue
```

The **SIMULATEPID()** data function is similar to the RETUNEPID() function except that there is no (normalized or standardized) input-move-error (or output-error) norm bound, cut or limit and the return or retrieved data is a datagroup for the closed-loop feedback time-serieses of both the process input (*u, OP*) or the PID controller output and the process output (*y, PV*) or process measured state variable in deviation or perturbation variable form respectively i.e., minus their mean, average, steady-state or equilibrium value. Also refer to TFPDT() and ARIMA() data functions which simulate (i.e., performs the automated calculations) the well-studied Box and Jenkins (BXJK) deterministic and stochastic transfer functions. The right-hand-side data-vectors and calc-scalars are identical to those of the RETUNEPID() except that no process output limiting bounds are respected. The SIMULATEPID() routine simulates the closed-loop feedback behavior of a PID control loop using the single-input-single-output (SISO) auto-regressive exogenous (ARX) dynamic model of the deterministic process and stochastic noise; please refer to the SISOARX() data function. As usual, the SIMULATEPID() simulates the perturbation or deviation variable form of the input and output closed-loop feedback responses only i.e., the variables are in perturbation or deviation form. To recover or return back to the non-perturbation, non-deviation or non-mean-centered form, simply add back the input and output means to the time-serieses as well as

the setpoint time-series and white noise load disturbance is inherently in perturbation / deviation form. SIMULATEPID() is useful to configure or specify the Kp (proportional-gain), Ti (integral-time), Td (derivative-time), Ts (sampling-interval / -instant, scan-rate, Tc, execution/cycle-time or tpd, time-period duration) and the well-know PID controller equation type (i.e., 0 = A = output-error in P, I and D terms, 1 = B = output-error in P and I terms and 2 = C = output-error in I term only) to simulate the performance of virtually any SISO ARX and PID feedback control loop. For interest and in contrast, single-input-single-output (SISO) feedforward control (which may be implemented in static or dynamic form) compensates the controller output or process input (u) based on the measured disturbance or feedforward variables' value and its anticipated, expected or modeled effect on the process output (y) via a transfer function. Another important aspect to mention is whether the PID controller is "direct-acting" or "reverse-acting" (i.e., the latter being the default PID behavior where the controller output or process input decreases as the process output increases) and is determined by the sign of the PID's proportional-gain and depends on the steady-state process gain which must be correctly configured by the user, modeler or analyst to ensure negative feedback i.e., the output-errors (setpoint/target/reference signal minus the dependent process output signal, SP,t – PV,t or ysp,t – y,t) are driven to zero (0.0) or minimized. Stated more succinctly by Skogestad, "Advanced control using decomposition and simple elements", *Annual Reviews in Control* (2023) that the "controller gain should have the same sign as the process gain". And, Skogestad also states that on-off or bang-bang control is essentially proportional-only control with a near-infinite Kp and a near-infinite Ti integral or reset time (and zero derivative or preact time) where a setpoint deadband instead of a fixed setpoint can be used and thereby including hysteresis into the controller. Further, if the supplied white noise load disturbance needs to be replaced by a non-white noise or colored noise load disturbance, then prefiltering the supplied non-setpoint disturbance as discussed for the RETUNEPID() data function may be an option as required. Internally for SIMULATEPID(), the IMPL Solvers' IMPLsimulatePIDability() routine is implemented using the BXJK model form and as such the denominator polynomial usually denoted as A(z^-1) for the exogenous input (typically referred to as OP,t, CV,t, MV,t or u,t) in the SISO ARX parametric model and the white noise load disturbance (a,t or e,t) are the same i.e., y,t = B(z^-1) / A(z^-1) * u,t + 1 / A(z^-1) * a,t given the underlying SISO ARX formula A(z^-1) * y,t = B(z^-1) * u,t + e,t.

