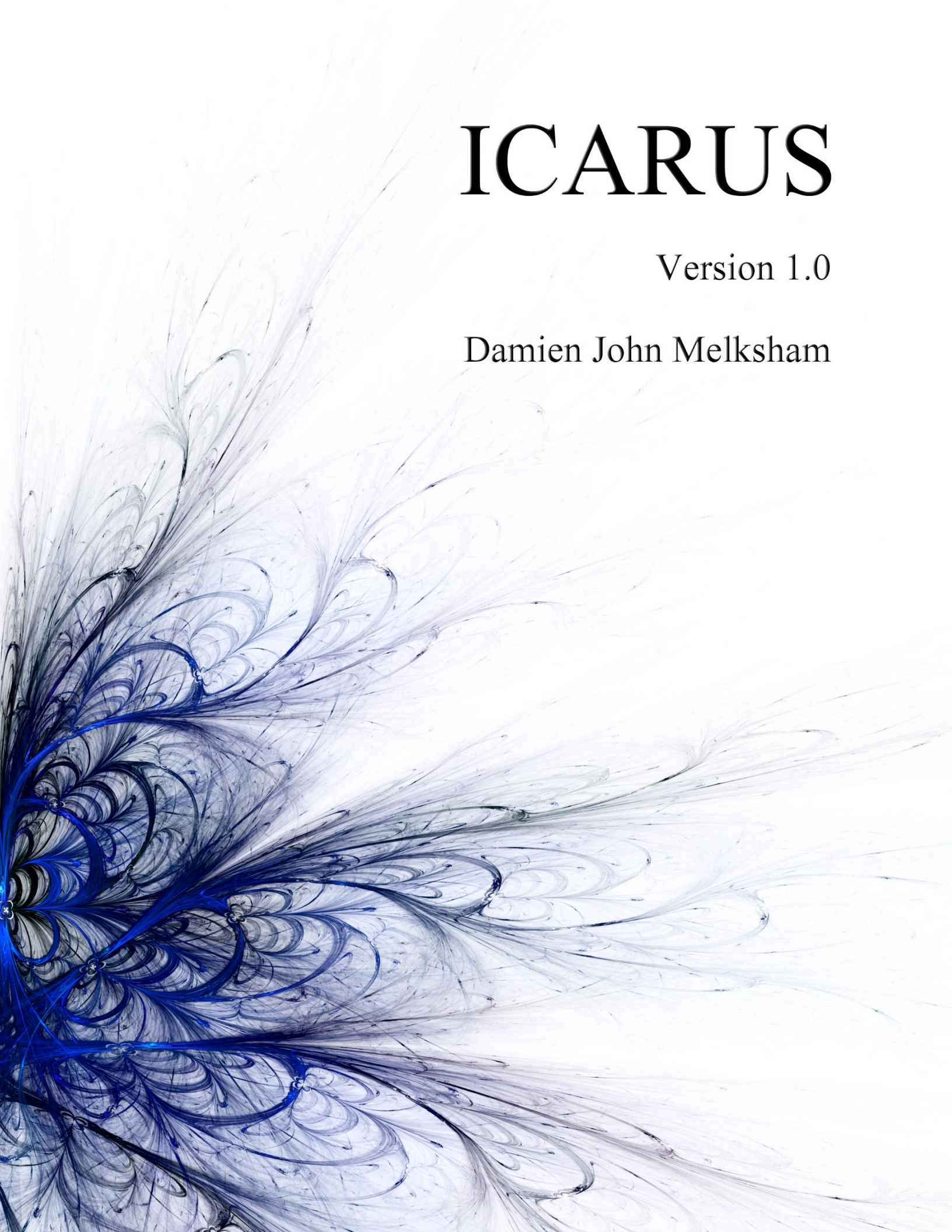


ICARUS

Version 1.0

Damien John Melksham



ICARUS

A SYSTEM FOR DATA LINKING

VERSION 1.0

DAMIEN JOHN MELKSHAM

29TH MAY 2013

Table of Contents

Introduction: What is Icarus?	11
Icarus: A more detailed explanation	12
Operational Information and Setup	14
Utility and Programming Macros	20
applyid	21
<i>Apply a numerical ID variable to a data set</i>	
commalist	22
<i>Convert a space delimited list to a comma delimited list</i>	
countwords	23
<i>Count the number of words in a string</i>	
createsimpledataset	24
<i>Create a simple data set from macro variables or text</i>	
deletedsets	25
<i>Delete a list of data sets or views</i>	
deleteprograms	26
<i>Delete a list of precompiled data step programs</i>	
dsetparse	27
<i>Obtain the data set part of a library.dataset reference</i>	
dsetvalidate	28
<i>Determine whether a data set exists</i>	
dsetvrename	29
<i>Rename the variables on a data set</i>	
estimatesize	30
<i>Estimate the memory required to store variables from a dataset</i>	
filesindir	31
<i>Obtain information on files in a directory</i>	
findreplace	36
<i>Perform find and replace operations on text and macros</i>	

getoption.....	37
<i>Programmatically obtain SAS option details</i>	
interleave	37
<i>Interleave two space delimited lists</i>	
keepwordpattern	38
<i>Keep words from a list that match a regex expression</i>	
lengthfixer	39
<i>Automatically minimise the length of character variables on a data set</i>	
libnameparse.....	40
<i>Obtain the library part of a library.dataset reference</i>	
libvalidate	41
<i>Determine whether a library reference has been assigned</i>	
memsize	42
<i>Obtain the value of the SAS memsize option</i>	
numofobs	43
<i>Obtain the number of observations in a data set</i>	
numordset.....	44
<i>Determine whether a string starts with a number or a letter</i>	
numstochars.....	45
<i>Create data step syntax to convert numeric variables to character variables</i>	
obtomacro.....	47
<i>Import an observation from a data set into a macro</i>	
pl.....	48
<i>Add a prefix to words in a space delimited list</i>	
progvalidate	49
<i>Determine whether a compiled data step program exists</i>	
qclist	50
<i>Convert a space delimited list of words into a comma delimited, double-quote-surrounded list of words</i>	
quotelist	51
<i>Convert a space delimited list of words into a double-quote-surrounded list of words</i>	

realmem	52
<i>Obtain the value of the xmrlmem SAS option</i>	
removelibraries	53
<i>Deassign a list of libraries</i>	
removewordfromlist	53
<i>Remove a word from a list of words</i>	
repeater.....	54
<i>Repeat a word N times</i>	
repeaterandnum	55
<i>Repeat a word N times and append an incrementing number to the end of each word</i>	
report_date	55
<i>Resolve to the current date</i>	
report_datetime.....	56
<i>Resolve to the current date and time</i>	
report_time	56
<i>Resolve to the current time</i>	
rpl	57
<i>Remove a prefix from each word in a space delimited list of words</i>	
rsl.....	58
<i>Remove a suffix from each word in a space delimited list of words</i>	
sl	59
<i>Add a suffix to each word in a space delimited list of words</i>	
termlistpattern	60
<i>Implement a common programming pattern</i>	
tvtdl	61
<i>Implement a common programming pattern</i>	
uniquewords	62
<i>Return unique words from a space delimited list of words</i>	
varkeeplist	63
<i>Keep words from a list based upon on a list of 1's and 0's</i>	

varkeeplistdset	64
<i>Keep variables from a data set based upon an observation of 1's and 0's in the data set</i>	
varlengths	66
<i>Return the lengths of variables in a data set</i>	
varlistfromdset	67
<i>Obtain a list of the variables in a data set</i>	
varpos	68
<i>Obtain the position of a variable within a data set</i>	
varsindset	69
<i>Determine whether a list of variables are found in a data set</i>	
vartype	70
<i>Obtain a list of the types of variables in a data set</i>	
vordset	71
<i>Determine whether a data set is a view or a physical data set</i>	
wordinlist	72
<i>Determine whether a word is contained in a space delimited list of words</i>	

New SAS Functions.....73

Caverphone 2.0	74
Chebyshev Distance	76
Cityblock Distance	78
Double Metaphone	80
Euclidean Distance	82
Expectunique.....	84
Fuzznums.....	85
Hamming Distance	87
Jaro/Winkler	89
Minkowski Distance	92
NYSIIS	94

Hash Macros..... 96

hashcount.....	97
<i>Output counts of variable values within a data set</i>	
hashdistinct	101
<i>Output the unique variable values within a data set</i>	
hashisin.....	104
<i>Output values from one data set that are in another data set</i>	
hashisnotin	106
<i>Output values from one data set that are not in another data set</i>	
hashjoin	108
<i>Automatically perform a hash-based join between two data sets</i>	
hashsort.....	122
<i>Use the hash object to sort a data set</i>	
hashsum	125
<i>Output the sum of a variable from a data set grouped by the values of other variables</i>	

Blocking and Indexing Macros..... 127

icarusindex	128
<i>Manually create an Icarus index</i>	
icarusindexdset	132
<i>Create Icarus indexes using a control data set</i>	
icarusindexjoin	136
<i>Use Icarus indexes to perform a complex join operation between two data sets</i>	
multihashjoin	143
<i>Use multiple hash objects to perform a complex join operation between two data sets</i>	
multihashpointjoin	149
<i>Use multiple hash objects holding pointers to perform a complex join operation between two data sets</i>	
snhood.....	155
<i>Perform a sorted neighbourhood operation on a data set</i>	
plainblocking	166
<i>Perform a standard blocking procedure on two data sets</i>	

Icarus Data Linking Macros 170

apderive.....	171
<i>Derive common agreement patterns from two data sets</i>	
app.....	179
<i>Produce all agreement patterns/combinations</i>	
apsummary.....	183
<i>Output a summary of agreement patterns</i>	
djmassignment.....	186
<i>A one to one assignment macro</i>	
dsetslicer	196
<i>Partition a data set into a number of smaller data sets</i>	
dsetsmoosh	200
<i>Combine partitioned data sets into one larger data set</i>	
genprob	203
<i>Generate probabilities from summarised agreement patterns</i>	
genweight.....	208
<i>Generate weights from generated probability files</i>	
hashmapper	212
<i>Perform the Icarus hashmapper operation on two data sets</i>	
icarus_em	216
<i>Use the EM algorithm to generate M and U probabilities from summarised agreement patterns</i>	
local_fam.....	222
<i>Identify independent bipartite graphs within a larger supplied bipartite graph</i>	
ngramdsetletter	227
<i>Place ngrams of letters in a variable onto a data set</i>	
ngramdsetword.....	230
<i>Place ngrams of words in a variable onto a data set</i>	
ngramletterssummary	233
<i>Summarise the letter ngrams in a variable into a separate summary data set</i>	
ngramwordsummary.....	236
<i>Summarise the word ngrams in a variable into a separate summary data set</i>	

optimiseap.....	240
<i>Optimise a set of agreement patterns for use in other Icarus joining macros</i>	
pointy	244
<i>Use point data in one data set to join records in two other data sets</i>	
royalsampler	248
<i>Randomly bring together records from two data sets</i>	
simpleencrypt.....	251
<i>Use the MD5 hash to encrypt selected variables on a data set</i>	
simpleevidence	254
<i>Use weight files on joined data sets to output weighted record comparisons</i>	
topn	259
<i>Output the top N records from a data set of weighted record pairs</i>	

Distributed Computing Macros 263

icarus_addnode.....	264
<i>Establish a control data set with settings for remote sessions</i>	
icarus_connect	269
<i>Use a control data set to automatically set up multiple remote sessions</i>	
icarus_distcode	271
<i>Distribute code to multiple remote sessions and run it</i>	
icarus_distslice	272
<i>Partition a data set and automatically distribute the partitions amongst the remote sessions</i>	
icarus_distsmoosh.....	275
<i>Recombine partitioned data sets that have been distributed amongst remote sessions</i>	
icarus_distvar	278
<i>Distribute a global macro variable to the remote sessions</i>	
icarus_distvardel	279
<i>Delete a global macro that exists in all of the remote sessions</i>	
icarus_germinate	280
<i>Temporarily upload, compile, and setup Icarus functionality on a remote session</i>	

Closing comments, deliberate omissions and a word on further techniques	281
Thanks and Acknowledgements	288
Contacting the Author, Licensing and Use	290

Introduction: What is Icarus?

Icarus consists of a collection of meta-programs, functions and algorithms primarily for use with Base SAS¹ software. These programs were designed for application in the field of data linking, but can also be applied to many other tasks. The areas of efficiency, flexibility, and scale were specifically targeted relative to other data linking products and SAS macros.

Data linking can be understood as the task of finding information relating to the same entity across disparate sets of data, and has been historically referred to by many different names. A trivial example involves finding information on a subject from multiple databases using high quality identifiers such as social security numbers, birth certificates, and registration details. More complex and valuable applications involve the same activity when unique identifiers are unavailable or do not explicitly exist.

There is a long history of using data linking to identify persons in birth, death, and health records for medical research, but there is no reason to limit ourselves to health studies or physical persons. Financial transactions, electronic fingerprints, households, publications, institutions, businesses and other abstract classes fit just as easily in the definition of entity and as suitable targets for data linking.

Icarus itself was written due to my natural curiosity and tendency towards architecture, experimentation and improvement. I wanted to address the shortcomings and frustrations I experienced with current data linking software, theory and methods.

I designed Icarus to deal with realistic nation-sized problems. Performance does not noticeably suffer when applied to small problems; on the contrary, the main implication with small projects is that traditional resource saving techniques can largely be ignored.

Icarus was created using my own resources. No funding, grant, body or external direction was responsible. This has meant practical obstacles on my behalf, but it allows me to retain intellectual property rights and control the design of the entire project as a technical expert in the field.

¹ I am not affiliated with SAS software or the SAS Institute in any way, nor do I claim to speak for them or represent them or their products in any fashion.

ICARUS: A MORE DETAILED EXPLANATION

Why SAS?

The decision to use SAS² to build a new data linking system was based on several key points:

1. It is an environment I was already familiar with.
2. For the tasks at which Icarus would be targeted, SAS was already commonly present. SAS products are a key component in many projects of appreciable size specifically because it is suited to extracting, manipulating and standardising large amounts of data.
3. SAS architecture doesn't rely upon loading data sets into memory by default.
4. SAS possesses a convenient model regarding macro-code, predefined meta-data, and natural breaks for cyclic write/compile/run cycles. This allows the expert to write flexible and dynamic programs that blur the line between data and code, and compile and run-time. Expertly crafted programs are able to operate extremely quickly.
5. SAS operates on a wide variety of operating systems and environments.
6. SAS has taken care of the boring and tedious aspects of implementing such a system, by which I mean interfaces and specifications for locations, files, databases and datasets. With SAS Enterprise Guide, SAS even provides a workable graphical user interface in which to plan and construct an entire project, though Icarus does not enforce its use.

It could be said that Icarus is primarily composed in the SAS macro language, but this is a simplification. The macro language is used to write programs in other SAS languages, and is not the goal itself. The generated code of Icarus consists of SAS data steps, SQL, Perl regular expressions, custom functions via PROC FCMP, a subset of C, and also the macro language itself. It is not necessary for the user to understand all these sub-languages. Rather, the macros write code specifically crafted for the task at hand.

² On a more jovial note, when I wrote most of Icarus I was unfamiliar with the LISP family of languages. Now, looking at all the space delimited lists fed to macros, I cannot help but reflect that I have "Greenspun'ed" SAS. If point 6 and historical precedence were not an issue, I'd be very tempted to jokingly borrow from one of Randall Munroe's comics and conclude: "Should we write it all again, I'd end it with a close-paren".

Why Icarus?

Icarus developed from my frustrations working on data linking research and projects.

Current programs written in Java, R or Python were usable in an idealised sense. But with real world dimensions and complexities I found all left something to be desired. Performance, flexibility, and ability to scale always became problematic.

I understood that the US Census Bureau had its own programs written in C for large scale Census data linkage projects. But with the benefits also comes the cost of C; complexity, verbosity, security, flexibility, etc. It also doesn't help that modern computing and statistical education seems to involve the non-conscious repetition of "C is a dead language."

In practice, I found many people ended up relying on SAS to perform tasks at some stage of the process. Whether it was cleaning and standardisation, extraction of data, conversion of data between formats, or statistical analysis and modelling on the resulting data, the involvement of SAS seemed to be relatively constant in a large scale professional environment. Perhaps SAS was not used for the data linking itself, but it was commonly present for at least part of the process. The importing and exporting of data from SAS merely added another point of failure to already complicated processes.

Coding ad-hoc data linking projects in SAS is usually prohibitively complex to say the least. Optimisation and efficiency then adds another layer of complexity on top. Hence, I understood all too well why it wasn't commonly done. But with a curious and non-traditional mindset, and working in a research and methodological area, I had additional frustrations with the status quo.

I wanted to write arbitrarily complex algorithms, or just throw the paradigms out the window. I needed the freedom to customise research and projects to fit my own desires. I wanted programs that were transparent, that had their basic operations and techniques documented, but which didn't achieve this by locking the user into one set method or technique.

I wanted to be able to work with any subject matter in which data linking was applicable, and not only on the comically small and easy problems found in academic literature.

I hoped that with my own experience I could achieve this while providing equal or superior performance compared to its contemporaries. Looking for solutions for my problems, I soon found myself with designs and ideas for an entire customisable data linking environment in SAS, and subsequently, Icarus was born.

OPERATIONAL INFORMATION AND SETUP

About This Document

This document consists of basic documentation of Icarus. It records what each part does and tries to give the necessary information and theory on how it operates. It also provides some tips and a few examples to learn from which can also be used as test cases.

The goal is not to teach you how to think, nor to understand traditional data linking theory, nor supply you with prerequisite strategies for how all the pieces could fit together.

That is left up to you.

Installation

The default Icarus installation method is itself a macro: %icarus_install.

Icarus_install defines and compiles all requisite macros and the new SAS functions in a SAS session. It sets the noquotelenmax option, and if there is no cmplib option set, icarus_install sets this for the user so they can use the Icarus SAS functions. By default, the custom functions are compiled and stored in the work library. If you wish to install these functions to another location, this library must be defined before calling icarus_install.

If the macro detects that there are already settings in the cmplib option, then it is up to the user to change their cmplib option to include a reference to the compiled function data set produced while retaining their current settings to their satisfaction. The data set output by the macro is called icarus_functions.

The macro comes with two parameters. The first parameter is called Location. This needs to be the system path to the directory/folder where icarus_install is located. Please remember to supply this location with a trailing slash/backslash as appropriate.

The second parameter is called Functionlib, which details the library to which the custom functions will be compiled and stored. If it is not supplied by the user, it defaults to work.

Icarus also comes natively with a file of keyboard macros for SAS Enterprise Guide, which allow a template for each macro to be produced by typing a macro's name without a leading percent sign. However, use of the keyboard macros or SAS Enterprise Guide is not compulsory.

Example Icarus_install on a windows computer

```
%icarus_install(Location=C:/Users/AUserName/Desktop/, Functionlib=work);
```

Icarus_install was actually constructed with a Common Lisp program, which pulls macro development source code together from individual development files into one giant macro. In doing so, it removes formatting, superfluous white space, and parses out all source code comments to make the final installation file small and lightweight. However, it does have the side effect of making production source decidedly unfriendly to look at.

The good news, from my point of view, is that if a particular user has a design or need for a different implementation or method of installing or using Icarus, I may be able to change this program to easily produce a file/installation according to the user's specification.

Not a SAS Manual

Icarus is an extension of an analytical system. All data sets can be operated on with regular statistical procedures, programs, and manipulations just like any other part of SAS.

There is nothing stopping you from placing Icarus commands in your own macros, calling Icarus functions in your own code, or running SAS procedures on output data sets for analysis. You can use parts of Icarus to make your regular SAS operations more powerful and flexible. After a while you might wonder how you ever programmed in SAS without them.

I won't be re-documenting techniques and procedures already in SAS in this document. This shouldn't make you conclude that an action isn't possible.

Base SAS is one of the best documented products on the planet. Nine times out of ten, someone has performed the task you wish to perform, and they've written and uploaded a succinct and well-written document showing you how to do it. When you combine the included abilities of SAS with the additional programs provided in Icarus, you begin to realise just how powerful such a system can be.

Icarus dos and don'ts

There are several options in SAS that directly apply to Icarus.

Warnings for strings longer than 262 characters are automatically turned off

SAS has an option to warn the user if it obtains a quoted string longer than 262 characters.

Since Icarus consists of programs that write programs, and 262 characters is not particularly taxing for a computer, installation of Icarus automatically disables this option.

The SAS log file and messages written to the log:

SAS prints information to a log file. I have ensured that many macros perform common internal error checks during their operation, and some will even return relatively coherent error messages when they fail.

However, what seemed like a good amount of documentation for human written programs can now become intolerably large. Rather than meddle in your affairs by changing settings from within the macros, I let you retain control. Make yourself familiar with the options for controlling the verbosity and inclusion of messages in the SAS log file.

As a loose guide, I recommend always turning off the MPRINT and the MLOGIC options unless needed for debugging. Icarus is primarily composed of interacting macro programs, variables and functions, and printing macro resolutions and logic during operation will quickly make the log enormous and indecipherable.

You may also wish to consider experimenting with the NOSOURCE option to disable the printing of some of the generated source code Icarus produces.

The status of the NONOTES option I leave up to the user's discretion. Many of the macros may print useful notes to the log by default, but you may not be interested in them.

I do not, as a general rule, ever recommend suppressing errors and warnings. It is always good practice to generally test routines and segments of programs individually rather than running them all at once and assuming they will work, especially when dealing with large scale data.

As with all of Icarus, I encourage experimentation and customisation to your own ends.

Reserved variables/names

I went to great lengths compared to most SAS macro writers in the design and creation of Icarus to ensure macro variables behave well. As much as possible remains compartmentalised and localised in scope. There should not be cases of Icarus leaking macro variables into global scope unnecessarily or interacting incorrectly with other macros. Icarus tries to be a polite neighbour: it puts temporary data in the work library by default, and attempts to clean up temporary data if it is no longer needed.

However, I do wish to stake a claim on names to ensure operations are as smooth and efficient as possible.

As a general rule, do not use data sets, variables, or arguments that start with the prefix `_djm`, `_icarus`, or `_ic`. These are considered reserved by Icarus for its own internal operations.

The phrase `_DJM_NONE` is particularly of note. Icarus commonly uses this phrase internally as a default value for variables that are not supplied, in order to differentiate non-supplied macro parameters from those that have been supplied but which have been left blank. Many automatic behaviours are based upon the implicit passing of this value to parameters when the user does not supply a macro parameter, and it should be recognised that in Icarus, passing a blank parameter value is not the same as not passing a parameter at all.

SAS Variable and Macro Variable limitations

SAS contains relatively strict naming conventions and limitations. No doubt this is to ensure interoperability across many different systems, but it can make programming somewhat difficult and conservative.

The two limitations most relevant to Icarus are the maximum length of variable names, and the maximum size of macro variables.

The maximum length of a regular variable name in SAS is 32 characters, but to ensure stable operation of Icarus, I recommend variable names be short. The internal `_DJM` and `_ICARUS` prefixes can use some of these 32 characters, and if the user supplies additional prefixes or suffixes, the likelihood of inadvertently exceeding this limit in internal operations increases. I recommend not exceeding 20 characters in length for variable names. In simple terms, keep your variable names short.

The second limitation has to do with the maximum size SAS allows for macro variables.

As of version 9.3, which is the version Icarus requires for operation, this is set at 65,534 characters.

Allowing for example, phrases involving two variables 32 characters in length, connected with spaces and a couple of other characters, this should allow enough space for clauses involving more than 900 variable pairs. This is far more than are usually found in data linking exercises, but not unheard of in all operations. Smaller variable names will naturally result in more leeway before one reaches this limit.

Much of Icarus' basic components are fundamentally constituted of combinations of many macro variables.

The message to take away is that there is an upper limit to what Icarus and SAS can process.

This limit depends upon the specific operations and the size of the text being manipulated. I cannot guarantee when exactly you will hit this limit, but the example provided above should give you some indication as to the likely scale of the problems when errors may present themselves.