```
&sDataData,@sValue
DATAGROUPUY,SIMULATEPID(DATASP;DATAWNLD;DATAARX;DEGREED;DATAPID)
&sDataData,@sValue
```

It requires noting that the data expressions and functions provided by IML's datadata frame are not circular, repeating, looping, iterative nor recursive in nature. Typically, these recursive or iterative computations are classed or categorized as simulation or automated calculations i.e., they do not have single independent, forward, closed or explicit calculations, whereas recursive / recurrence / iterative formulations are dependent, open, backward or implicit calculations similar to the notion of forward/implicit and backward/explicit Euler's (modified) method of ordinary differential equation (ODE) in numeric or algorithmic integration solving. This is the reason why we have a specific and special data function SIMULATEPID() which internally performs a SISO linear and dynamic feedback PID control loop simulation. The same is true for the TFPDT() and ARIMA() data functions which by their nature are also recursive as the present value is a function of past values or the immediate future value (one-step, -interval, -increment or -period ahead) is a function of present and past values.

The relatively well-known fuzzy c-means clustering algorithm (FCM, or "soft" k-means) with missing-, absent- or non-existent-data handling (cf. RNNON) is another multivariable data function callable in IML as **CLUSTERT()** and **CLUSTERW()** found in the paper by Bezdek, Ehrlich and Full, "FCM: the fuzzy c-means clustering algorithm", *Computers & Geosciences*, 1984. The X-block of data to be clustered with CLUSTERT() is contained in a data-group which is the first (1st) argument with the number of X's in the X-block equal to the number of data-vectors in the data-group. The second (2nd) argument is the fuzzification or fuzzifier real number varying from one (1.0) to two (2.0) where a value of 1.01 computes the sharper c-means clustering. The third (3rd) argument is the random seed as a positive integer number which randomly initializes the starting target centroid means in the FCM algorithm and the fourth (4th) argument is the 1-norm or 2-norm type for the clustering objective function which is to minimize either the absolute or squared deviations / errors / residuals of the X variables minus their target, mean or centroid value per cluster. The fifth (5th) and sixth (6th) arguments are the internal convergence tolerance for the FCM algorithm and its internal maximum number of iterations with suggested values of 1D-6 and one hundred (100) respectively. Since the FCM solves a non-convex problem, the number of restarts is required to iterate, loop or traverse over a number of randomized restarts of the FCM algorithm, typically around fifty (50), with the random seed automatically incremented by one (1) for each restart iteration, loop or cycle. The best or global minimum objective function is returned or received in the seventh (7th) or last argument as local minimums do exist given the non-convexity of the weights times targets and there is no guarantee that the supplied number of restarts will encounter the global minimum. The CLUSTERW() data function is similar to the CLUSTERT()

except that technically no restarts are required as the data-group of cluster targets or means is supplied exogenously in the second argument computed exclusively from the CLUSTERT() data function. For on-line or real-time purposes, only CLUSTERW() is required to be invoked to compute the membership weights of the clusters as the cluster targets are known a priori and provided externally via CLUSTERW()'s second data-group argument.

It is important to highlight that the number of clusters, collections, patterns, regimes or zones is determined or specified by the number of data-vectors contained in DATAGROUPT (targets) and the DATAGROUPW (weights) where the maximum number of clusters is limited by the number of observations, samples or trial-points in the X-block or data-group. Please refer to the well-known "elbow method" and the "silhouette score" to aid in the determination of the number of clusters.