SAS system options

I recommend the SAS system options of memsize and realmemsize be set to a physical number for a SAS installation. The hashjoin macro uses the value of these settings and combines them with the dimensions available in the metadata of data sets to choose algorithms for combining data sets and variables.

Resources and Knowledge

Icarus likes to have it both ways: it tries to be extremely efficient, which means you can do far more than many other data linking packages, but it achieves performance by optimising all code where it can and allowing consumption of what resources are made available to it.

This includes heavy use of in-memory data structures, precompiled programs, and various other techniques where appropriate. This increased performance does not mean that you can entirely forget about resource concerns, as we all know that work tends to expand to fulfil the resources allocated to it. If you simultaneously load a database table into 10, 20, 50, 100, or 1000 hash objects, you will eventually hit a resources limit. No data linking program is able to deal with the largest data linking exercises today without some strategy of resource management.

A database hashed into memory takes up space you could be using to apply frequency based weights to your comparison space, or instead it could hold pre-prepared common string comparison values to speed up the calculations. In an ideal world, you should provide SAS with as much memory and hard drive space for the work library as you can comfortably allow, which is doubly recommended if you are performing large scale data linking and operations. Icarus uses the work library to store temporary disk based data, so the size of your work library is important.

I have tried to be responsible in my use of memory and the assumptions about the resources that SAS will have available to it. Again, in my experience, Icarus is actually much more efficient with its use of memory than many other programs, but I cannot foresee all possible permutations of problems and computer environments.

With regards to knowledge, the best use of Icarus is achieved when you fully understand the SAS programming environment. You can use Icarus in the most trivial sense with a limited knowledge of SAS by calling macros in a predefined order or using a template set up by a more experienced user. But it is capable of so much more...

If you understand how macro resolution works, step boundaries, macro nesting, and the difference between macro resolution/compile-time/run-time, then you can use Icarus to develop in minutes what would be physically impossible for many.

More so, your programs can be flexible, dynamic, and efficient. Your programs can change and rewrite themselves depending on the properties of the data and systems with which they are presented.

Keep Icarus together

Icarus is itself a system.

Small macros build medium macros which build big macros. They all then work together to write a program that will produce the results the user requests. Sometimes they do this in steps, only dynamically producing later code once earlier parts of processing have completed.

You can't separate the parts from the whole, and the entire project would have been practically impossible to write in SAS without designing it this way.

Icarus compiles and installs quickly, and is exceptionally small for the punch it packs, so the resources required by Icarus itself should not present any practical barriers.

Utility and Programming Macros

The macro functions and programs documented in this section are the simplest operations underlying Icarus.

They are primarily designed to increase the ease with which one partakes in meta-programming and automating common tasks and operations in SAS. They deal with manipulating strings of text and lists into common patterns. They obtain meta-data and system information to drive your programs. They also manage data sets and the SAS environment.

In this sense, most are not directly related to data linking any more than all tasks in SAS.

They underlie and enable the more complicated macros to exist. They can also make regular SAS programming more powerful and dynamic for anybody who wishes to use them.

Most of the macros in Icarus are parameter type macros, requiring parameters to be fed in via keywords. Because of their complexities and crossing of step boundaries, these larger macros are unable to be nested. However, because of their minimal nature, it is more appropriate for many of the utility macros to be written as functional macros with positional parameters instead.

Hence, the macro type is specified for each macro in this section.

Functional pure macros can be combined and nested within each other, as well as in regular SAS code. They can also be combined with the more complex parameterised macros to provide dynamic values for keyword parameters as well.

APPLYID

Function Style Macro

%applyid(dataset reference, ID variable name);

Description

Applyid adds a record ID variable to a data set. It changes the original data set and cannot be used with a view. The ID is a unique number for each observation, which also has the quality of being the position of the record within the data set. The ID variable will be a numeric variable.

Operation and Restrictions

Applyid implements a data step. It can only be used where an independent data step is valid.

Example: Add “ID_Var” to the data set “work.persons”

```
%applyid(work.persons, ID_Var);
```

work.persons	
Name	Age
Bob	8
Charlie	17
Rachel	25



work.persons		
ID_Var	Name	Age
1	Bob	8
2	Charlie	17
3	Rachel	25

COMMALIST

Function Style Macro

%commalist(list);

Description

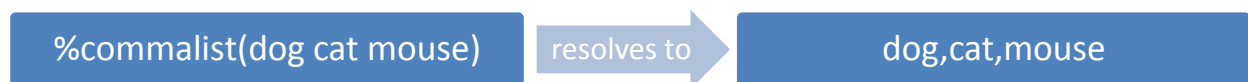
Commalist accepts a list of words separated by spaces.

Commalist resolves to the same list of words with a comma separating them instead of spaces.

Operation and Restrictions

Commalist is a pure macro. It can be used in any SAS code.

Example: Convert space delimited words to comma delimited words



COUNTWORDS

Function Style Macro

%countwords(list, delimiter);

Description

Countwords resolves to the number of words in a string of text. Words are delimited by the symbol supplied via delimiter. Countwords believes a word has been found if a character is found between consecutive delimiters, thus consecutive delimiters do not add to the word count.

Because of the nature of macros and SAS syntax, there is room for misunderstanding when supplying many common delimiters. A quoting function may be necessary.

Operation and Restrictions

Countwords is a pure macro. It can be used in any SAS code.

Examples: Resolve to the number of words in a text string

<code>%countwords(dog cat mouse,%STR());</code>	resolves to	3
<code>%countwords(%STR(dog,cat,mouse),%STR(,))</code>	resolves to	3
<code>%countwords(dog8cat8mouse,8)</code>	resolves to	3
<code>%countwords(dog8cat8mouse888,8)</code>	resolves to	3
<code>%countwords(dog8cat8mouse88ant8,8)</code>	resolves to	4

CREATESIMPLEDSET

Function Style Macro

%createsimpledset(name, variable list, variable list delimiter, value list, value list delimiter);

Description

Createsimpledset is a macro for creating a 1 x N data sets from textual parameters.

The first input is the name of the data set in standard SAS syntax: i.e. work.mydata. The second input must consist of a list delimited by some character. This will determine the variables on the data set. The third input defines the character that acts as a delimiter for the previous list. The fourth input defines the value of the variables within the data set, and must be a list with the same dimensions as the second input. The fifth input defines the character that acts as the delimiter for the second list.

The variable types on the resulting data set are determined depending on the values to be entered for each variable. If the user wishes to create non-numeric variables, each value must be surrounded by quotes, or else sas will interpret the values as variable references within the data set itself. Input must also follow SAS rules for its use (i.e. variable names must constitute valid SAS variable names).

Operation and Restrictions

Createsimpledset implements a data step. It can only be used where an independent data step is valid.

Example: Create a simple data set

```
%createsimpledset(work.mydata,dog cat horse,%str( ),1 2 3,%str( ))
```



work.mydata		
dog	cat	horse
1	2	3

DELETEDSETS

Function Style Macro

%deletedsets(dataset1 dataset2 ... datasetN)

Description

Deletedsets is responsible for deleting data sets and SAS views.

It accepts a list of data sets/views separated by spaces, and deletes each of them.

Data sets and views may both be intermixed within the same call.

If a data set or view does not contain a library reference, deletedsets assumes it is contained within the work library.

If a data set or view cannot be found at its specified location, deletedsets writes a note to the log. It continues to delete the data sets or views that can be found.

Note that the data sets/views to be deleted are the only argument to be fed to the macro.

There are no commas separating each data set/view, only spaces.

Operation and Restrictions

Deletedsets implements its own SQL commands. It can be used only where independent calls to PROC SQL would be valid.

Example: Attempt to delete the data set called work.mydata, the SAS view called work.myview, and the data set called mylib.myotherdata

```
%deletedsets(work.mydata work.myview mylib.myotherdata);
```

DELETPROGRAMS

Function Style Macro

%deleteprograms(compiledprogram1 compiledprogram2 ... compiledprogramN)

Description

Deleteprograms is an administrative macro responsible for deleting compiled SAS programs.

It accepts a list of compiled data step programs separated by spaces and deletes each program referenced.

If a program does not contain a library reference, deleteprograms assumes it is contained within the work library.

If a program cannot be found at its specified location, deleteprograms writes a note to the log. It continues to delete programs that can be found.

Note that the programs to be deleted are the only argument to be fed to the macro.

There are no commas separating each program, only spaces.

Operation and Restrictions

Deleteprograms implements its own datasets procedure. It can be used only where PROC DATASETS would be valid.

Example: Attempt to delete the compiled data step programs called work.program1 and work.program2

```
%deleteprograms(work.program1 work.program2);
```

DSETPARSE

Function Style Macro

%dsetparse(data set reference)

Description

Dsetparse accepts a SAS data set reference as an argument, of the form library.datasetname.

It resolves to the data set name with the library reference removed.

If you feed dsetparse a data set name that does not have a library reference, dsetparse will merely return the data set name.

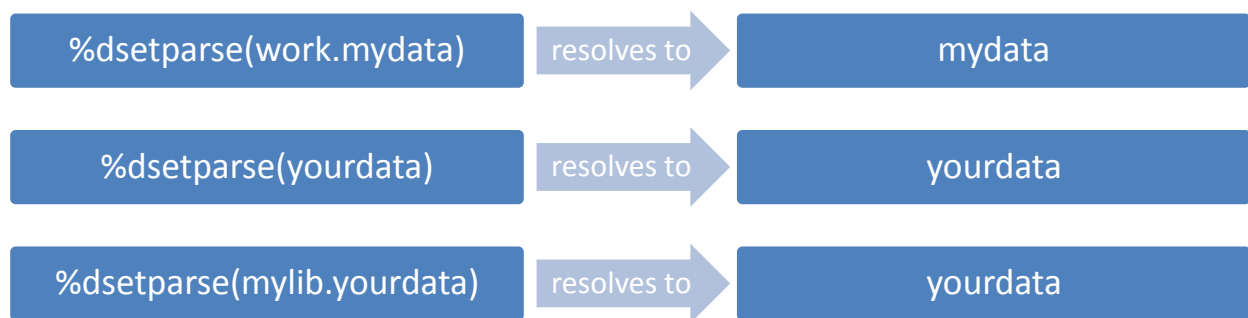
Dsetparse does not check that the data set name is in fact a valid SAS name, or that there is only one name, or that the data set exists.

It seeks for the first period in a string, and resolves to those characters following the first period. If a period is not found, it returns the original string.

Operation and Restrictions

Dsetparse is a pure macro. It can be used in any SAS code.

Examples: Return the name of a data set when supplied a reference



DSETVALIDATE

Function Style Macro

%dsetvalidate(dataset reference)

Description

Dsetvalidate is a macro used to confirm the existence of a data set or view.

It resolves to 1 if the data set exists, and to 0 if it does not.

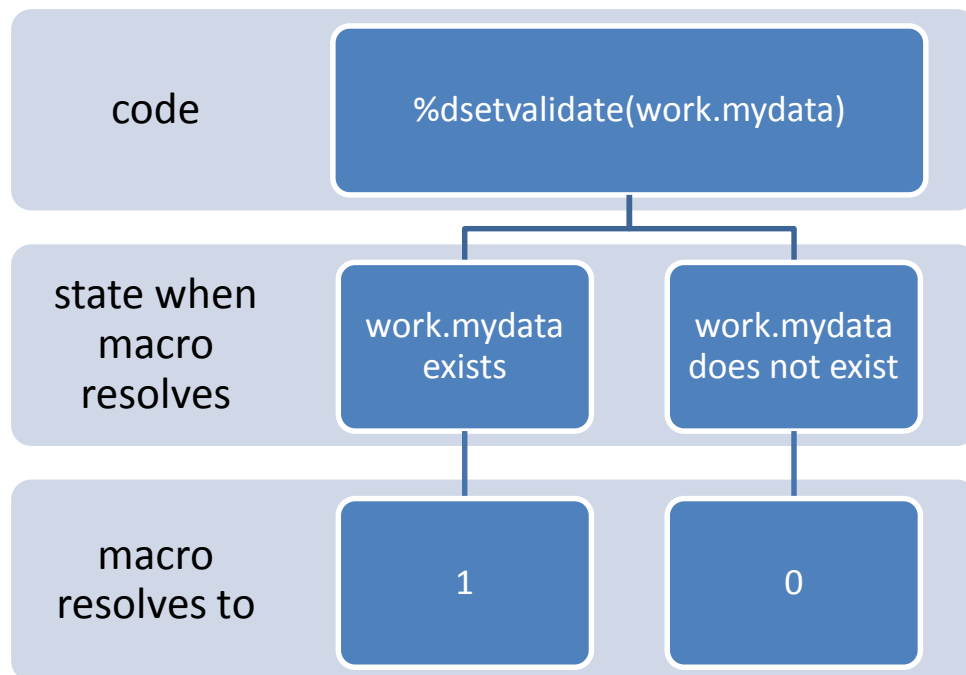
If there is no library reference supplied in the data set reference, dsetvalidate assumes the data set being sought is in the work library.

Operation and Restrictions

Dsetvalidate is a pure macro. It can be called in any SAS program.

Dsetvalidate is most useful combined with conditional macro statements. It allows code to be produced conditional upon whether particular data sets exist.

Example: Does a data set named work.mydata exist?



DSETVRENAME

Function Style Macro

%dsetvrename(dataset reference, variable list1, variable list2)

Description

Dsetvrename allows you to rename variables in a data set by supplying a data set reference, a list of variables to be renamed, and a list specifying how you would like them named instead.

All arguments to dsetvrename must comply with the requisite SAS rules. If a variable is being renamed, it must in fact exist on the SAS data set that is being referenced.

The two variable lists must be of the same dimension. Each word in the first list will be renamed to the word which shares the same position in the second list.


Each list must be delimited with spaces.

Operation and Restrictions

Dsetvrename implements its own call to the datasets procedure. It can be used only where calls to PROC DATASETS would be valid.

Example: Rename the variables “ID_Var” and “Age” to “Preference” and “Score”

```
%dsetvrename(work.persons, ID_Var Age, Preference Score);
```



work.persons		
ID_Var	Name	Age
1	Bob	8
2	Charlie	17
3	Rachel	25

work.persons		
Preference	Name	Score
1	Bob	8
2	Charlie	17
3	Rachel	25

ESTIMATESIZE

Function Style Macro

%estimatesize(dataset reference, list of variables)

Description

Estimatesize estimates the number of bytes used for storage of the listed variables in the reference data set. Mathematically, this equates to:

$$\sum length(var_i) * number of observations in data set$$

The function can be used on SAS views, but there is no way to know the number of observations in a view before it is accessed. If the view is particularly large, iterating through the view could take the macro a prohibitive amount of time. SAS data sets are more efficient at this calculation, as the requisite information is already contained in the metadata of the data set. The list of variables must be delimited by spaces and all must be in the data set.

Operation and Restrictions

Estimatesize is a pure macro. It can be called in any SAS program.

Example: Estimate the amount of memory needed to store the variables Name and Age from the data set work.persons

work.persons		
ID_Var	Name	Age
1	Bob	8
2	Charlie	17
3	Rachel	25

Because it is a numeric variable, the length of ID_Var and Age is 8. Name has a length of 7.

%estimatesize(work.persons, ID_Var Name)

resolves to

45

FILESINDIR

Parameterised Macro

```
%filesindir(  
  
Dir = ,  
  
Ext = ,  
  
Extension = ,  
  
Datasetflag = ,  
  
Dataset = ,  
  
PathVar = ,  
  
FileVar = ,  
  
ExtVar = ,  
  
Delimiter =  
  
);
```

Description

Filesindir is relatively complex for a utility macro.

It provides a list of files in a given directory. It can do this by either resolving to a textual list in a pure macro form, or it can implement its own data step and output results in a dataset for further processing.

By default, the user only needs to supply the **dir** argument. This is the directory from which you wish to receive the list of files. By default, the type of file you will receive is the list of all sas data sets within the directory location specified as a space delimited textual list.

Parameters

Dir: The system path to the directory from which you wish to obtain a list of files.

Ext: A flag indicating whether to target files with a specific extension. Set to either Y or N. By default, this parameter is set to Y and the macro targets SAS data set extensions.

Extension: If the ext flag is set to Y, then this parameter tells the macro what file extension you wish to target. You can supply the parameter if the ext flag is set to N, but it will not do anything. By default, it is targeting SAS data sets, and has the value of sas7bdat.

Delimiter: A variable that can be changed to determine how the list of files is delimited by the pure macro form. By default, its value is %STR().

Datasetflag: This flag tells the macro to drop out of pure macro form and author a SAS data set. It can be set to Y or N. By default, it is set to N. The data set output will contain information on the files targeted (which is defined by the previous parameters).

The data set contains three character variables. They contain the directory path, the file name, and the file extension. Lengths and dimensions for these variables are set by the macro to be as small as possible while retaining the requisite information. The name of the data set, and of the variables, can be set with the following parameters:

Dataset: The data set reference of the data set that the macro will output.

Pathvar: The name of the variable that holds the directory path information.

Filevar: The name of the variable that holds the file name.

Extvar: The name of the variable that holds the file extensions.

Operation and Restrictions

If the macro is in pure macro form, it can be called in any SAS code.

If the macro is authoring a data set, it can only be called in places where a SAS data step is valid.

The ability of the macro to change its behaviour can be confusing, and I recommend the user choose one application of the macro in any context and stick with it.

For the examples that follow, assume a directory defined by the following location:

```
\\network\SASfiles\icarusfiles\
```


The hypothetical directory used in this example contains the following files:

Alpha.sas
awesomefunctions.sas7bdat
awesomefunctions.sas7bndx
filea.sas7bdat
fileb.sas7bdat
Omega.sas
Rubicon.txt
Wuwei.sas

Example 1: In pure macro form, obtain a list of the sas data sets

```
%filesindir(  
dir=\\network\SASfiles\icarusfiles\  
)
```

resolves to

```
awesomefunctions.sas7bdat  
filea.sas7bdat fileb.sas7bdat
```

Example 2: In pure macro form, obtain a list of sas code files, delimited by a comma

```
%filesindir(  
dir = \\network\SASfiles\icarusfiles\  
extension = sas,  
delimiter = %STR(,)  
)
```

resolves to

```
Alpha.sas, Omega.sas,  
Wuwei.sas
```

Example 3: In pure macro form, obtain a list of all files

```
%filesindir(  
dir = \\network\SASfiles\icarusfiles\  
ext = N  
)
```

resolves to

```
Alpha.sas  
awesomefunctions.sas7bdat  
awesomefunctions.sas7bndx  
filea.sas7bdat fileb.sas7bdat  
Omega.sas Rubicon.txt  
Wuwei.sas
```

Example 4: In a data step, place the details of all files into a data set called work.details

```
%filesindir(  
dir = \\network\SASfiles\icarusfiles\  
ext = N,  
datasetflag = Y,  
dataset = work.details  
)
```

work.details		
Path	File	Extension
\\network\SASfiles\icarusfiles\	Alpha	sas
\\network\SASfiles\icarusfiles\	awesomefunctions	sas7bdat
\\network\SASfiles\icarusfiles\	awesomefunctions	sas7bndx
\\network\SASfiles\icarusfiles\	filea	sas7bdat
\\network\SASfiles\icarusfiles\	fileb	sas7bdat
\\network\SASfiles\icarusfiles\	Omega	sas
\\network\SASfiles\icarusfiles\	Rubicon	txt
\\network\SASfiles\icarusfiles\	Wuwei	sas

Example 5: In a data step, place the details of all files into a data set called work.details and name the variables mypath, myfilename, and myextensions respectively

```
%filesindir(  
  dir = \\network\SASfiles\icarusfiles\  
  ext = N,  
  datasetflag = Y,  
  dataset = work.details,  
  pathvar = mypath,  
  filevar = myfilename,  
  extvar = myextensions  
)
```



work.details		
mypath	myfilename	myextensions
\\network\SASfiles\icarusfiles\	Alpha	sas
\\network\SASfiles\icarusfiles\	awesomefunctions	sas7bdat
\\network\SASfiles\icarusfiles\	awesomefunctions	sas7bndx
\\network\SASfiles\icarusfiles\	filea	sas7bdat
\\network\SASfiles\icarusfiles\	fileb	sas7bdat
\\network\SASfiles\icarusfiles\	Omega	sas
\\network\SASfiles\icarusfiles\	Rubicon	txt
\\network\SASfiles\icarusfiles\	Wuwei	sas

FINDREPLACE

Function Style Macro

%findreplace(string, find, replace)

Description

Findreplace is used to implement the SAS prxchange function on a string of text. The first argument is the text on which the operation is to take place. The second and third arguments are the parts of the PERL regular expression representing the pattern and the value which is substituted when found.

In SAS macro syntax, the macro implements the function thusly:

```
Prxchange(s/&find./&replace./,-1,&string.)
```

For common find and replace operations on regular English Latin alphabet text, no real knowledge of PERL regular expressions is necessary, as the naïve operation will often generate the desired result. A full exposition on PERL Regular Expressions is not going to take place in this manual, but there remains plenty of information available online.

Operation and Restrictions

Findreplace is a pure macro. It can be called in any SAS code.

Example 1: Find the word cat in a string and replacing it with the word dog

```
%findreplace(I took the cat  
for a walk, cat, dog)
```

resolves to

```
I took the dog for a  
walk
```

Example 2: Find the word dog in a string and replacing it with the word person

```
%findreplace(dog1 dog2  
dog3, dog, person)
```

resolves to

```
person1 person2  
person3
```

GETOPTION

Function Style Macro

%getoption(SAS system option)

Description

Getoption resolves to the value of the SAS system option it receives as a textual parameter

Operation and Restrictions

Getoption is a pure macro. It can be used in any SAS code.

Example 1: Return the CPUCOUNT on a typical quad-core installation



INTERLEAVE

Function Style Macro

%interleave(list1, list2)

Description

Interleave is used to combine two space delimited lists of equal dimension.

Operation and Restrictions

Interleave is a pure macro. It can be used in any SAS code.

Example 1: Interleaving two lists delimited by spaces



KEEPWORDPATTERN

Function Style Macro

`%keepwordpattern(list, regex)`

Description

Keepwordpattern takes a list of space delimited words, and matches them against a pattern defined by a PERL regular expression. Those words that match the pattern remain in the list, while those which do not match the pattern are removed.

The function uses the SAS function prxparse, which parses the Perl Regular Expression provided. Depending on the specifics of the expression and its conformity to SAS macro language and parsing, quoting functions may need used to ensure it is passed to the function properly.

Operation and Restrictions

%keepwordpattern is a pure macro. It can be called in any SAS program.

Example: Use a Perl Regular Expression to keep words from a space delimited list



LENGTHFIXER

Parameterised Macro

```
%lengthfixer(  
  
DataSet = ,  
  
Align = N  
  
);
```

Description

Lengthfixer is a useful performance macro. It finds the minimal length attributable to character variables in a data set, under the assumption that leading and trailing blanks are removable.

It overwrites the data set with a new one of the same name, where the length of character variables have been set to their minimum possible value. The macro leaves the length of all numeric variables with their original value. The benefit from performing this optimisation is less disk space and memory required to store data, and real time gains from quicker input/output operations.

Parameters

Dataset: The reference to the data set that you wish to optimise.

Align: If this parameter is set to Y, then the lengths of character variables will be the smallest number perfectly divisible by 8. It does not need to be supplied, and the default value is N.

Operation and Restrictions

Lengthfixer implements its own data step. It can be used where ever a data step is valid.

Example: Optimise the variables on the data set work.bigdata

```
%lengthfixer(Dataset = work.bigdata);
```

LIBNAMEPARSE

Function Style Macro

%libnameparse(data set reference)

Description

Libnameparse accepts a data set reference of the form library.datasetname

It resolves to the name of the library with the data set name removed.