```
&sDataData,@sValue
DATAGROUPT,CLUSTERT(DATAGROUPX;
                    FUZZIFIER;RNDSEED;NORMTYP;CONVTOL;MAXITER;NRESTARTS;BESTOBJ)
DATAGROUPW,CLUSTERW(DATAGROUPX; DATAGROUPT;
                     FUZZIFIER;RNDSEED;NORMTYP;CONVTOL;MAXITER;NRESTARTS;BESTOBJ)
&sDataData,@sValue
```

The **RECURSE()** data function is similar in structure to the EVALUATE() and DERIVATIVES() functions with the single right-hand-side (R.H.S.) formula, expression, equation, relation or rule string syntax. The RECURSE() data function supports the ability to configure a known and existing left-hand-side (L.H.S.) data-set, -list or -vector which is recursively computed i.e., revised, modified or updated using the supplied recurrence, recursive or rolling relation, rule, expression, equation or formula. The special character **"X"** (i.e., upper case "x") references or indicates exactly in the same order or sequence of each data-vector member assigned in the single R.H.S. data-group argument i.e., **X1..XN** where N is the number, arity or cardinality of distinct or unique data-vectors contained in the data-group and there must be at least one (1) data-vector included in a data-group although it is not required to appear or be present in the recursive, recurring, recurrent or rolling formula. The L.H.S. data-vector is specifically indicated or referred to by **X0**. Any calc-scalar and any data-vector-element / -point / -row may be included in the recurrence equation as constant coefficients and all data-vectors referenced from X0 to XN must have opening and closing square brackets ("[]") configured immediately after their data-vector reference integer numbers i.e., XN[…]. Inside their square brackets, negative (-ve) / minus-signed index-shifts indicate the integer lag constants or calc-scalar index-expressions where for example XN[0] refers

to the current XN data-vector-element and XN[-1], XN[-2], XN[-3], etc. refer to their immediate previous values. If any positive (+ve) lead index-shift numbers are detected within the square brackets for X0..XN, then an error is raised and reported as lead index-shifting is not currently supported i.e., only explicit lag index-shifting with negative (-ve) values.

These recursive / recurrence / rolling relations or rules are useful to essentially simulate or successively / sequentially calculate data-sets, -lists or -vectors which are functions of their previous values as well as functions of other data-vectors' current and previous rows, points or elements such as simulating user, modeler or analyst defined dynamic and discrete linear and nonlinear transfer functions. Internal consistency and compatibility checks on the data-vector sizes or lengths are performed and if there is any violation, then an error is incurred. Essentially, the X1..XN corresponding data-vectors must be have their size or length equal to or greater than the X0 data-vector. And consistent with the other data functions, if any of the data-vector-elements / -points / -rows are missing, absent, non-existent or unavailable data, then the entire computation is replaced with the RNNON value.

```
&sDataData,@sValue
```
DATA,**RECURSE**(DATAGROUP)**,FSTRING *Note that the FSTRING is outside the parentheses**
```
&sDataData,@sValue
```

Similar to the single-value (multi-input single-output, MISO) XFCN user-, modeler- or analyst-coded functions also referred to as a callback, multi-value (multi-input multi-output, MIMO) DATAFCN's may also be configured and linked to DATAFC1, …, DATAFC9, DATAFCA, …, DATAFCZ whereby the user, modeler or analyst can compile their own C, C++ or Fortran DLL / SO dynamic / shared library functions and call them directly from IMPL in the IML files i.e., adhoc, bespoke, custom or user data functions. These DATAFCN functions may also be called from the computer programming or scripting language that calls IPL (IMPL-API) as well since they are regular C / C++ / Fortran DLL's or SO's. The input arguments to the DATAFCN's are simply a single calc-group and a single data-set / -list / -vector or a data-vector-group and the output or returned left-hand-side argument is a single data-vector of any size or length as coded in the DATAFCN described below:

```
&sDataData,@sValue
```
DATA,**DATAFCN**(CALCGROUP;DATAVECTOR)
DATA,**DATAFCN**(CALCGROUP;DATAGROUP)
```
&sDataData,@sValue
```

Essentially, the DATAFCN may be used to create or convert multiple, stacked or grouped calc-scalars and/or single / multiple data-vectors into a new or existing data-vector quickly and efficiently using a machine-coded computer programming language i.e., C, C++ or Fortran. DATAFCN's may also open and close specifically known files and can read and write from and to these files as required. The call statement or signature for these user-, modeler- or analyst-coded DATAFCN's are as follows where all of the 1D-arrays are linearized or vectorized in order to pass or transfer its multiple data-vector-elements, -points, -rows or -values across diverse computer programming and scripting language.

```
integer function DATAFCN(ncse: integer,
                         cse: real*ncs,
                         ndv: integer,
                         ndve: integer,
                         dve: real*(ndv*ndve),
                         ndve2: integer
                         dve2: real*MIN(ndve2,ndv*ndve))
```

```
      function datafunc(ncse,cse,ndv,ndve,dve,ndve2,dve2)
#if stdcalling == 1
cDEC$ ATTRIBUTES DLLEXPORT, STDCALL, REFERENCE, DECORATE, ALIAS : "datafunc" :: DATAFUNC
#else
cDEC$ ATTRIBUTES DLLEXPORT, ALIAS : "datafunc" :: DATAFUNC
#endif

c * IMPORTANT NOTE * - IMPLserver.mod and IMPLserver.lib are optional.
c
c        use IMPLserver

        implicit none

        integer(4) :: datafunc

c * IMPORTANT NOTE * - IMPLmodeler.fi is optional.
c
c        include "IMPLmodeler.fi"

c Number of calc-scalar elements (values) and the real number elements.

        integer(4), intent(in) :: ncse
cDEC$ ATTRIBUTES VALUE :: ncse
        real(8), intent(inout) :: cse(1:ncse)
cDEC$ ATTRIBUTES REFERENCE :: cse

c Number of data-vectors and the equal number of real number data-vector elements (values) for
c each data-vector in linearized or vectorized form.

        integer(4), intent(in) :: ndv
cDEC$ ATTRIBUTES VALUE :: ndv
        integer(4), intent(in) :: ndve
cDEC$ ATTRIBUTES VALUE :: ndve
        real(8), intent(inout) :: dve(1:ndv*ndve)
cDEC$ ATTRIBUTES REFERENCE :: dve

c Number of output or returned real number data-vector elements and the returned real number
c data-vector element (values).
c
c * Note that we need to define/declare the output array dve2 dimension as ndv*ndve instead of
c   ndve2 since ndve2 is not known
```

```
c   as it is an output argument.  Therefore we are tacitly assuming that the maximum length of
c   the returned or output data-vector dve2's dimension can be no more than ndv*ndve.

        integer(4), intent(out) :: ndve2
cDEC$ ATTRIBUTES REFERENCE :: ndve2
        real(8), intent(inout) :: dve2(1:ndv*ndve)
cDEC$ ATTRIBUTES REFERENCE :: dve2

c ... Insert developer user, modeler or analyst code here ...
c ...

c ...
c ... Insert developer user, modeler or analyst code here ...

        datafunc = 0

c ... Insert developer user, modeler or analyst code here ...
c ...

c ...
c ... Insert developer user, modeler or analyst code here ...

      end function datafunc
```

It is worth pointing out that the last argument (dve2) is a 1D-array that has intent of "inout" (cf. Intel Fortran which states that the argument is both an input and an output) meaning that if the L.H.S. data-vector exists prior to the external data function call, then it is populated with data-vector-element values immediately prior to the DATAFCN call. This is useful when computing simple or sophisticated recursive, recurring or rolling equations, expressions, formulas or relations for example (see also the RECURSE() data function). In addition, although both the cse and the dve arguments may have their intent's set as "inout" as opposed to "in" only, which means that any of their values may be altered, changed or modified inside the DATAFCN's source code, at present only the dv2 argument's data-vector-element values (dve2) are received or updated back into IMPL after the DATAFCN call has completed.

A very important and useful capability of the DATAFCN() callback's is that if programmed or coded with Intel Fortran (IFX), the IMPLserver.mod, IMPLserver.lib and IMPLmodeler.fi, provided by Industrial Algorithms Limited (IAL) upon request, may *optionally* be used and included when compiling and linking the DATAFCN() DLL / SO dynamic link library – see also the IMPC Manual. This means that the user, modeler or analyst has complete / full access to all of the IMPL data structures and routines / methods when IML is called or invoked by IMPL through the IMPLinterfaceri() routine.

The data function frame below configures or registers the path name and file name for the location of the DLL / SO library and its function name inside the library file where if the path name is absent, missing or not preset, then the path name of the IML file is assumed. The "@" represents the integer

number index field of the data functon: **0** (DATAFCN), **1** (DATAFC1), …, **9** (DATAFC9), **10** (DATAFCA), …, **23** (DATAFCN), …, **35** (DATAFCZ). Given that these DATAFCN's are not employed in the optimization nor estimization of the problem, only the pre-processing of its data in IML, the same DATAFCN may be dynamically re-assigned with different path, library and/or function names at any place, position or location in the IML file.