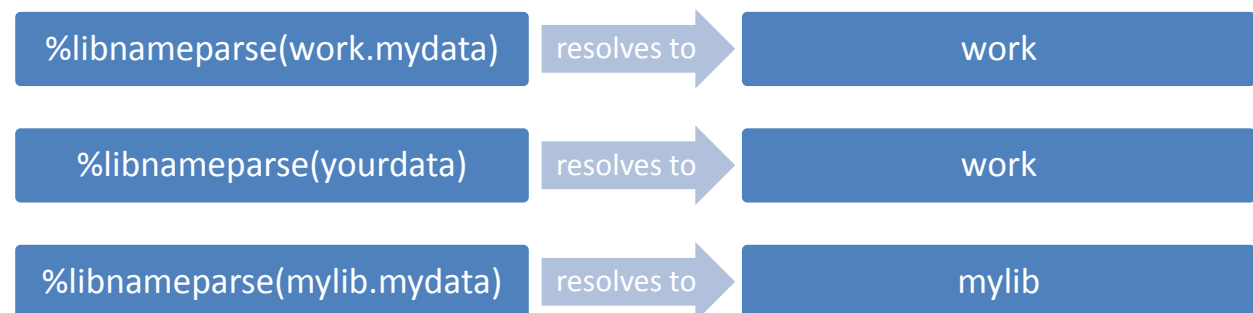
If you feed libnameparse a data set that does not have a library reference, libnameparse will resolve to WORK.

Libnameparse does not check that the data set name is in fact a valid SAS name. It merely seeks for the first period in a string, and resolves to those characters before the first period. If a period is not found, it resolves to the string WORK.

Operation and Restrictions

Libnameparse is a pure macro. It can be used in any SAS code.

Examples: Return the name of a data set when supplied a reference



LIBVALIDATE

Function Style Macro

%libvalidate(libname)

Description

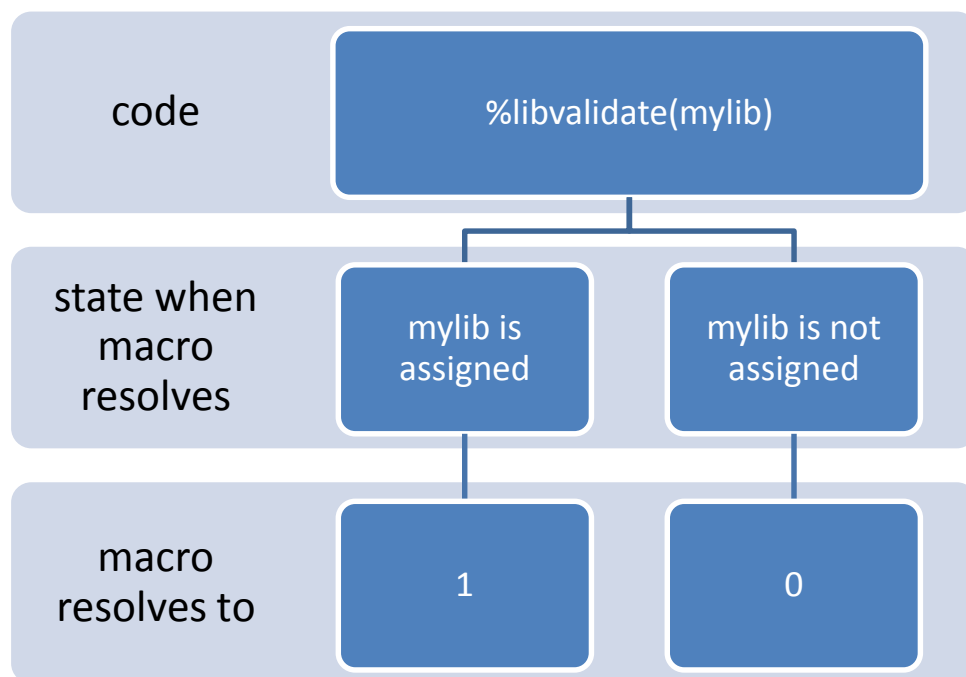
Libvalidate is used to confirm that a particular library reference exists.

It resolves to 1 if the libname exists, and resolves to 0 if it does not.

Operation and Restrictions

Libvalidate is a pure macro. It can be called in any SAS program.

Example: Does a library called mylib exist?



MEMSIZE

Simple Macro

%memsize

Description

Memsize resolves to the setting of the memsize SAS system option.

Memsize designates the limit on the total amount of virtual memory that can be used by a SAS session. Full discussion of the memsize option is available online in the SAS system documentation.

Operation and Restrictions

Memsize is a pure macro. It can be used in any SAS code. Ideally, Memsize has been set to an actual value, but there is the possibility that it has been set to the value of “MAX” or “0”. I have not been able to test %memsize under these conditions.

Example 1: Return the memsize option which is set to 2 Gigabytes



NUMOFOBS

Function Style Macro

%numofobs(dataset reference)

Description

Numofobs resolves to the number of observations in a data set. It can be used on both data sets and SAS views, but for performance reasons it is advisable to try to preference its use to data sets.

An earlier version and discussion on the issues of such a macro was written by Jack Hamilton in January 2001, which is suggested reading for anyone interested. Both that version and the Icarus version are essentially the same in operation, and only differ in internal representation and operation.

Operation and Restrictions

Numofobs is a pure macro. It can be used in any SAS code.

Example: Get the number of observations in the data set called `work.persons`

work.persons		
ID_Var	Name	Age
1	Bob	8
2	Charlie	17
3	Rachel	25

`%numofobs(work.persons)`

resolves to

3

NUMORDSET

Function Style Macro

%numordset(text)

Description

Numordset betrays its original purpose in its name, rather than its actual mechanism of operation. Because data sets in SAS are not allowed to begin with a number, numordset was created to test whether input was a number or a data set.

In all actuality, its mechanical operation is to check the first character in a string, and determine whether that character is a digit in the range of 0-9, or whether it is not.

It resolves to N if the first character in the string is a decimal digit and resolves to D if it is not.

Operation and Restrictions

Numordset is a pure macro. It can be called anywhere in SAS code.

Examples:

<code>%numordset(8347)</code>	resolves to	N
<code>%numordset(work.person)</code>	resolves to	D
<code>%numordset(8djwi58k)</code>	resolves to	N
<code>%numordset(number93874)</code>	resolves to	D
<code>%numordset(eight)</code>	resolves to	D

NUMSTOCHARS

Function Style Macro

%numstochars(dataset reference, variables, significant digits)

Description

Numstochars is a slightly unusual macro. It was designed to deal with the situation whereby a data set has numeric variables that we really wish could be treated as character variables.

Numstochars is a pure macro, so it does not drop into a data step or SQL and edit the data sets.

Instead, it looks in the data set for the variables that have been fed in as a space delimited list. It will cycle through each of the variables, and check their types.

If a variable is of type character, then it will be output from the original list unchanged.

However, if a variable is of type numeric, it wraps the variable in a put function, and the format applied in the put function is “BEST&significantdigits..”.

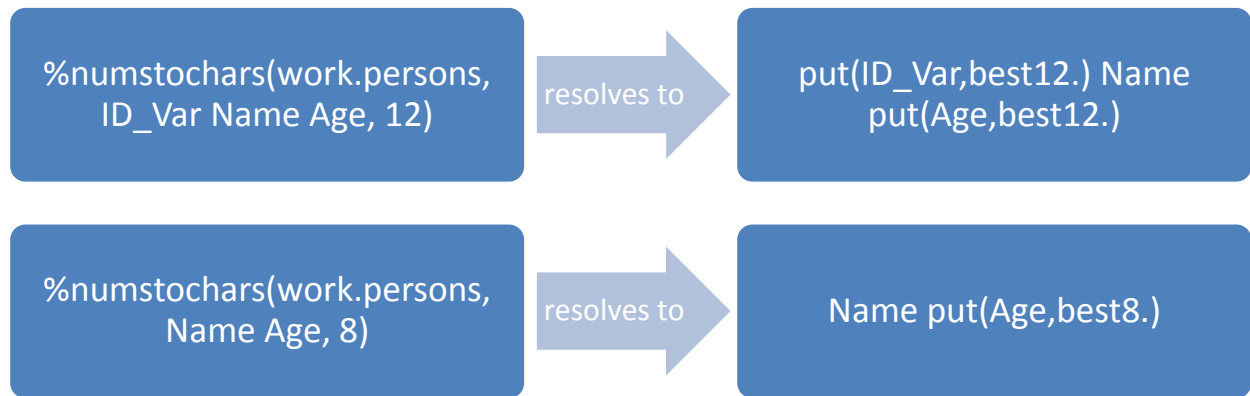
Some examples will be helpful in demonstrating the use of numstochars.

Operation and Restrictions

Numstochars is a pure macro. It can be called anywhere in SAS code. Whether it makes sense to call it in any SAS code is a much more context sensitive question relative to other macros however, and it likely has to be combined with other macros or code to be of any practical use.

Examples: Calling numstochars on the data set work.persons. The variables ID_Var and Age are of type numeric. Name is a variable of type character.

work.persons		
ID_Var	Name	Age
1	Bob	8
2	Charlie	17
3	Rachel	25



OBTOMACRO

Function Style Macro

%obtomacro (dataset reference, variable list, observation number)

Description

Obtomacro is designed as a macro function to pull observations from data sets into the scope of the macro language, where it can either be manipulated, stored, or used as text.

The dataset reference determines which data set the observation will be extracted from. The variable list, delimited by spaces, determines the variables within the data set that will have their values extracted. The observation number determines which observation holds the values that will be extracted.

Once extracted, the macro resolves to a list of space delimited text. The words within this list will be the values from the respective data set, variables, and observation.

Note: The user may have some practical problems if their data set already contains blank or missing data in the variables they try to import into the macro.

Operation and Restrictions

Obtomacro is a pure macro. It can be called anywhere in SAS.

Example: Extract the second observation from work.persons

work.persons		
ID_Var	Name	Age
1	Bob	8
2	Charlie	17
3	Rachel	25

%obtomacro(work.persons,
ID_Var Name Age, 2)

resolves to

2 Charlie 17

PL

Function Style Macro

%pl(list, prefix)

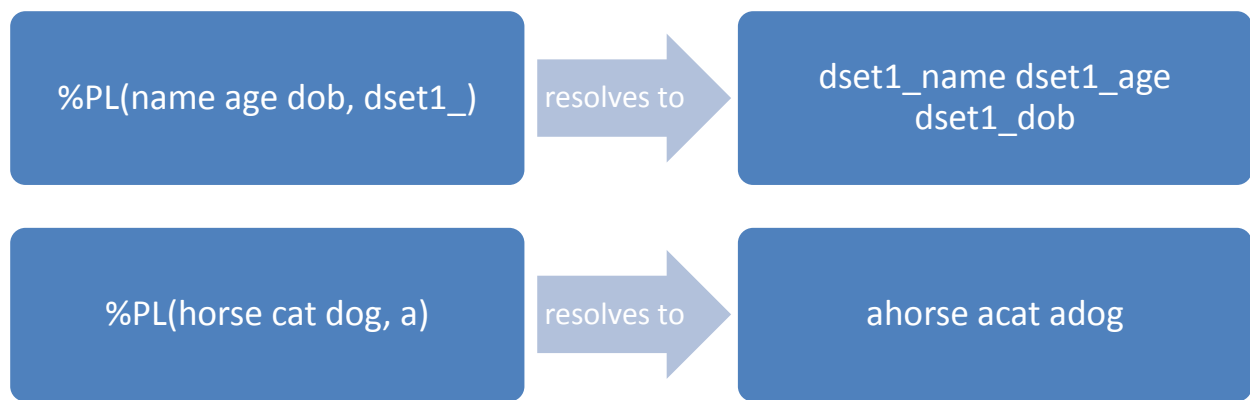
Description

PL stands for prefix list. PL is responsible for adding a prefix to a list of space delimited words, and resolves to the original list with the prefix appended to each word.

Operation and Restrictions

PL is a pure macro. It can be used anywhere in SAS code.

Example: Add a prefix to a list of words



PROGVALIDATE

Function Style Macro

%progvalidate(compiled data step program)

Description

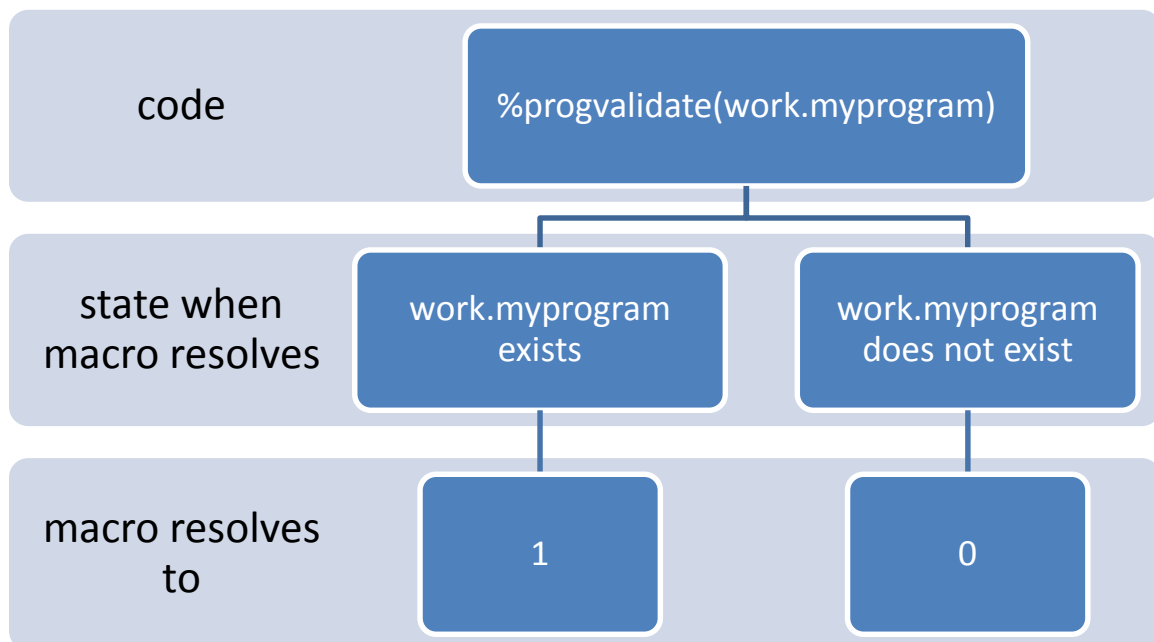
Progvalidate is a macro used to confirm that a particular compiled data step program exists.

It resolves to 1 if the program exists, and 0 if it does not.

Operation and Restrictions

Progvalidate is a pure macro. It can be called in any SAS program.

Example: Does a compiled data step program called myprogram exist in the work library?



QCLIST

Function Style Macro

%qclist(list);

Description

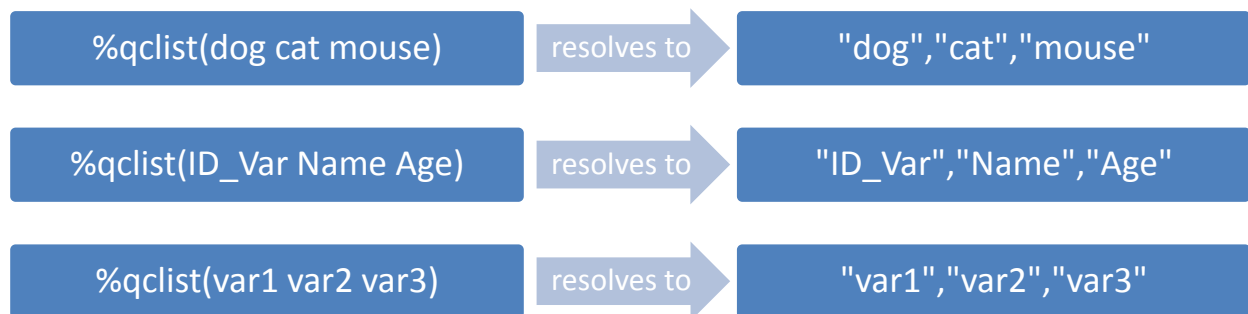
Qclist accepts a string of text consisting of words separated by spaces.

Qclist resolves to the same list of words, with a comma separating words instead of spaces, and double quotes surrounding each word.

Operation and Restrictions

Qclist is a pure macro. It can be used in any SAS code.

Examples: Convert space delimited words to comma delimited words, with each word surrounded in double quotes



QUOTELIST

Function Style Macro

%quotelist(list);

Description

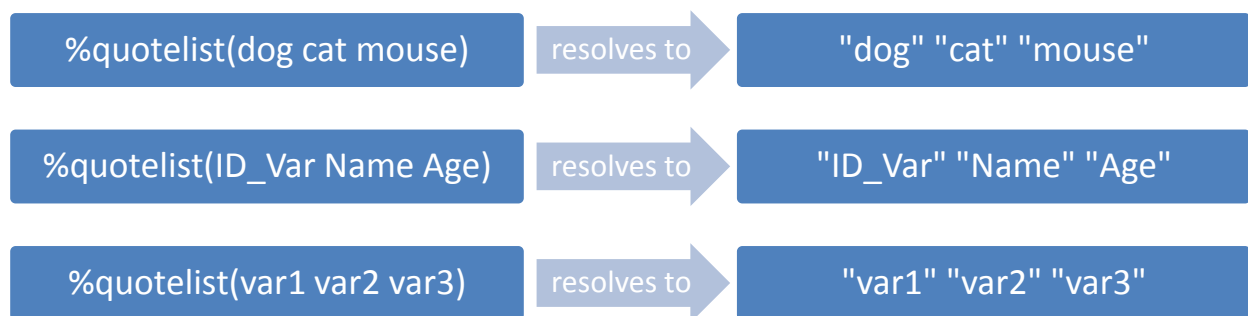
Quotelist accepts a string of text consisting of words separated by spaces.

Quotelist resolves to the same list of words, with double quotes surrounding each word.

Operation and Restrictions

Quotelist is a pure macro. It can be used in any SAS code.

Examples: Wrap space delimited words in double quotes



REALMEM

Simple Macro

`%realmem`

Description

Realmem resolves to the amount of real memory the SAS session has available to it at the time of resolution.

Operation and Restrictions

Realmem is a pure macro. It can be used in any SAS code. It uses the undocumented diagnostic option of 'xmrlmem' in SAS.

Example 1: A SAS session which has 2 Gigabytes of RAM available



REMOVELIBRARIES

Function Style Macro

```
%removelibraries(list);
```

Description

Removelibraries deassigns each library reference fed in via a list of words delimited by spaces.

Operation and Restrictions

Removelibraries initiates libname statements. It is valid where the libname statement is valid.

Example: Deassign the library references “place” and “stuff”

```
%removelibraries(place stuff);
```

REMOVEWORDFROMLIST

Function Style Macro

```
%removewordfromlist(word, list);
```

Description

Removewordfromlist accepts a word, and a list of words delimited by a space.

Removewordfromlist resolves to the list with the offending word removed.

Operation and Restrictions

Removewordfromlist is a pure macro. It can be used in any SAS program.

Example: Remove a word from a list

```
%removewordfromlist(cat, cat  
dog horse)
```

resolves to

```
dog horse
```

REPEATER

Function Style Macro

%repeater(word, N, delimiter);

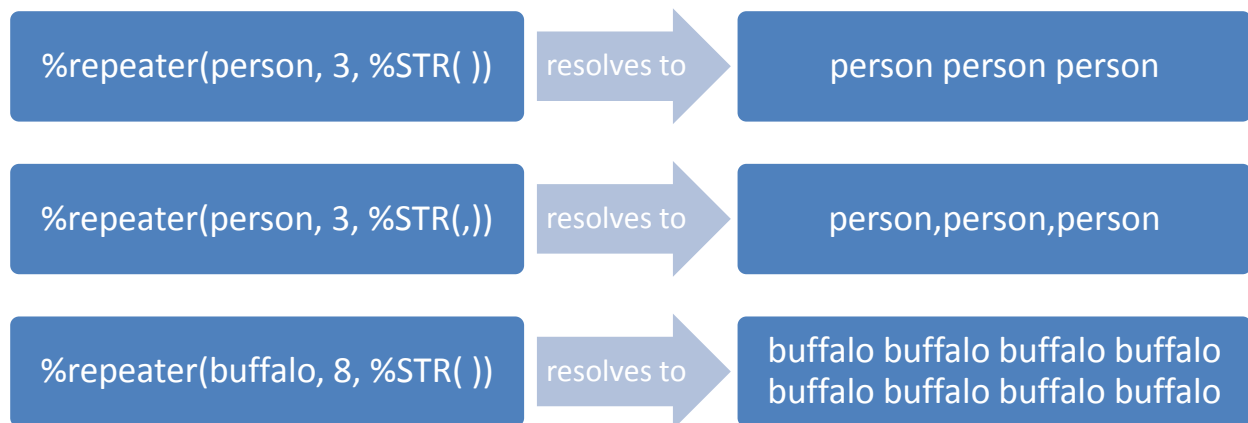
Description

Repeater takes a word, and resolves to a list consisting of that word repeated N times, separated by the symbol supplied to delimiter.

Operation and Restrictions

Repeater is a pure macro. It can be called in any SAS code.

Example: Repeat a string of text



REPEATERANDNUM

Function Style Macro

`%repeaterandnum(word, N, delimiter);`

Description

Repeaterandnum takes a word, and returns that word repeated N times separated by the delimiter. Repeaterandnum appends the numbers 1 through N to the end of each word.

Operation and Restrictions

Repeaterandnum is a pure macro. It can be called in any SAS code.

Example: Repeat a word and append a number to the end of each word



REPORT_DATE

Simple Macro

`%report_date`

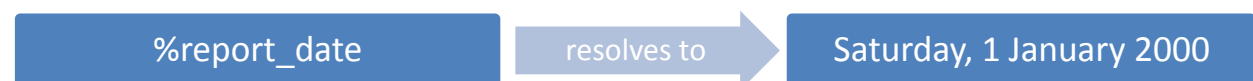
Description

Report_date resolves to the current date.

Operation and Restrictions

Report_date is a pure macro. It can be called in any SAS program.

Example: Running %report_date on the 1st of January, 2000



REPORT_DATETIME

Simple Macro

%report_datetime

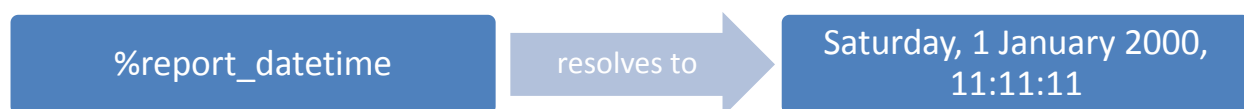
Description

Report_datetime resolves to the current date and time.

Operation and Restrictions

Report_datetime is a pure macro. It can be called in any SAS program.

Example: Running %report_date time on the 1st of January, 2000, eleven minutes and eleven seconds after eleven o'clock in the morning



REPORT_TIME

Simple Macro

%report_time

Description

Report_time resolves to a 24 hour representation of the current time.

Operation and Restrictions

Report_time is a pure macro. It can be called in any SAS program.

Example: Running %report_time at 11:30 in the morning



RPL

Function Style Macro

%rpl(list, prefix)

Description

RPL stands for remove prefix list. RPL is responsible for removing a prefix from a list of space delimited words.

RPL is a “dumb” macro. In operation it does not in fact check for equality of the prefix on the front of words fed to it.