```
DATAFCN-@sPath_Name,@sLibrary_Name,@sFunction_Name
DRIVE:\DIRECTORY\,LIBNAME.DLL,FUNCNAME,@
,LIBNAME.DLL,FUNCNAME,@
DATAFCN-@sPath_Name,@sLibrary_Name,@sFunction_Name
```

It should be highlighted that DATAFCN's may themselves call internally any of the IMPL Solvers' routines for example listed in the SSIIMPLE Infrastructure Manual as external or extrinsic functions and subroutines (or any other third-party DLL / SO) and can be used to automate, replicate, enhance, customize, protect, privatize, etc. sections of the IML file configuration by user, modeler or analyst developed C, C++ or Fortran source code compiled into machine-code. As well, since IMPL's internal data functions or operations do not support any explicit do-loop or for-loop recursive / iterative programming constructs, these external / extrinsic data functions provide the necessary capability for the user, modeler or analyst to extend IML's functionality where needed in a machine-coded DLL / SO for both speed and efficiency albeit at the expense of convenience and clarity in the sense that a mixed-language approach is required.

Similar to and consistent with the ILP / INP model function and formulary REPEAT() and REPLICATE"" respectively, the data formulary's **SPIN"", SPIN2""** and **SPIN3""** may be configured to systematically "spin" up, so to speak, multiple calc-scalars (DataCalc frame) and data-sets/-lists/-vectors (DataData frame) as well as data-groups (DataGroup frame) from a given formulary or base, root, prototype or signature formula using the special symbol characters **"$", "&"** and **"@"** respectively as shown immediately below. In essence, the SPIN"" data formulary is both similar to the REPEAT() model function using the same dedicated special characters ("$", "&" and "@") which internally creates or generates multiple datacalc and datadata frame features, rows or lines and the REPLICATE"" model formulary in terms of its syntax using the double-quotes as its delimiter as opposed to the left and right parentheses ("()") which are reserved for the data functions. In addition, the SPIN"" data formulary respects the **"%nnn%"** special symbols for lead (+nnn) / lag (-nnn) index-shift operators identical to the

REPEAT() and REPLICATE"" model functions and model formulary's where the nnn may also be a calc-scalar and/or data-vector-element expression / formula.

The SPIN"" data formulary's supports dense iterator-triples or tuples i.e., the **start;stop;step;** but also sparse iterator-sets, -lists or -vectors i.e., **set;span;** otherwise known as index-sets, -lists or -vectors. If step; is negative (-ve) for dense indexing, then the sequence or series starts at stop and increments in descending or decreasing order as opposed to the default ascending or increasing ordering of the indices. The set; must be a known data-set, -list or -vector (converted to integer numbers) and the span; is required to properly prefix the appropriate number of zeros (0's) for the leading-zero indexing similar to stop; and can be likened to a set's / list's / vector's domain or range of elements / members. Any combination of dense- and sparse-iterators are supported with the SPIN2"" and SPIN3"" data formulary's. **Please note that if the SPIN's stop (upper, tail) sub-iterator is less than (<) the SPIN's start (lower, head) sub-iterator, then SPIN simply skips, passes-over or ignores the statements inside the SPIN double quotes.**

`&sDataCalc,@sValue` OR `&sDataData,@sValue`
$DATA$,**SPIN**"start;stop;step;$DEXPRESSION$"
$&DATA$&,**SPIN2**"start;stop;step;start2;stop2;step2;$&DEXPRESSION$&"
$&@DATA$&@,**SPIN3**"start;stop;step;start2;stop2;step2;start3;stop3;step3;$&@DEXPRESSION$&@"