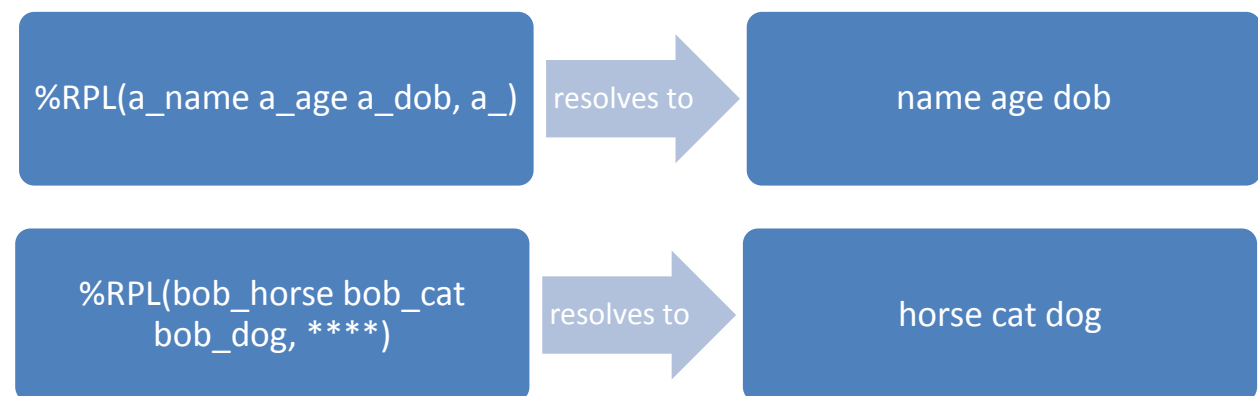
Instead, it obtains the length of the prefix, and removes the same number of characters from the front of each word in the list fed to it.

Hence, the prefix cannot be longer than any of the words in question.

Operation and Restrictions

RPL is a pure macro. It can be used anywhere in SAS code.

Example: Remove prefixes from a list of words



RSL

Function Style Macro

%rsl(list, suffix)

Description

RSL stands for remove suffix list. RSL is responsible for removing a suffix from a list of space delimited words.

RSL is a “dumb” macro. In operation it does not in fact check for equality of the suffix on the end of words fed to it.

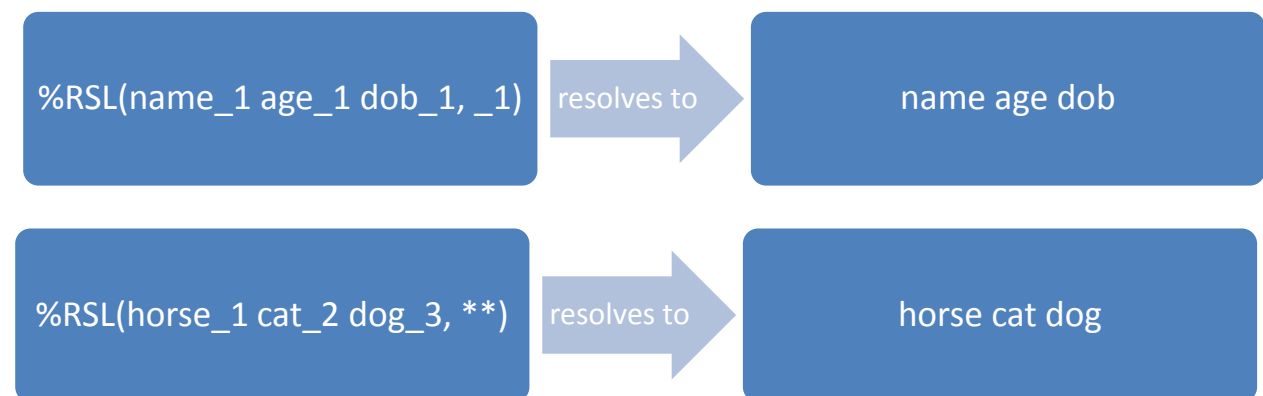
Instead, it obtains the length of the suffix fed to it, and removes the same number of characters from the end of each word.

The suffix cannot be longer than any of the words in question.

Operation and Restrictions

RSL is a pure macro. It can be used anywhere in SAS code.

Example: Remove suffixes from a list of words



SL

Function Style Macro

`%sl(list, suffix)`

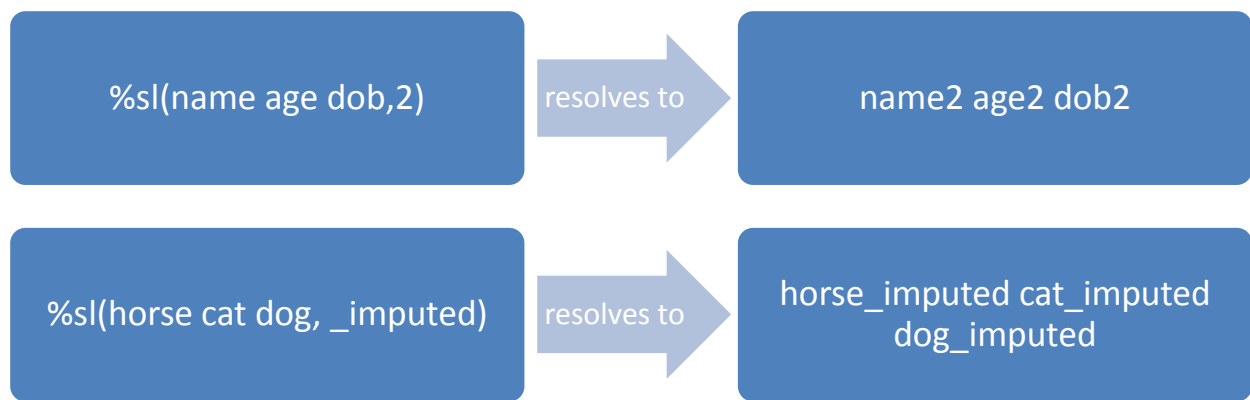
Description

SL stands for suffix list. SL is responsible for adding a suffix to a list of space delimited words, and resolves to the original list with the suffix appended to each word.

Operation and Restrictions

SL is a pure macro. It can be used anywhere in SAS code.

Example: Add suffixes to a list of words



TERMLISTPATTERN

Function Style Macro

%termlistpattern(list, word, delimiter1, delimiter2)

Description

Termlistpattern is a macro designed to rearrange common text into a pattern of syntax commonly found in SAS programming. The list must be space delimited.

List **L** is **N** words long, with each word being demarcated **L₁, L₂, ... , L_N**

The singular word is called **W**

Delimiter 1 is **D₁**

Delimiter 2 is **D₂**

Termlistpattern resolves to the following general pattern:

L₁ D₁ W D₂ L₂ D₁ W D₂ L₃ D₁ W D₂ ... L_N D₁ W

Note that on the final repetition of the pattern, whenever that may be, **D₂** is absent.

Operation and Restrictions

Termlistpattern is a pure macro. It can be called in any SAS code

Example: Create code to check consecutive flag variables

```
%termlistpattern(dogflag  
catflag mouseflag, 1, %STR(=  
, %STR( AND ))
```

resolves to

```
dogflag = 1 AND catflag = 1  
AND mouseflag = 1
```

TVTDL

Function Style Macro

%tvtdl(list A, list B, delimiter1, delimiter2)

Description

Tvtdl produces one of the more common patterns of text found when programming. Both list arguments must be space delimited. List A and List B must be of equal dimensions.

List **A** is **N** words long, with each word being demarcated **A₁, A₂, ... , A_N**

List **B** is **N** words long, with each word being demarcated **B₁, B₂, ... , B_N**

Delimiter 1 is **D₁**

Delimiter 2 is **D₂**

Tvtdl resolves to the following general pattern:

A₁ D₁ B₁ D₂ A₂ D₁ B₂ D₂ A₃ D₁ B₃ D₂ ... A_N D₁ B_N

Note that on the final repetition of the pattern, whenever that may be, **D₂** is absent.

Operation and Restrictions

Tvtdl is a pure macro. It can be called in any SAS code.

Example: Create code representing a simple composite AND clause

```
%tvtdl(dog1 cat1 mouse1,  
dog2 cat2 mouse2 , %STR( = ),  
%STR( AND ))
```

resolves to

```
dog1 = dog2 AND cat1 = cat2  
AND mouse1 = mouse2
```

UNIQUEWORDS

Function Style Macro

%uniquewords(list)

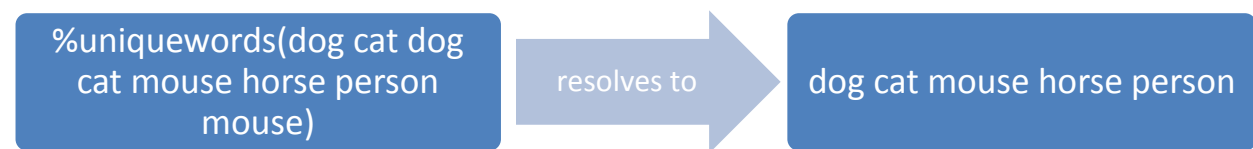
Description

Uniquewords takes a list of space delimited words as an argument. It returns a list of space delimited words, with any duplicate words removed.

Operation and Restrictions

Uniquewords is a pure macro. It can be called in any SAS code.

Example: Return a list of unique words



VARKEEPLIST

Function Style Macro

%varkeepelist(list, binary list)

Description

Varkeepelist accepts two space delimited lists as arguments.

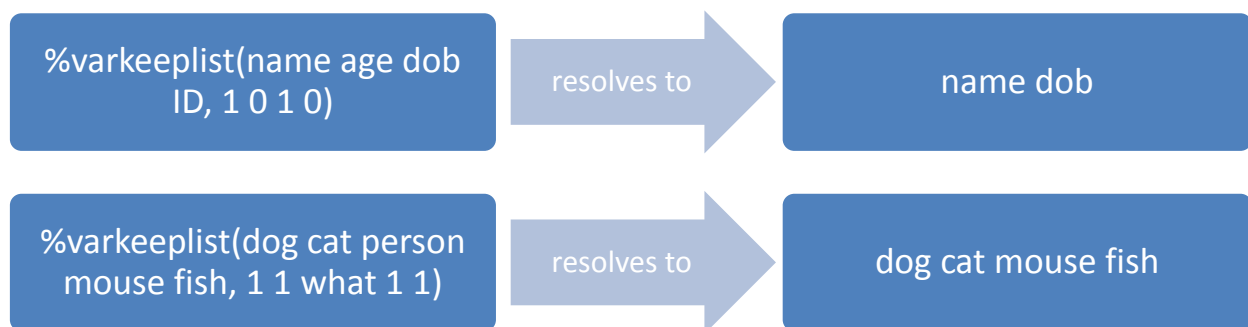
The first list consists of a series of words. Varkeepelist uses the values in the second list, which ideally is a string of space delimited 1s and 0s (but which can technically be any list of space delimited words) of equal dimension to the first list, to keep a subset of the words in the first list.

The words kept from the first list are those that share the same relative position as the 1s that are found in the second list.

Operation and Restrictions

Varkeepelist is a pure macro. It can be used in any SAS code.

Example:



VARKEEPLISTDSET

Function Style Macro

%varkeeplistdset(data set reference, observation number)

Description

Varkeeplistdset is a variation on the macro *varkeeplist*.

It is hence best contrasted against the *varkeeplist* macro.

Varkeeplist accepts two space delimited lists as arguments.

The first list consists of a series of words. *Varkeeplist* uses the values in the second list, which ideally is a string of space delimited 1s and 0s (but which can be technically any list of space delimited words) of equal dimension to the first list, to keep a subset of the words in the first list. The words kept from the first list are those that share the same relative position as the 1s found in the second list.

Varkeeplistdset however, asks for a dataset and an observation number instead of two space delimited lists of equal dimension. It performs a similar action, but does so by sourcing the first and second list from the data set and observation number fed to it.

The first list is obtained by accessing the list of variable names in the data set referenced. The second list is obtained by accessing the observation number in the data set referenced.

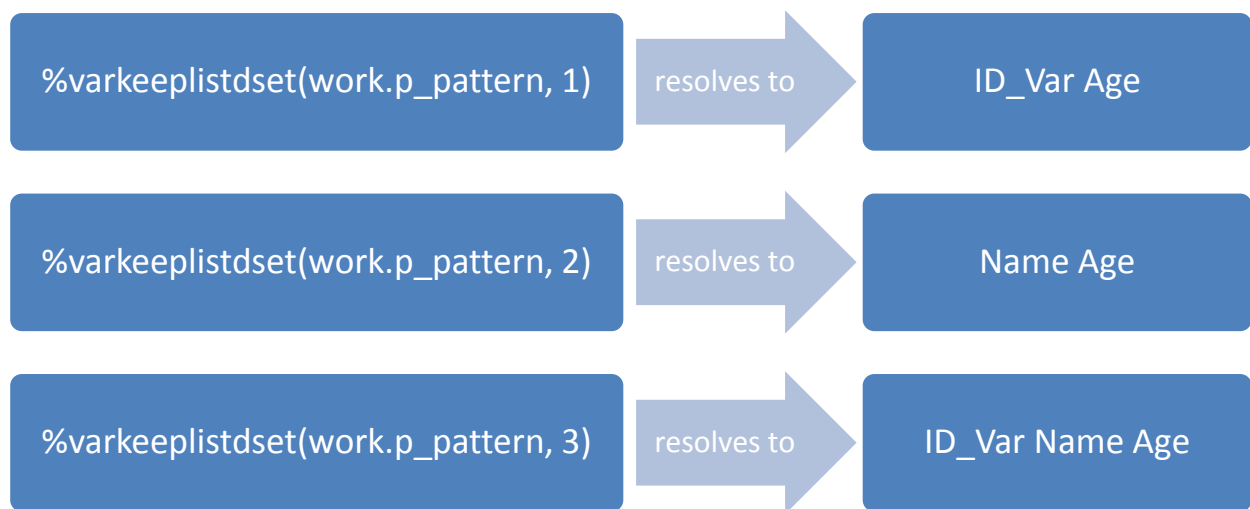
With the two lists sourced in this way, *Varkeeplistdset* resolves in a similar fashion to *Varkeeplist*.

Operation and Restrictions

Varkeeplistdset is a pure macro. It can be used in any SAS code.

Example: Using Varkeeplistdset on the data set work.p_pattern

work.p_pattern		
ID_Var	Name	Age
1	0	1
0	1	1
1	1	1



VARLENGTHS

Function Style Macro

%varlengths(data set reference, list of variables)

Description

Varlengths accesses the data set referenced, and returns the length of the variables in the list of variables. The variables must be in the data set referenced, and are supplied as a space delimited list.

The macro resolves to a string of numbers equal in dimension to the list of variables. Each number represents the length of the variable in the same relative position in the original list.

Operation and Restrictions

Varlengths is a pure macro. It can be used anywhere in SAS code.

Example: Obtain the length of some variables in the data set **work.persons**

Name is a character variable of length = 7. The other variables are of type numeric and thus equal to 8.

work.persons		
ID_Var	Name	Age
1	Bob	8
2	Charlie	17
3	Rachel	25

%varlengths(work.persons, ID_Var Name)

resolves to

8 7

VARLISTFROMDSET

Function Style Macro

%varlistfromdset(data set reference)

Description

Varlistfromdset is an incredibly useful macro in the world of SAS programming.

It accepts a data set reference, and resolves to a space delimited list. The list consists of the variable names from the data set.

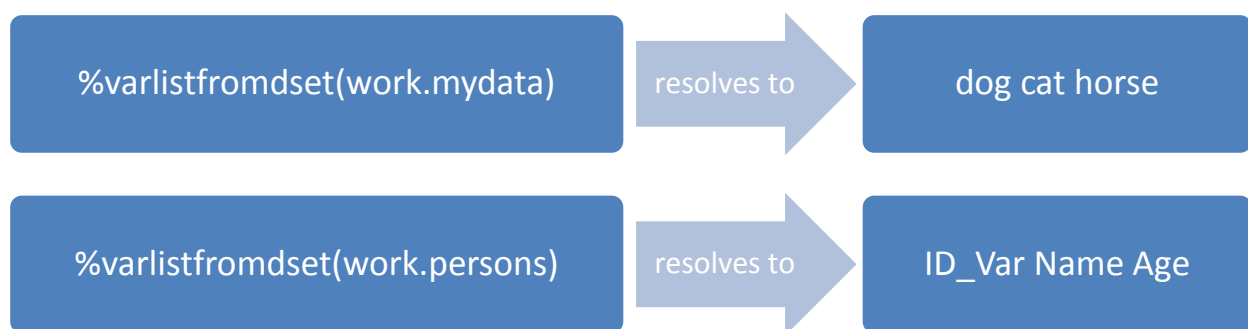
Operation and Restrictions

Varlistfromdset is a pure macro. It can be used in any SAS code.

Example: Obtain the names of the variables on a data set

work.mydata		
dog	cat	horse
1	2	3

work.persons		
ID_Var	Name	Age
1	Bob	8
2	Charlie	17
3	Rachel	25



VARPOS

Function Style Macro

%varpos(data set reference, list of variables)

Description

Varpos accesses the data set referenced, and returns the position of the variables listed within the data set. The list of variables must be delimited by spaces.

It returns the positions as numbers in a space delimited list.

Operation and Restrictions

Varpos is a pure macro. It can be used in any SAS code.

Example: Obtain the relative position of variables with a dataset

work.mydata		
dog	cat	horse
1	2	3

work.persons		
ID_Var	Name	Age
1	Bob	8
2	Charlie	17
3	Rachel	25

%varpos (work.mydata, dog cat)

resolves to

1 2

%varlistfromdset(work.persons,
ID_Var Age Name)

resolves to

1 3 2

VARINDSET

Function Style Macro

%varsindset(data set reference, list of variables)

Description

Varsindset allows the user to verify whether variables are present in a data set using macro code.

Varsindset checks that all of the variables supplied in the list are in fact found in the data set referenced.

If all variables are present, varsindset resolves to 1. If any of the variables are missing, varsindset resolves to 0.

Operation and Restrictions

Varsindset is a pure macro. It can be used in any SAS code.

Example: Check variables are in a data set

work.mydata		
dog	cat	horse
1	2	3

work.persons		
ID_Var	Name	Age
1	Bob	8
2	Charlie	17
3	Rachel	25

%varsindset (work.mydata, dog cat)

resolves to

1

%varsindset (work.mydata, dog cat Age)

resolves to

0

%varsindset (work.persons, ID_Var Age Name)

resolves to

1

VARTYPE

Function Style Macro

%vartype(data set reference, list of variables)

Description

Vartype accesses the data set referenced, and returns the type of the variables in the list. The variables must in fact be in the data set referenced, and are supplied as a space delimited list.

The macro resolves to a string of characters, equal in dimension to the list of variables. Each character represents the type of the variable in the same relative position in the original list. Character variables are demarcated with C, and numeric variables are demarcated with N.

Operation and Restrictions

Vartype is a pure macro. It can be used anywhere in SAS code.

Example: Determine the types of a list of variables in a data set

work.persons		
ID_Var	Name	Age
1	Bob	8
2	Charlie	17
3	Rachel	25

%vartype(work.persons, ID_Var Name Age)

resolves to

N C N

VORDSET

Function Style Macro

%vordset(data set reference)

Description

Vordset determines whether the data set referenced is a SAS view or whether it is in fact a physical data set.

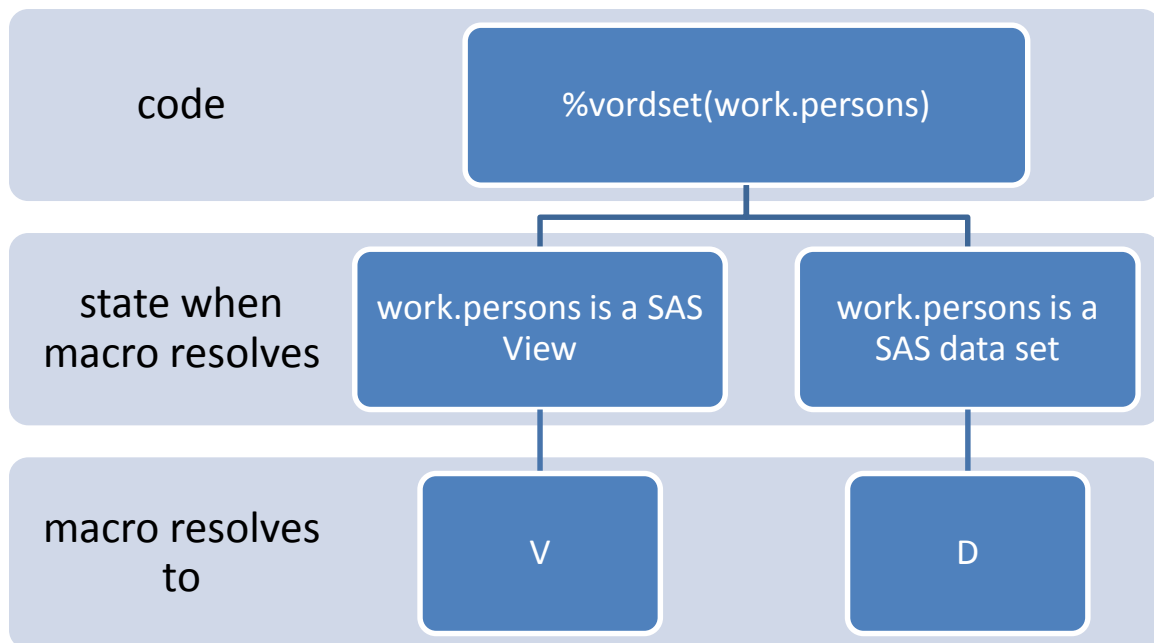
If the reference is to a view, vordset resolves to V.

If the reference is to a data set, vordset resolves to D.

Operation and Restrictions

Vordset is a pure macro. It can be used in any SAS code.

Example: Determine whether a reference is to a SAS view or a data set



WORDINLIST

Function Style Macro

%wordinlist(word, list)

Description

Wordinlist searches a list consisting of space delimited words for the original word.

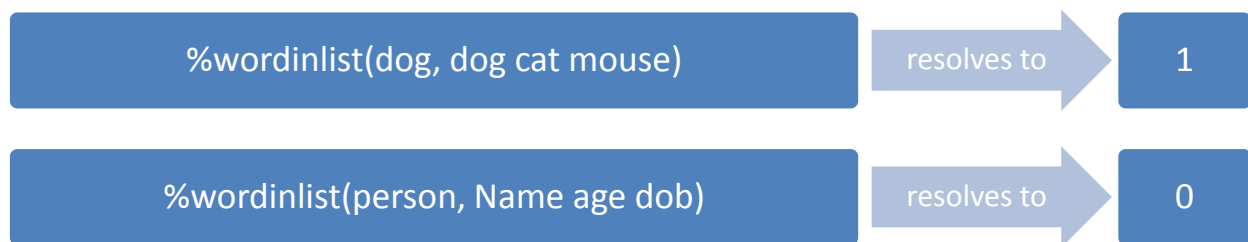
If the word is present in the list, Wordinlist resolves to 1.

If the word is not present, Wordinlist resolves to 0.

Operation and Restrictions

Wordinlist is a pure macro. It can be used in any SAS code.

Example: Determine whether a word exists in a list of words



New SAS Functions

The following section documents additional functions included in Icarus.

On installation, Icarus creates a data set containing the compiled functions in the user's work library³ called `Icarus_functions`. The addition of this custom function library and data set should automatically be handled by Icarus if no other custom function library has already been set by the user, allowing the user to use these functions in regular data step and PROC SQL programming just like native SAS functions.

These functions are in addition to those already supplied by SAS, many of which already have applications in data linking. I will not be describing the SAS internal functions in this document, however, a summary of some applicable to data linking include:

Soundex:

The SAS implementation of the soundex phonetic algorithm.

Geodist:

Calculate the geographic distance between two points

Spedis:

A SAS specific spelling distance function.

Complev:

The SAS implementation of the levenshtein spelling distance

Compgev:

A generalised spelling distance routine.

Compfuzz:

A fuzzy number comparison function

³ By default.

CAVERPHONE 2.0

Character Function

Caverphone(string)

Description

The original Caverphone was a phonetic encoding algorithm originally written by David Hood for the Caversham Project at the University of Otago, New Zealand. As such it was specifically designed for the project and not as a general phonetic algorithm. Caverphone 2.0 (the version implemented in Icarus) was then created as a general purpose English phonetic matching system.

The Icarus Caverphone SAS function is modelled on the instructions contained in the document “Caverphone Revisited” located at <http://caversham.otago.ac.nz/files/working/ctp150804.pdf> at the time of authorship.

A typo or two in the original documentation has been corrected in the Icarus implementation of Caverphone, as obvious intention is more important than literal instructions.

Furthermore, the Icarus implementation returns a missing value if it receives a missing value as input.

The returned string of the Caverphone function is of length 10.

User Note: Caverphone, like many other phonetic encoding algorithms has a natural tendency to be relatively expensive with respect to CPU cycles, it may be advisable to add a column to the data set in question representing the particular codes. Accessing these codes multiple times does not therefore incur the cost of multiple calls to the function each time the code is required, and they may then additionally be accessed via other techniques such as hash tables.

Because of the scale of the comparison space in the practice of data linking, it is rarely advisable to actual run phonetic encoding functions on the comparison space itself.

Example: The Caverphone function applied to a list of names

Caverphone = Caverphone(Names);

work.names	
Names	Caverphone
John	YN11111111
Michael	MKA11111111
Joseph	YSF11111111
Christopher	KRSTFA1111
Matthew	MTA11111111
Joshua	YSA11111111
Nicholas	NKLS111111
Jacob	YKP11111111
James	YMS11111111
Daniel	TNA11111111
Andrew	ANTRA111111
Christine	KRSTN111111
Mary	MRA11111111
Ashley	ASLA11111111
Jesse	YS1111111111
Alice	ALK11111111
Sarah	SRA11111111
Catherine	KTRN11111111
Emily	AMLA11111111
Britney	PRTNA111111
Caitlin	KTLN11111111

CHEBYSHEV DISTANCE

Numeric Functions

Chebyshev2(x_1, y_1, x_2, y_2)

Chebyshev3($x_1, y_1, z_1, x_2, y_2, z_2$)

Description

The two and three dimensional cases of the Chebyshev distance are defined respectively as:

2 Dimensions

$$\text{MAX}(|x_1 - x_2|, |y_1 - y_2|)$$

3 Dimensions

$$\text{MAX}(|x_1 - x_2|, |y_1 - y_2|, |z_1 - z_2|)$$

Each function requires numeric variables, and returns a numeric variable.

If any of the variables supplied to the function are missing, then the Chebyshev functions return missing.

Example: The Chebyshev functions applied to a set of coordinates

Chebyshev2 = Chebyshev2(x1, y1, x2, y2);

Chebyshev3 = Chebyshev3(x1, y1, z1, x2, y2, z2);

work.numbers							
x1	y1	z1	x2	y2	z2	Chebyshev2	Chebyshev3
13	99	29	63	45	29	54	54
92	1	4	71	86	30	85	85
10	75	7	15	27	44	48	48
95	87	44	46	78	32	49	49
86	82	13	24	75	24	62	62
81	36	62	98	77	53	41	41
56	65	76	55	64	3	1	73
50	30	22	55	21	75	9	53
88	74	33	84	76	28	4	5
47	1	85	85	36	85	38	38
88	26	36	42	29	34	46	46
6	7	22	94	14	61	88	88
13	30	66	69	1	44	56	56
41	3	17	39	91	92	88	88
14	78	99	6	36	11	42	88
20	77	37	28	63	41	14	14
23	58	46	76	23	94	53	53
17	1	2	85	4	71	68	69
64	64	81	95	85	43	31	38
29	25	5	8	59	22	34	34
1	83	53	71	63	99	70	70

CITYBLOCK DISTANCE

Numeric Functions

Cityblock2(x_1, y_1, x_2, y_2)

Cityblock3($x_1, y_1, z_1, x_2, y_2, z_2$)

Description

Cityblock distance is also commonly known under other names, such as Manhattan distance.

The two and three dimensional cases of the Cityblock distance are defined respectively as:

2 Dimensions

$$|x_1 - x_2| + |y_1 - y_2|$$

3 Dimensions

$$|x_1 - x_2| + |y_1 - y_2| + |z_1 - z_2|$$

Each function requires numeric variables, and returns a numeric variable.

If any of the variables supplied to the function are missing, then the Cityblock functions return missing.

Example: The Cityblock functions applied to a set of coordinates

Cityblock2 = Cityblock2(x1, y1, x2, y2);

Cityblock3 = Cityblock3(x1, y1, z1, x2, y2, z2);

work.numbers							
x1	y1	z1	x2	y2	z2	Cityblock2	Cityblock3
46	75	43	61	46	84	44	85
53	25	84	81	75	19	78	143
28	89	54	93	35	16	119	157
54	81	44	27	91	32	37	49
76	52	4	94	76	14	42	52
18	88	64	67	17	29	120	155
4	20	11	52	84	25	112	126
45	30	42	59	89	66	73	97
82	11	66	39	77	11	109	164
15	97	93	45	23	34	104	163
64	45	45	20	51	68	50	73
62	92	38	54	41	12	59	85
32	65	23	36	29	80	40	97
11	45	33	8	39	88	9	64
0	29	46	21	6	52	44	50
76	84	8	45	40	76	75	143
24	69	36	67	20	57	92	113
22	90	87	81	96	85	65	67
56	40	18	68	71	57	43	82
72	41	20	32	63	90	62	132
58	99	74	15	87	93	55	74

DOUBLE METAPHONE

Character Function

Dmetaphone(string, option number)

Description

The double metaphone function was authored by Lawrence Phillips, and published in the June 2000 issue of the C/C++ User's Journal.

Double metaphone is a phonetic algorithm. It was created to supersede the original metaphone algorithm written in 1990 by the same author.

Because double metaphone potentially returns up to two values, its implementation needs some adaption to make it SAS friendly. This has been achieved by adding an option number argument, which can be set to the value 1, 2 or 3.

Option number 1 returns only the first phonetic encoding.

Option number 2 returns only the alternative phonetic encoding. If there isn't an alternative encoding, option 2 returns the first phonetic encoding.

Option number 3 returns the first and second phonetic encodings, separated by a comma. It is perfectly acceptable for both encodings to be the same if there is only one possible encoding.

Furthermore, the Icarus implementation returns a missing value if it receives a missing value as input.

The returned string of the dmetaphone function is of length 9.

User Note: Double metaphone, like many other phonetic encoding algorithms has a natural tendency to be relatively expensive with respect to CPU cycles, it may be advisable to add a column to the data set in question representing the particular codes. Accessing these codes multiple times does not therefore incur the cost of multiple calls to the function each time the code is required, and they may then additionally be accessed via other techniques such as hash tables.

Because of the scale of the comparison space in the practice of data linking, it is rarely advisable to run phonetic encoding algorithms on the comparison space itself.

Example: The Dmetaphone function applied to a list of names

Dmetaphone1 = Dmetaphone(Names, 1);

Dmetaphone2 = Dmetaphone(Names, 2);

Dmetaphone3 = Dmetaphone(Names, 3);

work.names			
Names	Dmetaphone1	Dmetaphone2	Dmetaphone3
John	JN	AN	JN,AN
Michael	MKL	MXL	MKL,MXL
Joseph	JSF	HSF	JSF,HSF
Christopher	KRST	KRST	KRST,KRST
Matthew	M0	MTF	M0,MTF
Joshua	JX	AX	JX,AX
Nicholas	NXLS	NKLS	NXLS,NKLS
Jacob	JKP	AKP	JKP,AKP
James	JMS	AMS	JMS,AMS
Daniel	TNL	TNL	TNL,TNL
Andrew	ANTR	ANTR	ANTR,ANTR
Christine	KRST	KRST	KRST,KRST
Mary	MR	MR	MR,MR
Ashley	AXL	AXL	AXL,AXL
Jesse	JS	AS	JS,AS
Alice	ALS	ALS	ALS,ALS
Sarah	SR	SR	SR,SR
Catherine	KORN	KTRN	KORN,KTRN
Emily	AML	AML	AML,AML
Britney	PRTN	PRTN	PRTN,PRTN
Caitlin	KTLN	KTLN	KTLN,KTLN

EUCLIDEAN DISTANCE

Numeric Functions

Euclidean2(x_1, y_1, x_2, y_2)

Euclidean3($x_1, y_1, z_1, x_2, y_2, z_2$)

Description

Euclidean distance is the most common and familiar measure of distance.

The two and three dimensional cases of the Euclidean distance are defined respectively as:

2 Dimensions

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

3 Dimensions

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}$$

Each function requires numeric variables, and returns a numeric variable.

If any of the variables supplied to the function are missing, then the Euclidean functions return missing.

Example: The Euclidean functions applied to a set of coordinates

Euclidean2 = Euclidean2(x1, y1, x2, y2);

Euclidean3 = Euclidean3(x1, y1, z1, x2, y2, z2);

work.numbers							
x1	y1	z1	x2	y2	z2	Euclidean2	Euclidean3
75	53	27	44	83	75	43.1393	64.5368
57	53	27	10	46	18	47.5184	48.3632
99	6	21	84	33	31	30.8869	32.4654
51	7	53	77	63	93	61.7414	73.5663
51	42	49	16	45	87	35.1283	51.7494
55	47	66	73	36	62	21.0950	21.4709
20	0	94	53	10	36	34.4819	67.4759
69	1	3	46	57	99	60.5392	113.4945
55	59	46	29	33	61	36.7696	39.7115
25	75	16	8	81	85	18.0278	71.3162
89	16	10	43	14	34	46.0435	51.9230
35	65	59	34	68	37	3.1623	22.2261
15	50	87	28	49	43	13.0384	45.8912
76	67	76	65	85	27	21.0950	53.3479
27	30	3	76	35	22	49.2544	52.7920
47	40	18	7	61	7	45.1774	46.4973
97	59	76	44	31	41	59.9416	69.4118
29	22	99	94	56	59	73.3553	83.5524
61	90	37	60	77	78	13.0384	43.0232
33	49	0	76	0	46	65.1920	79.7872
78	37	63	76	7	79	30.0666	34.0588

EXPECTUNIQUE

Numeric Function

ExpectUnique(Inverse odds, Number in the Population)

Description

In the field of probability, there is a phenomenon that has come to be known as the birthday paradox. It is not a paradox in a technical sense, but another example of how probability can clash strongly with our intuition. The problem asks how likely it is for there to be any two people in a group that share the same birthday, and the paradox is expressed in how the probability of such a phenomenon rises as the number of people in the group rises. The unintuitive nature of this problem is that the probability of finding people who share the same birthday rises much faster than most would intuitively expect.

In data linking we're often interested in a related statistic. Not how likely it is to find someone who shares the same birthday, but how many people within a group we would expect to find that have unique birthdays. The reason for this is simple: uniqueness makes you easier to observe and link, and we can even use this statistic to make some simple predictions before partaking in a data linking project: given some measure of probabilistic information, say day, months or years of birth on two data sets, how many people do we think we could successfully identify in a population when only using such information⁴.

If we had a classroom of 20 children, and we wanted to know how many of them were expected to have unique birthdays in a 365 day year, we can use the ExpectUnique function to return such information.

Example: There are 20 children born in the same year. The year consists of 365 days, and we assume probability of birth is equally distributed across these days. How many children do we expect to find with unique birthdays?

$$\text{ExpectUnique}(365, 20) = 18.984$$

⁴ Of course, real world calculations have to take in a host of additional factors: missingness, quality, non-uniformity, error, migration, time, etc., but this idea of expected uniquely identifiable entities given a probability still works as a wonderful simplification.

FUZZNUMS

Numeric Functions

Highfuzz(x, y, z)

Genfuzz(x, y, z)

Lowfuzz(x, y, z)

Description

Icarus comes with three additional fuzzy number matching functions:

Highfuzz

If $x - y \leq z$ then Highfuzz returns 1.

Otherwise Highfuzz returns 0.

If x, y, or z are missing, Highfuzz returns missing.

Genfuzz

If $|x - y| \leq z$ then Genfuzz returns 1.

Otherwise Genfuzz returns 0.

If x, y, or z are missing, Genfuzz returns missing.

Lowfuzz

If $x - y \geq z$ then Lowfuzz returns 1.

Otherwise Lowfuzz returns 0.

If x, y, or z are missing, Lowfuzz returns missing.

Each function requires numeric variables, and returns a numeric variable.

Example: The Fuzznum functions applied to a set of coordinates

Highfuzz = highfuzz(x, y, z);

Genfuzz = genfuzz(x, y, z);

Lowfuzz = lowfuzz(x, y, z);

work.numbers					
x	y	z	Highfuzz	Genfuzz	Lowfuzz
8	5	4	0	1	1
0	1	3	1	1	0
7	7	3	1	1	1
1	1	8	1	1	1
4	5	9	1	1	0
3	2	0	0	0	0
2	8	7	1	1	0
9	9	6	1	1	1
0	2	7	1	1	0
9	6	7	0	1	1
8	7	4	0	1	1
3	2	7	0	1	1
8	2	2	0	0	0
4	1	1	0	0	0
4	5	5	1	1	0
8	2	1	0	0	0
8	1	3	0	0	0
0	0	2	1	1	1
2	1	5	0	1	1
3	2	4	0	1	1
5	9	9	1	1	0

HAMMING DISTANCE

Character Functions

Hamming(String1, String2)

Description

The Hamming distance between two strings of equal length are the number of positions at which the symbols comprising the two strings differ. Alternatively, one can think of the Hamming distance as the number of substitutions necessary to turn one string into another.

The two strings need to be of the same length in order to produce a meaningful return value for the function.

If the strings are not of equal length, or if either of the strings are missing, then Hamming returns missing.

Although Hamming accepts two character variables as input, its output is of type numeric.

Example: The Hamming function applied to two binary strings

Hamming = hamming(string1, string2);

work.numbers		
string1	string2	Hamming
10011010	11111001	4
01100011	00101010	3
01010100	11011010	4
01011001	01001101	2
00001100	11110110	6
10000111	10110001	4
11000111	01101111	3
11010101	01110111	3
01000010	11111101	7
10111000	10011011	3
01010010	00111111	5
11101000	00101101	4
10101100	11110101	4
11010000	10001010	4
00111000	10000110	6
10000001	01000110	5
11001101	01011111	3
01100110	10100111	3
11110000	11001001	4
00110011	00001111	4
10111110	11100100	4

JARO/WINKLER

Character Functions

Jaro(String1, String2)

Winkler(String1, String2, weight)

Description

The Jaro/Winkler algorithms have become standard string comparison functions in the data linking community, and thus are deserving of particular optimisation.

Matt Jaro originally authored the Jaro distance metric, and the application of the Jaro distance metric with an extra weight applied to consecutive matches at the beginning of the strings has come to be known as the Winkler function, after William E. Winkler of the US Census Bureau.

One of the more irritating features of the Winkler algorithm is the sheer variation in its implementations and lack of consistency between implementation and documentation. I have been blessed enough to work with the algorithm now in Python, SAS and C, and have re-written these algorithms for Icarus in my own subset-C implementation, having previously authored SAS versions of the function for use with PROC FCMP. The Icarus version involves what I consider a hack using the subset of C available in PROC PROTO to compile C wrapper functions to write an actual function rather than a wrapper.

Winkler's original C implementation contains numerous additional calculations, variations and adaptations specific to the project in which it was used. The Icarus implementation produces a function more faithful to the standard description, such as can be found at http://en.wikipedia.org/wiki/Jaro-Winkler_distance

String1 and String2 are the two strings to be compared. Jaro and Winkler return a number between 0 and 1, representing the similarity between the two strings.

It becomes apparent that Winkler necessarily behaves only as long as the weight factor is between 0 and 0.25. Winkler accepts a third argument, representing this additional weighting factor for consecutive matches in the first four characters. Standard practice at the Australian Bureau of Statistics, and around the world, has been to use a value of 0.1. It is my opinion that the original authors did not intend for the behaviour of the algorithm when the weight rises above 0.25, as it can create unintuitive values. For this reason, if the user enters a weight above 0.25, the Icarus implementation will behave as though 0.25 has been entered instead.

If either function is supplied with a missing value for the strings, it returns a missing value. Winkler/Jaro functions are designed for use on short strings. Icarus' current implementation will work to specification on strings up to 63 bytes in length, though I would not recommend their use on exceptionally long strings for theoretical reasons.

The Jaro Distance is defined as:

$$\frac{1}{3} \left(\frac{m}{s_1} + \frac{m}{s_2} + \frac{m - t}{m} \right)$$

s_1 represents the length of string 1.

s_2 represents the length of string 2.

m represents the number of *matching characters*.

t represents the number of *transpositions*.

If the number of matching characters between the two strings is equal to 0, then the Jaro distance is defined as 0, no doubt to avoid the problem of dividing by zero in the final term.

Two characters are only considered matching if they are within a minimum number of relative positions from each other. This distance is calculated by the following formula:

$$\left\lceil \frac{\text{MAX}(s_1, s_2)}{2} \right\rceil - 1$$

The Winkler Distance is defined as:

$$\text{Jaro} + (\alpha \times \beta(1 - \text{Jaro}))$$

α is the length of a common prefix at the start of both strings up to a maximum value of 4, and β is the weight (which is a constant scaling factor).

The two distances are symmetrical, in the sense that Jaro(string1, string2) should also equal Jaro(string2, string1). However, they are not metrics in the mathematical sense, because they do not obey the triangle inequality.

The SAS function strips blanks from the beginning and end of the string before comparison, so if you wish to compare such characters, you need to replace blanks in a string with other characters.

Example: Apply the Jaro and Winkler functions to a set of names

Jaro = jaro(Name1, Name2);

Winkler = winkler(Name1, Name2, 0.1);

work.names			
Name1	Name2	Jaro	Winkler
shackleford	shackelford	0.9697	0.9818
cunningham	cunnigham	0.9667	0.9800
campell	campbell	0.9583	0.9750
nichleson	nichulson	0.9259	0.9556
massey	massie	0.8889	0.9333
abroms	abrams	0.8889	0.9222
galloway	calloway	0.9167	0.9167
lampley	campley	0.9048	0.9048
dixon	dickson	0.7905	0.8324
frederick	fredric	0.9259	0.9556
michele	michelle	0.9583	0.9750
jesse	jessie	0.9444	0.9667
marhta	martha	0.9444	0.9611
jonathon	jonathan	0.9167	0.9500
julies	juluis	0.8889	0.9222
jeraldine	geraldine	0.9259	0.9259
yvette	yevett	0.8889	0.9000
tanya	tonya	0.8667	0.8800
dwayne	duane	0.8222	0.8400

MINKOWSKI DISTANCE

Numeric Functions

Minkowski2(x₁, x₂, y₁, y₂, p)

Minkowski3(x₁, x₂, x₃, y₁, y₂, y₃, p)

Description

Minkowski distance can be considered a generalisation of both the Euclidean distance and the Cityblock distance. Icarus comes with the respective 2 and 3 dimension functions for Minkowski distance.

The Minkowski distance can be defined as:

$$\left(\sum_{i=1}^n |x_i - y_i|^p \right)^{\frac{1}{p}}$$

Minkowski distance with p equal to 2 is the equivalent of the Euclidean distances.

Minkowski distance with p equal to 1 is the equivalent of the Cityblock distances.

If any of the inputs to the Minkowski distance functions are missing, then they return missing.

Example: Calculating the Minkowski Distance on a series of coordinates

MinkowskiA = Minkowski2(x1, x2, y1, y2, 1);

MinkowskiB = Minkowski3(x1, x2, x3, y1, y2, y3, 1);

MinkowskiC = Minkowski2(x1, x2, y1, y2, 2);

MinkowskiD = Minkowski3(x1, x2, x3, y1, y2, y3, 2);

work.numbers									
x1	x2	x3	y1	y2	y3	MinkowskiA	MinkowskiB	MinkowskiC	MinkowskiD
94	22	36	44	63	66	91	121	64.6607	71.2811
34	19	2	66	34	85	47	130	35.3412	90.2109
97	36	84	81	68	22	48	110	35.7771	71.5821
4	34	10	9	11	83	28	101	23.5372	76.7007
70	29	9	16	11	63	72	126	56.9210	78.4602
8	44	31	93	90	49	131	149	96.6488	98.3107
17	3	53	47	1	14	32	71	30.0666	49.2443
47	90	83	90	22	53	111	141	80.4550	85.8662
12	49	80	94	19	8	112	184	87.3155	113.1724
64	87	59	10	67	14	74	119	57.5847	73.0821
80	92	51	24	22	75	126	150	89.6437	92.8009
50	65	63	59	24	88	50	75	41.9762	48.8569
70	2	43	89	97	57	114	128	96.8814	97.8877
69	48	6	96	10	88	65	147	46.6154	94.3239
6	63	39	60	66	8	57	88	54.0833	62.3378
67	15	80	91	18	55	27	52	24.1868	34.7851
96	21	59	89	48	11	34	82	27.8927	55.5158
62	22	2	77	39	76	32	106	22.6716	77.3951
45	63	8	39	47	31	22	45	17.0880	28.6531
67	60	45	98	51	19	40	66	32.2800	41.4488
9	79	46	54	93	59	59	72	47.1275	48.8876

NYSIIS

Character Function

Nysiis(String)

Description

The New York State Identification and Intelligence System (NYSIIS) is a phonetic coding algorithm.

Various versions and documentations can be found at:

http://en.wikipedia.org/wiki/New_York_State_Identification_and_Intelligence_System

The Icarus implementation was based upon the original code found at:

<http://www.dropby.com/nysiisOriginal.js>

It has been mildly adjusted to make it more SAS friendly. It won't return arbitrary messages to the user about no numeric being allowed like in the original. Instead it converts all text to uppercase and removes non-alpha characters before calculating the NYSIIS code.

If fed missing input, NYSIIS returns missing.

NYSIIS returns a character string that has a length of 6.

Example: Apply the NYSIIS phonetic algorithm to a list of names

NYSIIS = nysiis(Name);

work.names	
Name	NYSIIS
John	J
Michael	MACAL
Joseph	JASAF
Christopher	CHRAST
Matthew	MAT
Joshua	JAS
Nicholas	NACAL
Jacob	JACAB
James	JAN
Daniel	DANAL
Andrew	ANDR
Christine	CHRAST
Mary	MARY
Ashley	ASLY
Jesse	JAS
Alice	ALAC
Sarah	SAR
Catherine	CATARA
Emily	ENALY
Britney	BRATNY
Caitlin	CATLAN

Hash Macros

The programs in this section use the SAS hash object to perform operations, summary statistics, and the production of commonly required derivative data sets.

Naturally, other macros also use the hash object, but these macros are categorised here because they do so as part of their most fundamental operation and in a relatively simple fashion instead of being tied to a particular analytical or theoretical practice.

There are, of course, other procedures and methods in SAS that may be used to arrive at the same result as these macros.

Whether the use of the hash macros is preferable, faster or uses less resources than other options depends upon the SAS environment and the specific circumstances, as well as the amount of programming the user can be bothered to write to mimic their behaviour and efficiency.

Additionally some procedures in SAS are naturally multi-threaded, and this may allow them to outperform the hash macros in practice.

On the other hand, the hash macro option commonly is both faster and uses less memory than the same activity in other SAS procedures, as there is limited additional overhead above the algorithmic creation and access of the hash object for the statistics in question.

In practicality, I have found the hash macros to commonly be some of the fastest methods to generate desired results.

HASHCOUNT

Parameterised Macro

```
%hashcount(  
  
DataSet = ,  
  
Vars = ,  
  
CountVar = _DJM_count,  
  
DorV = D,  
  
Outdata = work.counted,  
  
Exp = 12  
);
```

Description

Hashcount produces a data set that counts the number of times each unique combination of values appears within a data set.

If only one variable is supplied, then Hashcount will tell you how many times the values contained in that variable appear in a data set. If multiple variables are supplied, then Hashcount will tell you how many times the unique combination of values contained in all the variables appear in a data set.

The DataSet and Vars parameters must be supplied to the macro, but the user is free to exclude all the other arguments and only supply them as necessary.