SPIN"start;stop;step;$DATA$,$DEXPRESSION$"
SPIN2"start;stop;step;start2;stop2;step2;$&DATA$&,$&DEXPRESSION$&"
SPIN3"start;stop;step;start2;stop2;step2;start3;stop3;step3;$&@DATA$&@,$&@DEXPRESSION$&@"

$DATA$,**SPIN**"set;span;$DEXPRESSION$"
$&DATA$&,**SPIN2**"set;span;set2;span2;$&DEXPRESSION$&"
$&@DATA$&@,**SPIN3**"set;span;set2;span2;set3;span3;$&@DEXPRESSION$&@"
SPIN"set;span;$DATA$,$DEXPRESSION$"
SPIN2"set;span;set2;span2;$&DATA$&,$&DEXPRESSION$&"
SPIN3"set;span;set2;span2;set3;span3;$&@DATA$&@,$&@DEXPRESSION$&@"
`&sDataCalc,@sValue` OR `&sDataData,@sValue`

When the "$", "&" and "@" special characters or symbols are found in the data expression or formula string, they are automatically replaced with leading-zeros index numbers determined by the size of their respective stop (or span) number in terms of the number of individual digits or figures it represents as an integer number. Any data function and any calc-scalar / data-vector algebra may be configured in the data formulary expression though calc-scalar and data-vector symbol names must not of course include these special characters as they will confound / confuse the special character substitutions /

replacements. **It should be pointed out that negative (-ve) index numbers are not supported consistent with the required non-negative foreign-variable and -constraint index numbers (Xnnn and Fmmm) and data-sets / -lists / -vectors starting at index one (1) only.**

The SPIN"" data formulary is also useful to create, generate, synthesize or systematize many different but related calc-scalars / data-vectors with the same argumented, tokenized or parameterized formula contained inside datacalc and datadata frames by simply configuring or specifying a single feature, line or row in the IML file. For interest, internal to the IML file compiling / processing by the IMPL Interfacer routine, a multitude of internal feature lines or rows are programmatically created within three (3) layered or nested do- / for-loops i.e., one for-loop for each subscript or iterator tuple. That is, "$" is for the outer loop, "&" is for the middle outer / inner loop and "@" is for the inner most loop respectively.

Concatenation (Companion) Data

IMPL allows any number of include files (likewise referred to as companion or concatenation files) to be specified anywhere in the IML file, with any file path, name and type or extension configured, but only one include file may be nested or embedded within another include file i.e., IMPL supports only a two-level or two-layered include file depth. In addition, one and only one include file feature, row or line may be present in between or inside each include file frame's leader and trailer features, and as mentioned, at most two (2) include files may be opened at a time i.e., the upper- or higher-level primary include file, and if configured, its lower-level secondary nested or embedded include file.

Include files are very useful / handy for the user, modeler or analyst to have one main, master or overall IML file with several embedded or included individual sub-IML files. These individual sub-IML files are sometimes referred to as "IMLet" files similar to "commandlets" or "cmdlets" in Microsoft's Powershell i.e., fact.imlet or subject.imlet although the file type or extension is completely configurable at the discretion, preference and choice of the user, modeler or analyst. These included IMLet files are also useful to enable a rudimentary form of structured language programming with IML where specific IMLet's can be created to segregate, partition, group and organize the overall, master, primary or main IML file. Of interest, there are other "et" files used within IMPL such as ILPet, INPet and OMLet.