Parameters

DataSet: Determines the data set which will be analysed. This parameter must be supplied.

Vars: Determines the variables whose unique values will be counted. This parameter must be supplied. You can supply a combination of variables with a space delimited list.

CountVar: Is the name of the variable that will be used to store the counts of each variable value. If this parameter is not supplied by the user, then the variable will be called _DJM_count.

DorV: This parameter may take the value of either D or V. It determines whether the output will be defined as a SAS data set, or as a SAS data step view. If the user does not supply this parameter, it defaults to the value of D.

Outdata: Defines the output data set. If not supplied, it defaults to work.counted.

Exp: Like most hash centric macros, the macro also accepts a parameter that sets the exp property of the Hash Object. This property sets, as a power of 2, the number of “buckets” found in the resulting hash table. The default value is 12.

Example Dataset

work.example			
DAY	MONTH	YEAR	SEX
22	5	1941	F
21	9	2001	F
26	1	1990	F
22	9	1979	F
26	4	1995	M
22	3	1933	M
11	1	1975	M
31	3	1967	F
25	4	1995	F
4	1	1991	M
9	1	2000	F
3	9	1935	F
16	7	1983	F
19	5	1956	M
8	.	2000	F
27	7	1954	M
19	11	1997	M
13	.	2001	M
16	12	2003	F
24	12	1947	M
19	11	1930	F

Example 1: Using hashcount to count the number of people born in each month

```
%Hashcount(  
DataSet = work.example,  
Vars = Month,  
CountVar = Count,  
Outdata = work.counted  
);
```



work.counted	
MONTH	COUNT
9	3
.	2
5	2
7	2
1	4
11	2
3	2
4	2
12	2

Example 2: Using hashcount to count each sex born in each year

```
%Hashcount(  
DataSet = work.example,  
Vars = Year Sex,  
CountVar = Count,  
Outdata = work.counted  
);
```



work.counted		
YEAR	SEX	COUNT
1933	M	1
2001	F	1
2001	M	1
1990	F	1
1983	F	1
1947	M	1
1935	F	1
2003	F	1
1991	M	1
1975	M	1
1995	F	1
1995	M	1
1941	F	1
1997	M	1
1954	M	1
1930	F	1
1979	F	1
1967	F	1
1956	M	1
2000	F	2

HASHDISTINCT

Parameterised Macro

```
%hashdistinct(  
  
  DataSet = ,  
  
  Vars = ,  
  
  DorV = D,  
  
  Outdata = work.distinct,  
  
  Exp = 12  
);
```

Description

Hashdistinct is used to produce a data set that contains the unique values that appear within a data set for any given selection of variables.

If one variable is supplied, Hashdistinct will create a data set consisting of that variable and the unique values contained within said variable. If multiple variables are supplied, Hashdistinct will contain those variables on the output data set. Each observation will be a unique combination of values contained within the original data set for those respective variables.

The DataSet and Vars parameters must be supplied to the macro, but the user is free to exclude all the other arguments and only supply them as necessary.

Parameters

DataSet: Determines the data set which will be analysed. This parameter must be supplied.

Vars: Determines the variables whose unique values will be found. This parameter must be supplied. More than one variable can be supplied via a space delimited list.

DorV: This parameter may take the value of either D or V. It determines whether the output will be defined as a SAS data set, or as a SAS data step view. If the user does not supply this parameter, it defaults to the value of D.

Outdata: Defines the name of the output data set. If not supplied, it defaults to work.distinct.

Exp: Like most hash centric macros, the macro also accepts a parameter that sets the exp property of the Hash Object. This property sets, as a power of 2, the number of “buckets” found in the resulting hash table. The default value is 12.

Example Dataset

work.example			
DAY	MONTH	YEAR	SEX
27	5	1993	F
14	7	1943	M
13	7	1983	M
6	3	1951	M
10	3	1999	F
20	2	1987	F
13	9	1955	M
.	.	1985	F
1	10	2003	F
.	2	1987	F
16	1	1940	M
10	3	1958	F
5	7	1992	M
26	11	1926	F
15	.	.	M
13	8	1980	M
2	6	2000	M
20	2	1955	F
5	2	1968	M
21	4	1951	M
9	.	1946	M

Example: Using hashdistinct to produce the unique values of Month and Sex

```
%hashdistinct(  
DataSet = work.example,  
Vars = Month Sex,  
Outdata = work.unique  
);
```



work.unique	
MONTH	SEX
8	M
11	F
4	M
1	M
2	F
2	M
7	M
9	M
6	M
3	F
3	M
.	F
.	M
10	F
5	F

HASHISIN

Parameterised Macro

```
%hashisin(  
  
DataSet = ,  
  
InDataSet = ,  
  
Vars = ,  
  
InVars = ,  
  
DorV = D,  
  
Outdata = work.lsln,  
  
Exp = 12  
);
```

Description

The hashisin macro provides a way to check which values from one data set can also be found in variables from another data set. It returns a data set that contains the variables from the first dataset whose values are found in the specified variables of the second data set.

Parameters

DataSet: The first data set, containing the variables you are interested in.

InDataSet: The reference to the second data set, which contains the variables the original data set variables will be checked against.

Vars: The variables on the first data set. Supplied via a space delimited list.

InVars: The variables on the second data set that will be used to check the variables from the first data set. Supplied via a space delimited list.

DorV: Whether you wish for the output to be a physical data set or a data step view. Valid options are “D” or “V”. If the user does not supply this parameter, it defaults to “D”.

Outdata: The name of the output data set or view. If the user does not supply this parameter, it defaults to “work.IsIn”.

Exp: The variable that determines “buckets” used in the SAS hash object as a power of two. Valid values are between 1 and 20 inclusive. Defaults to 12 if not supplied.

Example datasets

work.example1			
ID_Var	Name	mob	yob
1	Charlie	7	1945
2	Enoch	6	1932
3	Xiu	7	1973
4	Carlos	3	1982
5	Sophie	1	1991
7	Marcel	.	1983
8	Akiko	2	1991

work.example2			
ID_Var	Name	mob	yob
1	Janice	8	1973
2	Jesse	7	1991
3	Roger	.	1965
4	Ella	1	1918
5	Frank	1	1991
7	Sarah	.	.
8	Yan	12	1982

Example: Find the values of mob, from work.example1 that are also in mob on work.example2

```
%hashisin(  
  DataSet=work.example1,  
  Vars=mob,  
  InDataSet=work.example2,  
  InVars=mob);
```



work.IsIn	
mob	
7	
1	
.	

HASHISNOTIN

Parameterised Macro

```
%hashisnotin(  
  
DataSet = ,  
  
NotInDataSet = ,  
  
Vars = ,  
  
NotInVars = ,  
  
DorV = D,  
  
Outdata = work.IsNotIn,  
  
Exp = 12  
  
);
```

Description

The hashisnotin macro provides a way to check which values from one data set cannot be found in variables on another data set. It returns a data set that contains the variables from the first dataset whose values are not found in the variables on the second data set.

Parameters

DataSet: A reference to the first data set, containing the variables you are interested in.

NotInDataSet: The reference to the second data set, which the original data set variables will be checked against.

Vars: The variables on the first data set that will be checked and returned. Supplied via a space delimited list.

NotInVars: The variables on the second data set that will be checked against the variables from the first data set. Supplied via a space delimited list.

DorV: Whether you wish for the output to be a physical data set or a data step view. Valid options are “D” or “V”. If the user does not supply this parameter, it defaults to “D”.

Outdata: The name of the output data set or view. If the user does not supply this parameter, it defaults to “work.IsNotIn”.

Exp: The variable that determines the number of “buckets” used in the SAS hash object as a power of two. Valid values are between 1 and 20 inclusive. Defaults to “12” if not supplied.

Example datasets

work.example1			
ID_Var	Name	mob	yob
1	Charlie	7	1945
2	Enoch	6	1932
3	Xiu	7	1973
4	Carlos	3	1982
5	Sophie	1	1991
7	Marcel	.	1983
8	Akiko	2	1991

work.example2			
ID_Var	Name	mob	yob
1	Janice	8	1973
2	Jesse	7	1991
3	Roger	.	1965
4	Ella	1	1918
5	Frank	1	1991
7	Sarah	.	.
8	Yan	12	1982

Example: Find the values of mob, from work.example1 that are not in mob on work.example2

```
%hashisnotin(  
  DataSet=work.example1,  
  Vars=mob,  
  InDataSet=work.example2,  
  NotInVars=mob);
```



work.IsNotIn	
mob	
2	
3	
6	

HASHJOIN

Parameterised Macro

%hashjoin(

DataSetA = ,

DataSetB = ,

JoinVars = ,

JoinVarsA = ,

JoinVarsB = ,

DataVars = ,

DataVarsA = ,

DataVarsB = ,

PrefixA = ,

PrefixB = ,

Jointype = IJ,

DorV = D,

Outdata = work.joined,

Exp = 12,

ForceA = N,

ForceAKey = N,

ForceB = N,

ForceBKey = N,

ExcludeMissings = N

);

Description

The hashjoin macro is for the purpose of quick, flexible, and efficient joins and comparisons between two data sets. What was previously the domain of back-end SQL joins now resides in data step hash tables algorithms implemented via a sophisticated macro.

At first glance it can be quite intimidating given the number of options available. But many options are related quite intimately to each other, such that there would never be a situation when all possible parameters could be supplied at the same time. Supplying some parameters necessarily excludes the possibility of supplying others.

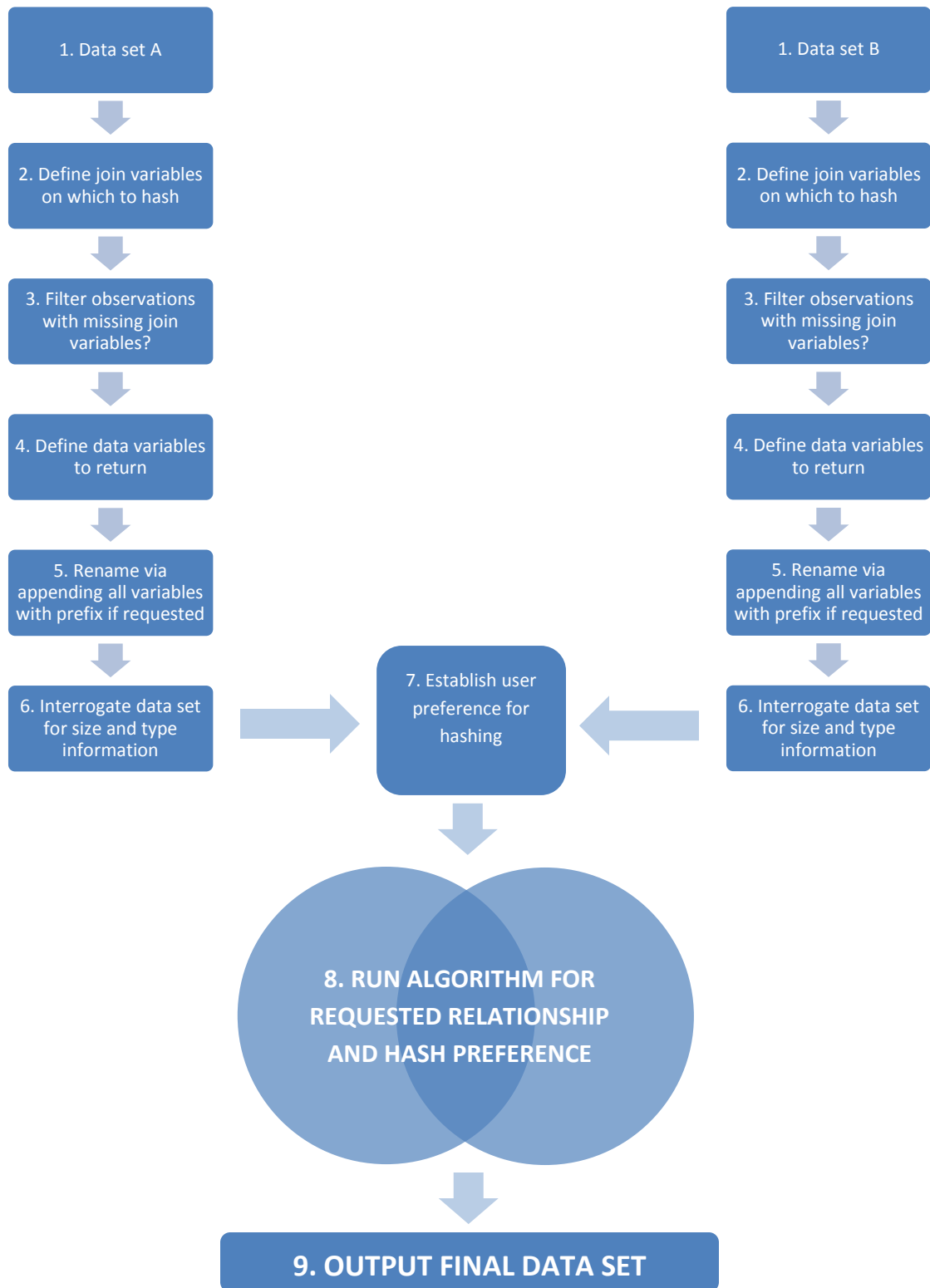
HashJoin requires a substantial amount of theory to fully understand its potential. Just describing the parameters as I have with previous macros will leave the user confused rather than enlightened.

Instead, I must explain the overall logic of the macro, and the relationship between the parameters. You can then see where each option fits in the larger picture, and how you can shift the operation of the macro at various points.

First comes some HashJoin theory on the relationships that exist between two different data sets. Users of SQL may be on familiar ground here, but I must also stress that the relationships in the hashjoin macro, while superficially similar, are not identical with the terms used to describe relationships in SQL operations.

With this theory, you can express most common relationships between two data sets, and the output created from those relationships will be produced via some of the most efficient code possible.

The Hash Join Flowchart



Hashjoin Data Set Relationship Theory

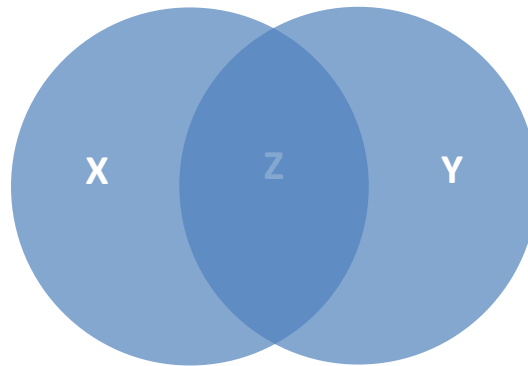
In the hashjoin macro, the relationship between any two data sets can be visualised via Venn Diagrams. Each circle represents a respective data set.



A selection of variables must be chosen as the join variables (JoinVars) between the two.

We define Z , represented by the union of the two circles in the diagram below, as the subset of the Cartesian product $A \times B$ where the values of the JoinVars in Data Set A equal the values of the JoinVars in Data Set B. X can be defined as those records in Data Set A where the values of the JoinVars are not found in the JoinVars in Data Set B. Conversely, Y is defined as those records in Data Set B where the values of the JoinVars are not in the JoinVars in Data Set A.

X and Y are not subsets of the Cartesian product $A \times B$. Rather, X is a subset of A and Y is a subset of B .



$$Z = \{(a, b) \in (A \times B) \mid \text{joinvar}_{a1} = \text{joinvar}_{b1} \bigwedge \cdots \bigwedge \text{joinvar}_{an} = \text{joinvar}_{bn}\}$$

$$X = \{a \in A \mid \{\text{joinvar}_{a1} \cdots \text{joinvar}_{an}\} \neq \{\text{joinvar}_{b1} \cdots \text{joinvar}_{bn}\} \forall b \in B\}$$

$$Y = \{b \in B \mid \{\text{joinvar}_{b1} \cdots \text{joinvar}_{bn}\} \neq \{\text{joinvar}_{a1} \cdots \text{joinvar}_{an}\} \forall a \in A\}$$

Hashjoin Data Set Relationship Definition

Having defined the sets X, Y, and Z, we can now explain the language and keywords the hashjoin macro uses to define the data set the user wishes to be returned and the algorithms that will be created and run.

Language is subtly similar to SQL, but not identical.

$$IJ = \text{Inner Join} = Z$$

$$FI = \text{Full Inner Join} = X + Y + Z$$

$$LO = \text{Left Outer Join} = X$$

$$RO = \text{Right Outer Join} = Y$$

$$LI = \text{Left Inner Join} = X + Z$$

$$RI = \text{Right InnerJoin} = Y + Z$$

$$FO = \text{Full Outer Join} = X + Y$$

Contemporary Parallels to the Hashjoin Relationships

The relationships can initially appear quite confusing. As it happens, several of these are practically synonymous with other concepts in data linking practice.

The Inner Join (IJ) is roughly equivalent in data linking parlance to basic blocking, and also to the inner join in SQL. Inner Joins can behave like a one-to-one join, a one-to-many join, or a many-to-many join depending on the context.

Left Inner (LI) and Right Inner (RI) are roughly equivalent to the Left Outer Join and Right Outer Join of SQL. When the value of the JoinVars are unique, it is the equivalent of merging records from one data set onto another data set. The SQL Full Outer Join concept has an equivalent in hashjoin's Full Inner Join (FI).

Finally, the Left Outer Join (LO) and Right Outer Join (RO) can be more commonly thought of as returning those records with keys that are not in the alternate data set, and the Full Outer Join (FO) is the combined results of both of these.

Regular Hashing vs. Key Hashing

PROC SQL typically uses the hash technique to join 2 data sets when the smaller one can be hashed into memory.

Hashjoin, on the contrary, has two techniques it can use.

The first is similar to the regular SQL technique of hashing requested keys and accompanying data variables into memory.

The second technique is typically used when the data sets concerned are too large. Rather than storing all requested variables in the hash, only the join variables themselves are hashed, and appended with an index number representing the position of each respective record on the data set. The numbers are used to access the data sets themselves using direct access rather than obtaining the data variables from the hash object itself.

Obviously there are a few caveats to using this technique. It doesn't solve all problems, and there is a performance hit to be taken for using direct access rather than sequential access or full in-memory options. It also doesn't follow for exceptionally large data sets that even the keys + index number will fit into memory.

But it does allow one more option for the user.

Parameters: Stepping through the HashJoin Macro

Step 1: Choose Data Set A and Data Set B

DataSetA: This is the reference to Data Set A.

DataSetB: This is the reference to Data Set B.

Step 2: Define the join variables on which to hash

The first option is to supply only the JoinVars parameter, which is acceptable if the variables are named the same on both data sets. If the variables are not named the same, then the macro allows the user to supply JoinVarsA and JoinVarsB to signify their respective names instead.

Parameters are to be supplied as space delimited lists.

JoinVars: A parameter to supply the name of the variables that will be used to join the data sets if the variables are named the same on both data sets. Do not supply if using the JoinVarsA and JoinVarsB parameters.

JoinVarsA: The parameter to supply the name of the variables from Data Set A that will be used to join the data sets. Use this parameter if the variables are named differently on both data sets. Do not supply if using the JoinVars parameter.

JoinVarsB: The parameter to supply the name of the variables from Data Set B that will be used to join the data sets. Use this parameter if the variables are named differently on both data sets. Do not supply if using the JoinVars parameter.

Step 3: Filter observations with missing join variables?

ExcludeMissings: This parameter accepts either N or Y. If the user does not supply it, it defaults to N. If it is set to Y, then Data Set A and Data Set B are redefined to be the subset of records which have no missing values in any of the join variables.

This is important, because in data linking, and in many administrative data sets, there can be high levels of missing variables. Combining records with missing variables as though they were actual matches is not often what the user intends.

Step 4: Define data variables to return

These are the variables that you actually wish to be returned in the resulting data set.

You can supply no parameters in this category, and the macro will automatically attempt to obtain all the variables from both data sets.

Parameters are to be supplied as space delimited lists of variable names.

DataVars: A parameter to supply the name of the variables that will be placed in the output data set. Use this parameter if the variables you wish to be returned are named the same on both data sets, and if you actually wish for those variables to be returned from both data sets. Do not supply if using the DataVarsA and DataVarsB parameters. Do not supply if you wish the macro to automatically retrieve all the variables from both data sets.

DataVarsA: A parameter to supply the name of the variables from Data Set A that will be placed in the output data set. Do not supply if you use the DataVars parameter. Make sure to include the JoinVars as well if you actually want them returned. Do not supply if you wish the macro to automatically retrieve all the variables from both data sets.

DataVarsB: A parameter to supply the name of the variables from Data Set B that will be placed in the output data set. Do not supply if you use the DataVars parameter. Make sure to include the JoinVars if you actually want them returned. Do not supply if you wish the macro to automatically retrieve all the variables from both data sets.

Step 5: Rename via appending all variables with prefix if requested

Variables across data sets often use similar names to represent similar data, but it doesn't follow that they both contain the same values. These two parameters let you supply a prefix to append to all the variables from each respective data set, which can avoid these potentially awkward clashes.

PrefixA: A prefix that will be appended to all the DataVars from Data Set A. Do not supply if you wish for there to be no prefix appended. The FO join type requires that you differentiate between the variables on the two data sets, and so it is likely that you must supply at least one prefix argument, whether it be PrefixA or PrefixB.

PrefixB: A prefix that will be appended to all the DataVars from Data Set B. Do not supply if you wish for there to be no prefix appended. The FO join type requires that you differentiate between the variables on the two data sets, and so it is likely that you must supply at least one prefix argument, whether it be PrefixA or PrefixB.

The user is responsible for supplying arguments which ensure there are no ambiguous variable names or variables named the same requested to be output from both data sets. The prefixes should ideally be set to something other than blank to ensure the algorithms work.

Step 6: Interrogate data sets for size information

The HashJoin macro will attempt to gather dimension statistics from the two data sets that have been referenced.

If a data set supplied to the macro is actually a SAS View, it assumes this is because the view is too large, and does not interrogate it for dimensions. If both data sets are SAS Views, it uses default algorithms without ascertaining the size of any data set.

Otherwise the hashjoin macro obtains the dimensions of the data sets, JoinVars, and DataVars from each data set. It uses these totals and compares them to the amount of system memory that the SAS session has available.

The hashjoin macro gives preference to the data sets that are not SAS views, and then the data set that it estimates will fit into the smallest amount of memory for hashing.

If the data set is estimated to be too big to fit into memory, Hashjoin attempts to calculate the dimensions required for only hashing the keys + pointers to records in the data set. It then resorts to hashing those values instead, and uses direct access to retrieve the records.

If the hashes estimated are too big to fit into memory even when using this second technique, then hashjoin will usually abort with an error message.

You can, however, still force the hashjoin macro to attempt a particular join and hashing method regardless of these estimates using the parameters in the next section.

Step 7: Establish user preference for hashing

By default, hashjoin will develop a preference for which data set to hash based upon the results of the calculations and metadata it accessed in step 6.

However, there are 4 optional parameters the user can supply to force a particular style of hashing. You can supply and set one of the following parameters to Y if you wish to force Hashjoin to take that particular action.

These options let you force particular strategies even if hashjoin believes that there isn't enough memory available to do so, or even if the data set to be hashed is a view.

ForceA: Supply this parameter and set it to Y to force hashjoin to produce your requested relationship by hashing the join variables and data variables from data set A into memory.

ForceB: Supply this parameter and set it to Y to force Hashjoin to produce your requested relationship by hashing the join variables and data variables from data set B into memory.

ForceAKey: Supply this parameter and set it to Y to force Hashjoin to produce your requested relationship by hashing the join variables and an associated key from data set A into memory.

ForceBKey: Supply this parameter and set it to Y to force Hashjoin to produce your requested relationship by hashing the join variables and an associated key from data set B into memory.

Step 8: Choose the relationship

The type of join the Hashjoin macro performs is controlled by the Jointype parameter.

Jointype: The join type parameter can be set to one of the following values:

- IJ: Perform an Inner Join
- FI: Perform a Full Inner Join
- LI: Perform a Left Inner Join
- RI: Perform a Right Inner Join
- LO: Perform a Left Outer Join
- RO: Perform a Right Outer Join
- FO: Perform a Full Outer Join

If the Jointype parameter is not supplied by the user, Hashjoin assumes the user wishes to use an Inner Join, which is largely synonymous with basic blocking under orthodox probabilistic data linking practice, and is also the most common join type in most situations.

Step 9: Implement the requisite algorithm and output data set

The user can supply three more parameters that determine the nature of the algorithm run, and the data set that is output by the Hashjoin macro.

DorV: This parameter can take the value “D” or “V”, indicating whether the user wants the output to be a data set, or as a SAS data step view. If not supplied, it defaults to D.

Exp: This parameter can take a value 1 to 20. In powers of two, it supplies the number of buckets used in the SAS hash object within the hashjoin macro. If not supplied it defaults to 12.

Outdata: This parameter defines the reference of the data set that the Hashjoin macro will output. If not supplied, it defaults to work.joined.

Example Data Sets for Hashjoin:

work.example1			
ID_VAR	NAME	MOB	YOB
1	Charlie	7	1945
2	Enoch	6	1932
3	Xiu	7	1973
4	Carlos	3	1982
5	Sophie	1	1991
7	Marcel	.	1983
8	Akiko	2	1991

work.example2			
ID_VAR	NAME	MOB	YOB
1	Janice	8	1973
2	Jesse	7	1991
3	Roger	.	1965
4	Ella	1	1918
5	Frank	1	1991
7	Sarah	.	.
8	Yan	12	1982

Example 1: Create a SAS View called work.block representing the comparison space when blocking (inner join) on MOB with missing join variables excluded as per standard probabilistic linking practice. Return all variables in each data set, with a prefix of a_ appended for work.example1 variables, and prefix of b_ for work.example2 variables.

```
%Hashjoin(  
  DataSetA = work.example1,  
  DataSetB = work.example2,  
  JoinVars = MOB,  
  Prefixa = a_, Prefixb = b_,  
  Outdata = work.block,  
  DorV = V,  
  Excludemissings = Y  
);
```



work.block							
b_ID_VAR	b_NAME	b_MOB	b_YOB	a_ID_VAR	a_NAME	a_MOB	a_YOB
2	Jesse	7	1991	1	Charlie	7	1945
2	Jesse	7	1991	3	Xiu	7	1973
4	Ella	1	1918	5	Sophie	1	1991
5	Frank	1	1991	5	Sophie	1	1991

Example 2: Create a data set representing a left outer join of work.example1 with work.example2 on the variable MOB. No renaming is necessary.

```
%Hashjoin(  
DataSetA = work.example1,  
DataSetB = work.example2,  
Jointype = LO,  
JoinVars = MOB  
);
```



work.joined			
ID_Var	Name	mob	yob
2	Enoch	6	1932
4	Carlos	3	1982
8	Akiko	2	1991

Example 3: Forcing data set A to be hashed with the JoinVar/numeric key method to save on memory used while implementing an inner join on YOB. Returning only the NAME and MOB variables from both data sets. Appending “b_” to the resulting variables from the second data set to avoid clashing of similarly named variables.

```
%Hashjoin(  
DataSetA = work.example1,  
DataSetB = work.example2,  
JoinVars = YOB,  
DataVars = NAME MOB,  
ForceAKey = Y,  
Prefixb = b_  
);
```



work.joined			
b_NAME	b_MOB	NAME	MOB
Janice	8	Xiu	7
Jesse	7	Sophie	1
Jesse	7	Akiko	2
Frank	1	Sophie	1
Frank	1	Akiko	2
Yan	12	Carlos	3

Example 4: Return those records from both data sets with no corresponding matches on the variable MOB (full outer join).

Prefixes for this option must be supplied to let the algorithm differentiate between variables during hashing/comparison.

```
%Hashjoin(  
DataSetA = work.example1,  
DataSetB = work.example2,  
JoinVars = MOB,  
Jointype = FO,  
Prefixa = a_,  
Prefixb = b_  
);
```



work.joined							
b_ID_VAR	b_NAME	b_MOB	b_YOB	a_ID_VAR	a_NAME	a_MOB	a_YOB
.		.	.	2	Enoch	6	1932
.		.	.	4	Carlos	3	1982
.		.	.	8	Akiko	2	1991
1	Janice	8	1973	.		.	.
8	Yan	12	1982	.		.	.

HASHSORT

Parameterised Macro

```
%hashsort(  
  
DataSet = ,  
  
SortVars = ,  
  
DorV = D,  
  
AorD = A,  
  
Outdata = work.sorted,  
  
Exp = 12,  
  
TagSort = N  
);
```

Description

Hashsort was an experimental macro based on the question of whether there was any gain to be had in using a SAS hash object to sort a data set. Subsequently it is relatively simple.

Whether it in fact outperforms SAS's own PROC SORT for the equivalent activity is an open question. Given that PROC SORT can implement a multithreaded sort, the answer may increasingly be no. PROC SORT is also infinitely more flexible than hashsort, and so the user is encouraged to experiment to determine which of the two is the preferred method.

Parameters

DataSet: The reference to the data set that you wish to sort.

SortVars: The variables, supplied via a space delimited list, on which you wish to sort.

DorV: Determines whether the output will be a physical data set or a SAS data step view. Valid values are "D" or "V". Defaults to "D" if not supplied.

AorD: Determines whether the variables are sorted in ascending or descending order. Valid values are "A" and "D" respectively. Defaults to "A" if not supplied.

Outdata: Determines the name of the output data set. Defaults to work.sorted if not supplied.

Exp: The variable that determines the number of “buckets” used in the SAS hash object. Valid values are between 1 and 20 inclusive. Defaults to “12” if not supplied.

TagSort: The tagsort parameter can be set to “Y” or “N”. If not supplied it defaults to “N”. The tagsort option loads only the sort variables from the data set and an accompanying numeric tag into memory. The output data set is constructed by retrieving the records indicated from the sorted variables via the numeric tag and random access to the original data set.

Hashsort example data set

work.example			
ID_VAR	NAME	MOB	YOB
1	Charlie	7	1945
2	Enoch	6	1932
3	Xiu	7	1973
4	Carlos	3	1982
5	Sophie	1	1991
7	Marcel	.	1983
8	Akiko	2	1991

Example: Use hashsort to sort the data set work.example into a SAS view named work.sortview

```
%hashsort(  
DataSet = work.example1,  
SortVars = YOB,  
DorV = V,  
Outdata = work.sortview  
);
```



work.sortview			
ID_Var	Name	mob	yob
2	Enoch	6	1932
1	Charlie	7	1945
3	Xiu	7	1973
4	Carlos	3	1982
7	Marcel	.	1983
5	Sophie	1	1991
8	Akiko	2	1991

HASHSUM

Parameterised Macro

```
%hashsum(  
  
DataSet = ,  
  
SumVar = ,  
  
ClassVar = ,  
  
OutSumVar = ,  
  
DorV = D,  
  
Outdata = work.summed,  
  
Exp = 12  
);
```

Description

Hashsum uses the SAS hash object to sum a numeric variable, grouped into categories distinguished by the unique values of the ClassVar variables. It outputs the ClassVar variables and the sums derived from SumVar into a data set.

Parameters

DataSet: The data set which contains the variable you wish to sum.

SumVar: The name of the variable you wish to sum.

ClassVar: The variables by which the summed variable will be grouped.

OutSumVar: The name of the variable on the output dataset which holds the summed totals of the SumVar for each group. If this parameter is not submitted, it defaults to djm_sum.

DorV: Determines whether the output will be a physical data set or a SAS data step view. Valid values are D or V. Defaults to D if not supplied.

Outdata: Determines the name of the output data set. Defaults to work.summed if not supplied.

Exp: The variable that determines the number of buckets used in the SAS hash object. Valid values are between 1 and 20 inclusive. Defaults to 12 if not supplied.

hashsum example data set

work.sumexample			
ID_Var	Name	City	Income
1	Charlie	Honolulu	1900
2	Enoch	Brisbane	1500
3	Xiu	Honolulu	785
6	Carlos	Brisbane	183
5	Sophie	Chicago	456
7	Marcel	Chicago	1212
8	Akiko	Chicago	1000

Example: Sum the Income variable grouped by the City variable

```
%HashSum(  
DataSet = work.example1,  
SumVar=Income,  
ClassVar=City,  
OutSumVar=Total_Income  
)
```



work.summed	
City	Total_Income
Brisbane	1683
Honolulu	2685
Chicago	2668

Blocking and Indexing Macros

In data linking, the set of all possible comparisons between records from two data sets is a problem that scales poorly. By the time we deal with data of any significant size the comparison of all records with each other easily outweighs any computing power available today or in the foreseeable future.

For that reason, we need algorithms that can bring together records in which we are interested in a quick and efficient fashion without wasting time or resources.

Data linkers have traditionally called this technique blocking, and the most common form is essentially the equivalent of implementing an equality join or using simple indexes like in a database. SAS, of course, can create and use some forms of indexes with its usual procedures, but sometimes these are not fast or flexible enough.

The HashJoin macro, documented in an earlier section, may itself be thought of as a relatively quick, dynamic, and efficient method of implementing a simple blocking technique.

But we shouldn't be content to settle with either of those techniques.

This section contains a variety of advanced macros designed to efficiently bring together records from two data sets.

ICARUSINDEX

```
%icarushindex(  
  
DataSet = ,  
  
Vars = ,  
  
Index1 = work.Index1_1,  
  
Index2 = work.Index2_1,  
  
ExcludeMissings = N,  
  
Exp = 12  
  
);
```

Description

Icarushindex produces the indexes required for icarushindexjoin. Icarushindexdset also uses this macro to generate each of the indexes it produces, and can be thought of as an industrialisation of icarushindex.

Alternatively of course, the user can manually create these indexes by calling the icarushindex macro, and then use them by directing icarushindexjoin towards them. When creating multiple indexes with icarushindex for the use in icarushindexjoin, it is important to name them in the form of:

Root1i for Index1

Root2i for Index2

Where i is an increasing integer starting at 1 and incrementing to the number of index pairs that will be used by icarushindexjoin. Of course, to know why there would be index pairs, you have to know how icarushindex works...

IcarushIndex Theory

It is assumed that the user has a basic understanding of how a regular SAS or database index works by combining lookup values (keys) with the locations of records containing those values in a data set (locations). A trivial index may be visualised by the relationships in the following diagrams:

Example data set

work.example			
ID_Var	Name	City	Income
1	Charlie	Honolulu	1900
2	Enoch	Brisbane	1500
3	Xiu	Honolulu	785
6	Carlos	Brisbane	183
5	Sophie	Chicago	456
7	Marcel	Chicago	1212
8	Akiko	Chicago	1000

Representation of a trivial index on example data set using “City” as the key

Index using City in work.example	
City	Locations
Brisbane	2,4
Honolulu	1,3
Chicago	5,6,7

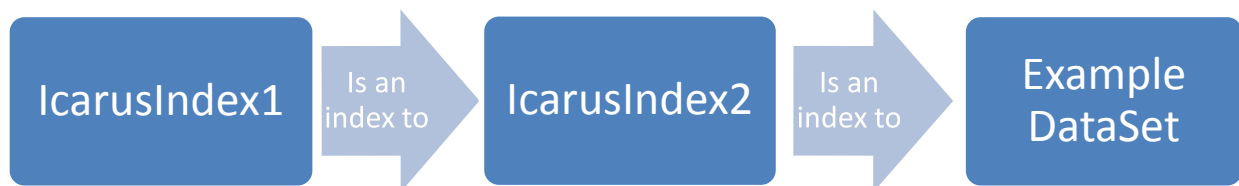
To find those records in the example data set where the variable City contains the values of “Brisbane”, “Honolulu” and “Chicago”, the original data set does not have to be sorted or directly searched. Instead, you could look at the index and discover the location quickly. Finding the value and location in the index is quicker and easier than searching through the larger and complex original data set.

An icarusindex can be thought of as taking this original idea and splitting it into two more constituent indexes represented in the next two tables:

IcarusIndex representations of the previous index

IcarusIndex1			IcarusIndex2	
City	_djm_start	_djm_end	_djm_pointer	
Brisbane	1	2	2	
Honolulu	6	7	4	
Chicago	3	5	5	
			6	
			7	
			1	
			3	

IcarusIndex1 can be thought of as an index to IcarusIndex2, which is itself an array of pointers working as an index to work.example with no explicit key information attached.



The values of 1 and 2 for the key of “Brisbane” in IcarusIndex1 says that the location information we are seeking for that key in the example data set can be found between records 1 and 2 inclusive in IcarusIndex2. Sure enough, these records in IcarusIndex2 hold the values 2 and 4, which are the same points found for “Brisbane” in our regular index.

Working through the same logic for the values of “Honolulu” and “Chicago”, we find the locations in IcarusIndex2 correspond to the locations in the original simple index.

Although it may not be initially obvious, IcarusIndex2 is sorted into non-overlapping blocks, whereby grouped records represent location information for specific key values. The location of these blocks, along with the keys to which they correspond are found by looking at the values in IcarusIndex1.

In the trivial example used here, it appears relatively unnecessary and overly complex to use IcarusIndex. Doing so for simple or small comparisons is overkill.

IcarusIndex comes to the fore when the data set being indexed become sufficiently large, and the values of the key variables can be found in many records.

Using the icarusindexjoin macro, IcarusIndex1 is hashed and held in memory for quick look up and access. No matter how many records match each key value, only two numbers and a hashed key value need to be in memory at any one time, minimising memory use. SAS uses the value in these hash tables to directly access IcarusIndex2 which, by default, continues to reside on disk.

Parameters

DataSet: The reference to the data set on which you wish to create an index.

Vars: A list of variables delimited by spaces. These variables make up the “key” for the index concerned.

Index1: The output IcarusIndex1 dataset. If the user does not supply this parameter, it defaults to the value work.Index1_1.

Index2: The output IcarusIndex2 dataset. If the user does not supply this parameter, it defaults to the value work.Index2_1.

As the user may wish to use multiple indexes with the icarusindexjoin macro, it is required that each pair of indexes manually defined be named in a similar fashion, with an incrementing last digit so that the program will be able to recognize that these indexes are meant for the same operation. For more information on the naming convention required for this use, see the documentation for the icarusindexjoin macro.

ExcludeMissings: Can be set to either Y or N. If it is set to Y then any records with a missing value for any of the key variables will be excluded from the index. This means it is impossible for those records to be compared with any other records when using that index/key. If the user does not supply this parameter, it defaults to N.

Exp: IcarusIndex uses a hash object to create Index1. This ensures the file can actually be hashed into memory. This setting accepts a number between 1 and 20, which sets the number of buckets in the hash object as a power of 2. If the user does not supply this parameter, it defaults to 12.

ICARUSINDEXDSET

```
%icarusedxset(  
  
  DataSet = ,  
  
  ControlDataset = ,  
  
  Index1Root = work.Icarusindex1_,  
  
  Index2Root = work.Icarusindex2_,  
  
  ExcludeMissings = N,  
  
  Exp = 12  
  
);
```

Description

Icarusedxset presents a more industrialised version of the icarusindex macro. It uses a specially formatted data set containing observations and variables to determine the keys used to create each pair of indexes. This data set is referred to as the Control Data Set, and icarusedxset will attempt to create as many index pairs as there are observations in the Control Data Set. In SAS macro code, each index pair takes the form:

&Index1Root.&i

&Index2Root.&i

Where i is an internal macro variable incrementing from 1 to the number of observations in the Control Data Set.

The Control Data Set is interesting because its form creates some convenient relations allowing it to be generated via analytical programs operating on patterns commonly found in SAS, Icarus and data linking in general.

Icarusindexdset Theory

The internal workings of an Icarus index are already explained in the theory section for the icarusindex macro and won't be repeated here. Instead, I focus on the properties of the Control Data Set, and how it is used to generate indexes.

Example data set

work.example			
ID_Var	Name	City	Income
1	Charlie	Honolulu	1900
2	Enoch	Brisbane	1500
3	Xiu	Honolulu	785
6	Carlos	Brisbane	183
5	Sophie	Chicago	456
7	Marcel	Chicago	1212
8	Akiko	Chicago	1000

Simple Control Data Set to create Icarus Indexes for work.example

work.control		
Name	City	Income
0	1	0
1	1	0
0	1	1

The Control Data Set must only contain variables with the same names as those found in the data set that it will be used to index. The quick of wit might also recognise that the control data set is similar in form to binary agreement patterns often found in data linking, and this is no accident. Writing a program to output a subset of specific agreement patterns is a very quick way of building a Control Data Set, and a data set of agreement patterns can itself be thought of as a Control Data Set.

Within the Control Data Set, each record/observation is used to construct the keys for the indexes that will be generated. One can think of the macro as obtaining two pieces of information from each observation: Firstly an array of the variable names, and secondly an array of equal dimensions consisting of the values for that observation. The second array masks/filters the first array, keeping only those values from the first array that correspond to a value of 1 in identical positions in the second array. In this fashion, the keys for the index pairs based upon each observation are formed.

Example key generation from observation 1

ARRAY 1

Name	City	Income
0	1	0

ARRAY 2



KEY

City

Example key generation from observation 2

ARRAY 1

Name	City	Income
1	1	0

ARRAY 2



KEY

Name	City
------	------

Example key generation from observation 3

ARRAY 1

Name	City	Income
0	1	1

ARRAY 2



KEY

City	Income
------	--------

In this way, the Control Data Set provides keys for multiple index pairs. Icarusindexdset then uses these keys to create multiple Icarus index pairs from the original data set.

Parameters

DataSet: The reference to the data set that will be indexed.

ControlDataset: The reference to the data set that will be used as the Control Data Set. Must be in the form expected for a Control Data Set.

Index1Root: The first index of an index pair is referenced by the pattern &Index1Root.&i, where i is an incremental counter up to the number of observations in the Control Data Set. If the user does not supply this parameter, it defaults to “work.IcarusIndex1_”.

Index2Root: The second index of an index pair is referenced by the pattern &Index2Root.&i, where i is an incremental counter up to the number of observations in the Control Data Set. If the user does not supply this parameter, it defaults to “work.IcarusIndex2_”.

ExcludeMissings: This flag determines whether variables with missing values for any of the key variables are indexed. Can be set to Y or N. If the user does not supply this parameter, it defaults to N.

Exp: This parameter sets the number of buckets used in the hash objects that are created in the process of generating the indexes. Valid values are 1 to 20. If the user does not supply this parameter, it defaults to 12.

ICARUSINDEXJOIN

```
%icarusexindexjoin(  
  
IndexedDataSet = ,  
  
IndexedDataSetVars = ,  
  
PrefixIndexedDataSet = ,  
  
  
  
OtherDataSet = ,  
  
OtherDataSetVars = ,  
  
PrefixOtherDataSet = ,  
  
  
  
ControlDataSet = ,  
  
  
  
Index1Root = work.Icarusexindex1_,  
Index2Root = work.Icarusexindex2_,  
  
  
  
FirstIndexNumber = ,  
LastIndexNumber = ,  
  
  
  
Outdata = work.IcarusexIndexJoined,  
  
DorV = V,  
  
ExcludeMissings = N,  
  
Exp = 12  
);
```


Description

Icarusindexjoin is the apex of the three “Icarusindex” macros. If the user is not familiar with the theory of the earlier two programs, it is recommended that they read the documentation on the icarusindex and icarusindexdset macros to understand the nature and operation of an Icarus index, and how one may be derived from a Control Data Set.

In summary, Icarusindexjoin creates an output data set containing comparisons between records in two data sets where the comparison space is defined by multiple equijoins defined either in a control data set or in a set of Icarus index pairs.

Regular blocking in the field of data linking is practically equivalent to using equijoin predicates in an SQL or database join. At the time of writing, most SQL implementations, including PROC SQL, are capable of using hash tables or indexes when presented with the simple case of an equijoin request. However, efficiently implementing more complex requests typically presents problems, such as when the join requested consists of a union between sets when each set can be individually represented by equijoin predicates.

Icarusindexjoin can be thought of as implementing a relatively efficient algorithm to return a subset of record pairs from the Cartesian product as defined by the union of multiple equijoin predicates.

In data linking parlance, it efficiently implements multiple blocking passes at once.

When two (or more) sets of record pairs are defined by a collection of equijoin predicates, there may be a number of record pairs that are found within the intersection of these sets. Running each of these joins individually duplicates these comparisons. Icarusindexjoin additionally identifies those records contained in the intersection of these sets, and ensures that the output data set does not contain more than one instance of these record pairs.

Icarusindexjoin requires that the join variables on the two data sets to be named the same.

A more detailed discussion of the Icarusindexjoin algorithm follows.

Icarusindexjoin Theory

Icarusindexjoin is the most complex macro in the family of multiple equijoin blocking macros which also include the multihashjoin and multihashpointjoin macros. It is recommended for the user to first read and comprehend the operations of icarusindex, icarusindexdset, multihashjoin and multihashpointjoin, for these either play a direct part in the operation of icarusindexjoin, or can be interpreted as earlier evolutionary stages of similar algorithms.

Multihashjoin introduces the notion of using a control data set to generate keys, and then using these keys to store data in multiple hash objects. Multihashpointjoin then introduces the concept that these memory based hash objects need not actually use resources to store the data itself, but merely keys and pointers to where the data is stored in the data set. In this way, it lessens the memory storage requirements of hashing while achieving relatively impressive performance.

IcarusIndexJoin uses less memory again to implement the hash objects. It does not need to store all pointers themselves in memory. Rather, it stores all of the pointers in an ordered array (or more factually, in this case in the form of a SAS data set) on disk. Only the unique keys and two pointers for each unique key value are stored in the hash object. These two pointers refer respectively to the start and end positions within the on-disk array. This array is loaded at these points when required, and the pointers inside it used to access records in the original data set.

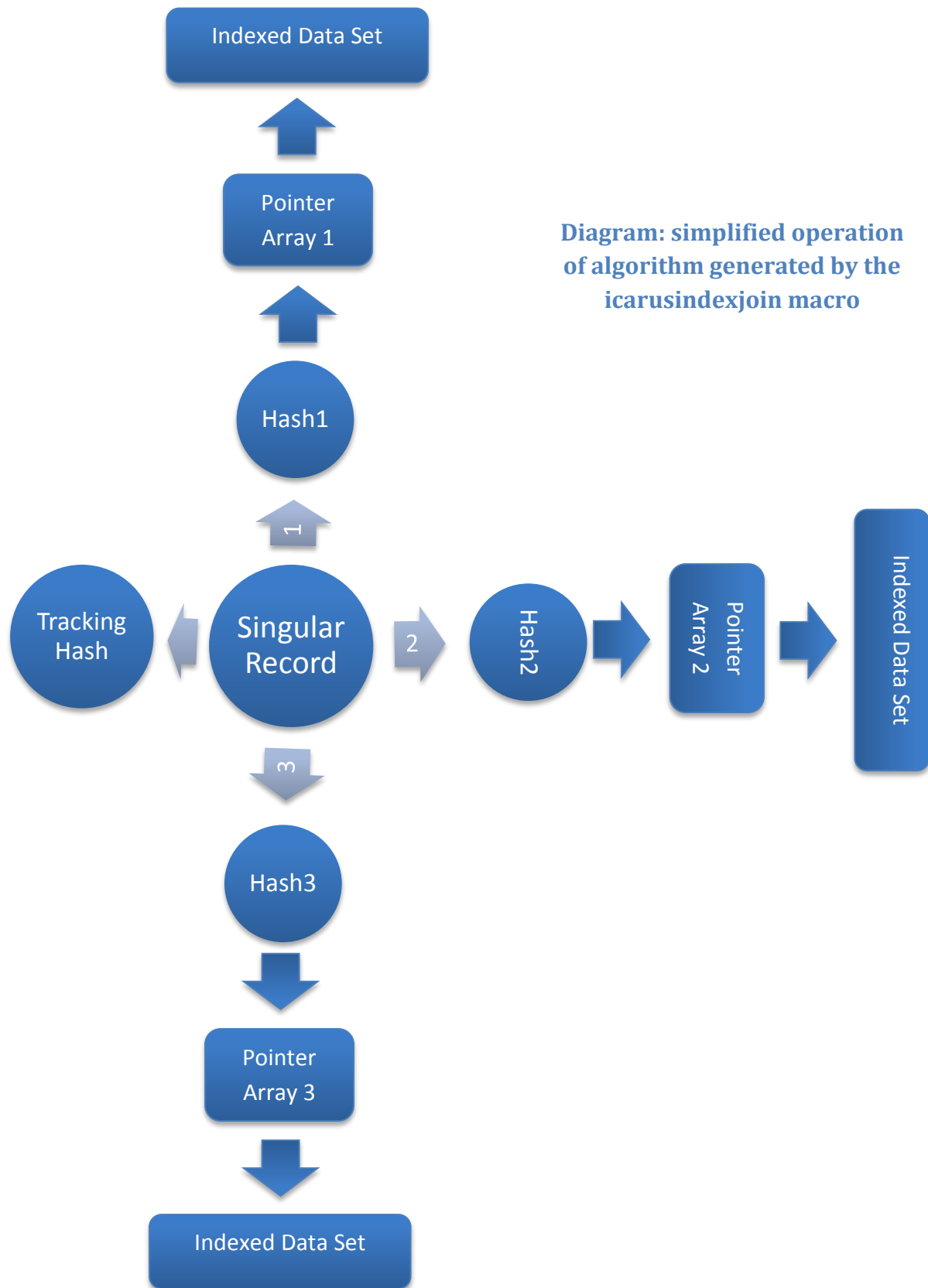
Using the earlier concepts of index pairs produced by the icarusindex and icarusindexdset macros, we can think of the first index as the hash object and pair of pointers loaded into memory, and the second index is the on-disk array of pointers for each key.