```
Include-@sFile_Name
```

DRIVE:\DIRECTORY1\DIRECTORY2\FILENAME.EXT,@,@,@,@,@
`Include-@sFile_Name`

The file name must specify the drive, directories and extension delimited by either "\" (backslash, ASCII code 92) or "/" (forwardslash, ASCII code 47) if a path name is required. If the include file cannot be found as-is in the first field of the include file frame above, then IMPL will prefix any extracted path name from the IML file name (i.e., fact.iml / subject.iml). If after the second attempt the include file cannot be opened, then a file not found error will occur. If there is no path name specified with the IML file name, then the assumed directory is the working directory for the second try of opening the include file.

Please see the INCLUDETRACEIMLFILE setting in the IMPL.set file which if set to one (1) will be output, echo, shadow or trace to a subject.tml file, all of the lines read into IMPL Interfacer including the include file frame. This is helpful for the user, modeler or analyst to observe in a single file, all of the IML content read, imported or loaded from all included files. In addition, include file indirection is also possible via the IMPL Console file1..file9, fileA..fileZ flags where if the include file's name is replaced by the IML keyword **INCLUDEFILE1, …, INCLUDEFILE9, INCLUDEFILEA,…, INCLUDEFILEZ**, then IMPL will substitute the path name, file name and file extension specified in the corresponding file1, …, file9, fileA, …, fileZ flags. Indirecting the include file's name is useful when the include file's name changes from IMPL execution-to-execution or run-to-run; for example when real-time data is inputted, imported or read.

It is also possible to add two hidden and augmented / appended string fields after the include file name in the single feature of the frame such as the first two (2) "@,@" above and this is to both prefix and suffix any aliases (see convenience data) found in the include file i.e., the first "@" is the prefix string fragment and the second "@" is the suffix string fragment. An aliases must be configured completely within the same frame so as to make their alias names unique and thus the prefix and suffix string (hidden, augmented / appended) fields available here allow this. Therefore, if there are multiple IML include files to configure the problem's construction data and each IML include file has aliases with the same names, then the user, modeler or analyst must change the names of the aliases in each include file using these (hidden) prefix and suffix fields provided. Of course, throughout the rest of the IML file these altered, changed or modified alias names must be configured / specified to reference the aliased construction data accordingly.

There are also the IML keywords called **CALCFRAME** and **DATAFRAME** which if found in the **second (2nd) field / first "@"** of the include file feature, row, record or line will simply, internally and implicitly supply or add the calc or data frame leader and trailer features i.e., &sCalc,@sValue or &sData,@sValue respectively. Only configure CALCFRAME or DATAFRAME when the file to be included and read, imported, inputted or loaded does not have these two (2) begin / start / initial and end / finish / final features (i.e., referred to as the IML frame leader and trailer features) and all of the feeder features in the include file are expected to be valid for calc or data frames. The **third (3rd) field / second "@"** of the include frame's feature, the user, modeler or analyst may configure or specify a field or column index number bounded from one (1, default if left blank) to nineteen (19 = 20 - 1) relating to the value column or position in the file to be included given that the include file may have multiple columns of data values on the right-hand-side of the data-vector name. This third field / second "@" is a 4-byte or 32-bit integer and may also be a calc-scalar / data-vector-element expression or formula. For data frames (DATAFRAME) only, the **fourth (4th) and fifth (5th) fields / third and fourth "@"** of the include-file frame's feature are the prefix and suffix character strings which are concatentated to every included data frame's data-set / -list / -vector names to make them unique or distinct (i.e., previously unknown to IMPL) especially when the same include file is read, inputted, loaded or imported multiple or many times with different value column index numbers to load from different data columns. **It is important to recall that the prefix and suffix characters are required as known or existing data-sets, -lists or -vectors will not be imported, read, loaded or inputted if they already exist as IMPL's data-vector processing expects all data-sequences-, -signals or -series to be unknown prior to parsing the data frame.** Of course, if the fourth and fifth fields / third and fourth "@" are not represented by their comma delimiters, left blank or contain blank spaces, then no prefix and suffix character strings are added. There is also a dense and a sparse version of the single-column and multi-data-vector format. The default is the sparse version and if the **sixth (6th) field / fifth "@"** is greater than zero (0, default) or one (1), then the dense version is configured where it asserts or assumes that every data-set / -list / -vector must have exactly the same number of elements, points or rows (or values) in the include-file file to be inputted, loaded, read or imported when the DATAFRAME keyword is present.

It should be highlighted that other hidden frame-feature fields are supported in IML as highlighted throughout the manual(s) and are indicated by the "@" special character which represents integer, real or string hidden fields either in the leader or more typically the feeder features, rows or lines. These

hidden fields are potentially available for any IML frame and allow for extra address and/or attribute data (delimited by commas) to be configured into IMPL when there is a special requirement to do so without incurring the extra overhead of incorporating a new separate and specific IML frame.