Icarusindexjoin is also responsible for implementing a tracking hash for exactly the same purpose as the tracking hashes used in the multihashjoin and multihashpointjoin macros. Like the multihashpointjoin macro, it does not need to be supplied with an ID variable, because it will use the value of the pointers themselves as unique record identifiers.

Like the other multiple equijoin macros, we will use a 3 record Control Data Set for our example.

Simple Control Data Set

work.control		
Name	City	Income
0	1	0
1	1	0
0	1	1



Although not directly illustrated, the records from the Control Data Set are used to generate keys via the same method as the `icarusindexdset` macro.

These keys are then used to generate Icarus index pairs, as per the specifications in the `icarusindex/icarusindexdset` documentation.

It is at this stage that the `icarusindexjoin` macro takes over and implements the real algorithm, similar in nature to that of `multihashjoin` and `multihashpointjoin`.

In our example, the first member of the three Icarus index pairs are loaded into the three hash objects respectively.

The second member of each Icarus index pair serves as the pointer array that accompanies each respective hash object.

`Icarusindexjoin` then begins to read records sequentially from the data set referenced in the `OtherDataSet` parameter. The entire algorithm can now be talked about as though it operated on each record individually, because at the heart of it, that is what happens as it cycles through each of the records in the other data set.

1. The singular record that has been loaded is compared against Hash 1 via the Hash 1 key. The two pointers are retrieved from Hash 1. These pointers define the subset of the pointer array that corresponds to that particular key. This subset of pointers are then loaded, and used to retrieve the resulting observations from the indexed data set. Combinations of records consisting of the singular record combined with the data retrieved via the pointers from the array are output to the data set referenced by the `outdata` parameter. Before output however, these pointers are hashed and added to the tracking hash.
2. The singular record is now compared against Hash 2 via the Hash 2 key. Two pointers are retrieved from Hash 2. These pointers are used to access the subset of the second pointer array that corresponds to the records with that particular key. However, before the pointers in this array are used, they are first checked against the pointers already stored in the tracking hash. If the pointer is not found it is added to the tracking hash. Data is retrieved using the pointer to access the indexed data set, and a new record consisting of the singular record and the retrieved record data shall be output to `outdata`. If the pointer is already found within the tracking hash, then that observation is ignored. No additional data is retrieved and no additional observation is output.

3. The singular record is compared against Hash 3 via the Hash 3 key. A similar process takes place as in step 2, but now with Hash 3.

With the singular record having been through the requisite process with all three hashes, the tracking hash is wiped clean and a new singular record from the other data set is read in. The process continues again from step one, and finishes when step 3 for the last record obtained from the other data set has been completed.

Parameters

IndexedDataSet: This parameter is a reference to the data set that is either already indexed, or which shall be indexed via the provision of a Control Data Set.

IndexedDataSetVars: This parameter determines the variables that shall be included from the indexed data set on the output data set. Variables are supplied via a space delimited list. If the user does not supply this parameter, the macro will include every variable from the indexed data set.

PrefixIndexedDataSet: A prefix that is applied to all of the variables requested to be kept from the indexed data set on the output data set.

OtherDataSet: This parameter is a reference to the other data set that will be joined with the indexed data set via the multiple equijoins signified in a Control Data Set or in pre-prepared indexes.

OtherDataSetVars: This parameter determines the variables that shall be included from the other data set in the output data set. Variables are supplied via a space delimited list of variables. If the user does not supply this parameter, the macro will include every variable in the other data set.

PrefixOtherDataSet: A prefix that is applied to all of the variables requested to be kept from the other data set on the output data set.

ControlDataSet: This parameter is a reference to an optional control data set. The control data set must be in the specific format of a control data set. If this argument is supplied, the FirstIndexNumber and LastIndexNumber will automatically be filled in based upon the dimensions of the control data set, and thus do not need to be supplied.

Index1Root: Is the root of the first index of an Icarus index pair. If a control data set parameter is supplied, this root will be used to create the indexes from the control data set. If a control data set is not supplied, this is the root of the first indexes that the algorithm will attempt to use in its operation.

Index2Root: Is the root of the second index of an Icarus index pair. If a Control Data Set parameter is supplied, this root will be used to create the second indexes from the control data set. If a control data set is not supplied, this is the root of the indexes that the algorithm will attempt to use in its operation.

FirstIndexNumber: If a Control Data Set is supplied, this parameter need not be supplied. Otherwise, it is used to access the multiple sets of indexes required by the algorithm. Assuming that the indexes are named &Index1Root.&i and &Index2Root.&i, i is incremented from FirstIndexNumber to LastIndexNumber.

LastIndexNumber: If a Control Data Set is supplied, this parameter need not be supplied. Otherwise, it is used to access the multiple sets of indexes required by the algorithm. Assuming that the indexes are named &Index1Root.&i and &Index2Root.&i, i is incremented from FirstIndexNumber to LastIndexNumber.

Outdata: The name of the output data set from the icarusindexjoin operation. If the user does not supply this parameter, it defaults to work.icarusindexjoined.

DorV: This parameter can be set to D or V, and is used to determine whether the output data set from this macro is a physical data set, or a view. Defaults to V if the user does not supply it, which is not at all surprising, because such comparison spaces and data sets are often far greater in size than any available storage medium.

ExcludeMissings: Determines whether records with missing variables in the various keys are included as candidates within the comparison space/output data set.

Exp: A parameter between 1 and 20 that is used by the SAS hash object to determine the number of buckets in a hash object. If the user does not supply this parameter, it defaults to 12.

MULTIHASHJOIN

```
%multihashjoin(  
  
  HashedDataSet = ,  
  
  HashedDataSetVars = ,  
  
  PrefixHashedDataSet = b_,  
  
  HashDataSetIDVar = ,  
  
  
  
  OtherDataSet = ,  
  
  OtherDataSetVars = ,  
  
  PrefixOtherDataSet = a_,  
  
  
  
  ControlDataSet = ,  
  
  
  
  Outdata = work.multihashjoined,  
  
  DorV = V,  
  
  ExcludeMissings = N,  
  
  Exp = 12  
);
```

Description

Multihashjoin is a macro that can be used to create an output data set that is the equivalent of the record pairs generated from multiple blocking passes/equijoin between two data sets. The technique used to determine the blocking keys/equijoin predicates is via the supply of a Control Data Set, identical to the technique used to create Icarus indexes via the icarusindexdset macro.

If the user is unfamiliar with this technique, it is advisable that they peruse the documentation of said macro before continuing.

There are two other macros that achieve similar ends supplied with Icarus (icarusindexjoin and multihashpointjoin), both choosing a different trade-off in their use of resources and complexity.

When it comes to terms of memory, multihashjoin is easily the greediest.

But because it is the greediest, it is also the simplest. If the user has no particular concerns with regards to memory use, because of the relative size of their resources versus one of their data sets, then it may also be the fastest. Additionally, because it is the simplest, it may also be a good place to start when trying to understand the more advanced algorithms used in the likes of multihashpointjoin or icarusindexjoin.

For each blocking criteria/set of equijoin predicates, multihashjoin hashes the entire target data set into memory, using the values of the keys specified in the Control Data Set. Because it duplicates the original data set multiple times, it can easily chew up resources, and so should be used with this fact in mind. But also for this reason, it can perform exceptionally well, as records defined by each key can be found and accessed extremely quickly.

Multihashjoin requires the join variables on the two data sets to be named the same.

An explanation of the multihashjoin algorithm now follows.

MultiHashJoin Theory

It is assumed that the reader has a basic understanding of computing concepts such as associative arrays and hash tables. I am not going to go into the details of how such concepts operate, nor explicitly discuss the issues of collision resolution and hash buckets which the SAS hash object largely tackles. Conceptual understanding of how the SAS data step operates is also helpful. Because of the necessary complexity of the operation, and the relative pointlessness of such operations on small data, I will be using abstract concepts rather than explicit examples to illustrate the operation of Multihashjoin.

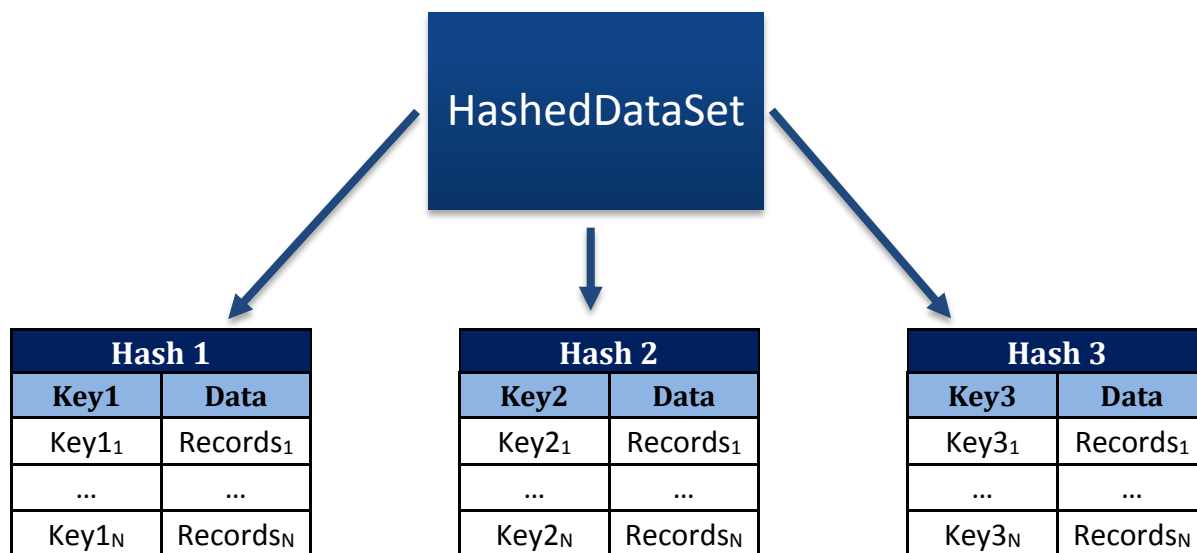
The first step in the algorithm is to obtain the definition of the keys that will be used from the Control Data Set.

For our exercise, we will assume that the Control Data Set contains three observations, which by definition corresponds to three blocking pass/equijoin key definitions. In this case, Key1 = City, Key2 = Name, City, and Key3 = City, Income.

Simple Control Data Set

work.control		
Name	City	Income
0	1	0
1	1	0
0	1	1

Copies of the records from the data set referenced in the HashedDataSet parameter are loaded into separate SAS hash objects, with each corresponding to one of the particular keys supplied in the control data set.

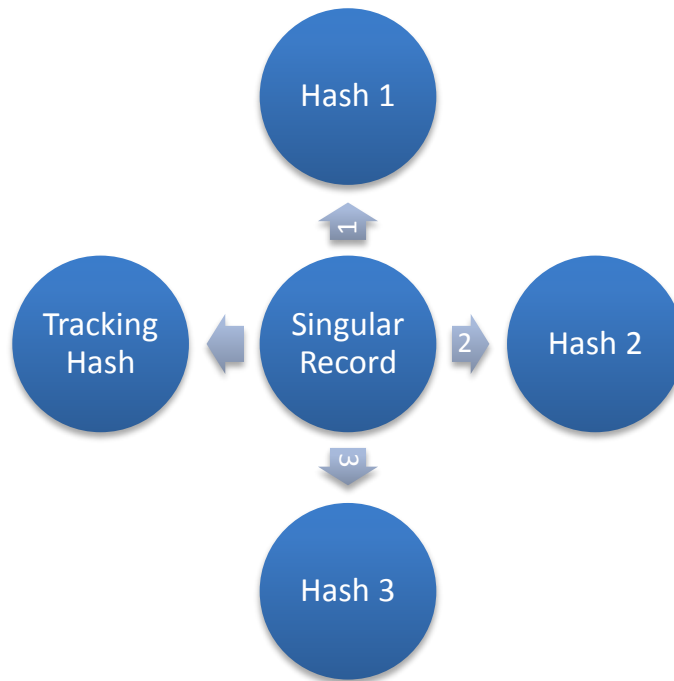


Additionally, the multihashjoin macro requires the identification of an ID variable, supplied via the HashDataSetIDVar parameter and contained in the HashedDataSet. This variable must uniquely identify each record. This variable is used as the key to an additional hash object called the “tracking hash” which will contain keys but no data items.

The tracking hash is initially empty. While the other hash objects remain static once loaded, the tracking hash will be constantly loading and deleting its contents during the operation of the algorithm.

With each of the hash objects defined, and the necessary data loaded, the algorithm now turns to the second data set, referenced by the parameter OtherDataSet.

SAS implements a data step, which reads through the other data set sequentially. We can think of it as loading one record at a time from this data set even if the specifics inside the computer are slightly more complex.



Each singular record is initially loaded from the other data set. Once in, it goes through the following process:

1. The singular record is compared against Hash 1 via the Hash 1 key. It is combined with all records that share this key, and the resulting observations consisting of the singular record combined with the data from the hash object for that key are output to the data set referenced by the outdata parameter. Before output however, the ID Variable is hashed from each record and added to the tracking hash.
2. The singular record is now compared against Hash 2 via the Hash 2 key. It is combined with all records that share this key in Hash 2. However, the resulting observations are not automatically output to outdata. The ID Variables from these observations must first be checked against the keys already stored in the tracking hash. If the ID Variable key from an observation is not found in the tracking hash, its key is added to the tracking hash, and the observation consisting of the singular record and the Hash 2 data shall be output to outdata. If the ID variable from an observation is found within the tracking hash, then that observation is ignored and not output.

3. The singular record is compared against Hash 3 via the Hash 3 key. A similar process takes place as in step 2, but now with Hash 3.
4. With the singular record having been compared to all records in all three original hashes, the tracking hash is wiped clean and a new singular record from the other data set is read in. The process continues again from step one.

When the last record from the other data set has been processed, the algorithm has now finished.

Some minor efficiencies might be gained by placing the “widest” blocking scope in the first position in the control data set. The reason for this is that it can be seen from the algorithm that no resources need to be spent during the interrogation of the first hash object on simultaneous interrogation of the tracking hash. The program knows that the tracking hash will be empty as each singular record is loaded, and so it does not need to waste time checking the tracking hash for records having been previously included.

Multihashpointjoin and icarusindexjoin are similar in philosophy to multihashjoin, but each carries a respectively greater level of complexity. To understand them, it is essential to first understand the operation of the multihashjoin macro.

Parameters

HashedDataSet: This parameter is a reference to the data set that shall be loaded into the multiple hash objects.

HashedDataSetVars: This parameter determines the variables that shall be included from the hashed data set in the output data set. Variables are supplied via a space delimited list. If the user does not supply this parameter, it will default to include every variable in the hashed data set.

PrefixHashedDataSet: A prefix that is applied to all of the variables to be kept from the hashed data set on the output data set.

HashDataSetIDVar: This parameter must be a variable on the hashed data set that uniquely identifies each record/observation on that data set. You must supply this parameter if you wish to use the MultiHashJoin macro.

OtherDataSet: This parameter is a reference to the other data set that will be joined with the hashed data set via the multiple equijoins signified in ControlDataSet.

OtherDataSetVars: This parameter determines the variables that shall be included from the other data set in the output data set. Variables are supplied via a space delimited list. If the user does not supply this parameter, it will include every variable from the other data set.

PrefixOtherDataSet: A prefix that is applied to all of the variables requested to be kept from the other data set on the output data set.

ControlDataSet: The MultiHashJoin macro requires a reference to a Control Data Set. To understand how a control data set behaves and how it must be formatted, please read the documentation to the icarusindexdset macro.

Outdata: The name of the output data set. If the user does not supply this parameter, it defaults to work.multihashjoined.

DorV: This parameter can be set to D or V, and is used to determine whether the output data set from this macro is a physical data set, or a view. Defaults to V if the user does not supply it, which is not at all surprising, because such comparison spaces are often far greater in size than any available storage medium.

ExcludeMissings: Determines whether records with missing values in the key variables are included within the comparison space/output data set.

Exp: A parameter between 1 and 20 that is used by the SAS hash object to determine the number of buckets in a hash object as a power of 2. If the user does not supply this parameter, it defaults to 12.

MULTIHASHPOINTJOIN

```
%multihashpointjoin(  
  
  HashedDataSet = ,  
  
  HashedDataSetVars = ,  
  
  PrefixHashedDataSet = b_,  
  
  
  OtherDataSet = ,  
  
  OtherDataSetVars = ,  
  
  PrefixOtherDataSet = a_,  
  
  
  ControlDataSet = ,  
  
  Indexviewroot = work.mhpjindex,  
  
  
  Outdata = work.mhpjoined,  
  
  DorV = V,  
  
  ExcludeMissings = N,  
  
  Exp = 12  
);
```

Description

Multihashpointjoin is a macro to create an output data set that is the equivalent of the record pairs generated from multiple blocking passes/equi joins between two data sets. The technique used to determine the blocking keys/equi join predicates is via the supply of a Control Data Set, identical to the technique used to create Icarus indexes via the icarusindexdset macro. If the

user is unfamiliar with this technique, it is advisable that they peruse the documentation of that macro before continuing.

There are two other macros that achieve similar ends supplied with Icarus (icarusindexjoin and multihashjoin), both choosing a different trade-off in their use of resources and complexity.

Multihashpointjoin occupies somewhat of a middle-ground between its two relative macros.

Unlike Multihashjoin, only the keys specified in the Control Data Set and pointer data (rather than the entire record) are stored in the hash table. The pointers for each key are then accessed and used to retrieve the requisite records from the original data set for the required joins to be fulfilled.

An explanation of the multihashpointjoin algorithm now follows.

Multihashpointjoin Theory

It is assumed that the reader has a basic understanding of computing concepts such as associative arrays and hash tables. I shall not go into the details of how such concepts operate, nor explicitly discuss the issues of collision resolution and hash buckets which the SAS hash object largely tackles. Conceptual understanding of how the SAS data step operates is also helpful. Because of the necessary complexity of the operation, and the relative pointlessness of such operations on small data, I will be using abstract concepts rather than explicit examples to illustrate the operation of multihashpointjoin.

If the reader is not already familiar with the operation and theory behind the multihashjoin macro, it is advised that they read the documentation of that macro before continuing. Multihashpointjoin is essentially an adaption to the operation of that algorithm, and hence I will try not to repeat myself needlessly.

The difference between the algorithms produced by the two macros has to do with how each of them store and retrieving data from the hashed data set. This in turn leads to two additional differences in the setup and operation of the two algorithms.

Rather than storing copies of the entire data set in the hash objects it creates, multihashpointjoin only stores the keys and pointer data referencing the location of the records of interest in the hashed data set. Then, rather than retrieving the records from the hash objects, multihashpointjoin retrieves the pointers and uses them to retrieve the records of interest from the data set. This results in less memory use for the storage of the hash objects, but can add latency due to the use of random read access to the hashed data set. Alternatively, and as is more practically suggested, the hashed data set may itself be loaded into memory to

reduce this latency, either via the use of the sasfile statement when working within one sas process, or alternatively by using some form of shared memory management. Ramdisk software is readily available and cheap for the windows family of operating systems, and the lucky users of later versions of Linux now have this ability practically built into the operating system itself.

Multihashjoin required the provision of a unique ID variable from the hashed data set, in order to explicitly identify each record for use in the tracking hash. Multihashpointjoin instead requires the provision of an index view root.

When Multihashpointjoin operates, it automatically creates virtual data sets defined on the hashed data set, with key and pointer data corresponding to the information that will be entered into each hash for each blocking strategy. The Indexviewroot argument is used to give these virtual data sets a name and location. These are the data sets that are loaded into the hashes.

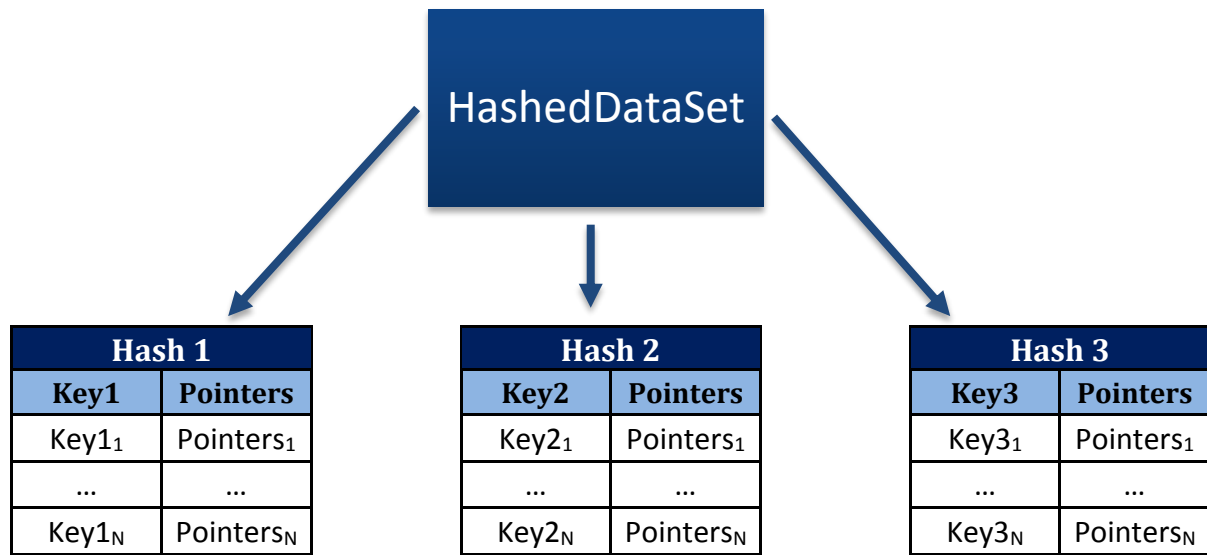
Because the macro uses pointer data, there is no need to supply an ID variable. It is impossible for each record to take up the same physical space on the data set, and so the algorithm automatically uses this pointer as the unique identifier of each record for storage in the tracking hash, rather than an ID variable.

A distinctly summarised recap of the algorithm relative to the explanation given in the documentation of the MultiHashJoin macro now follows.

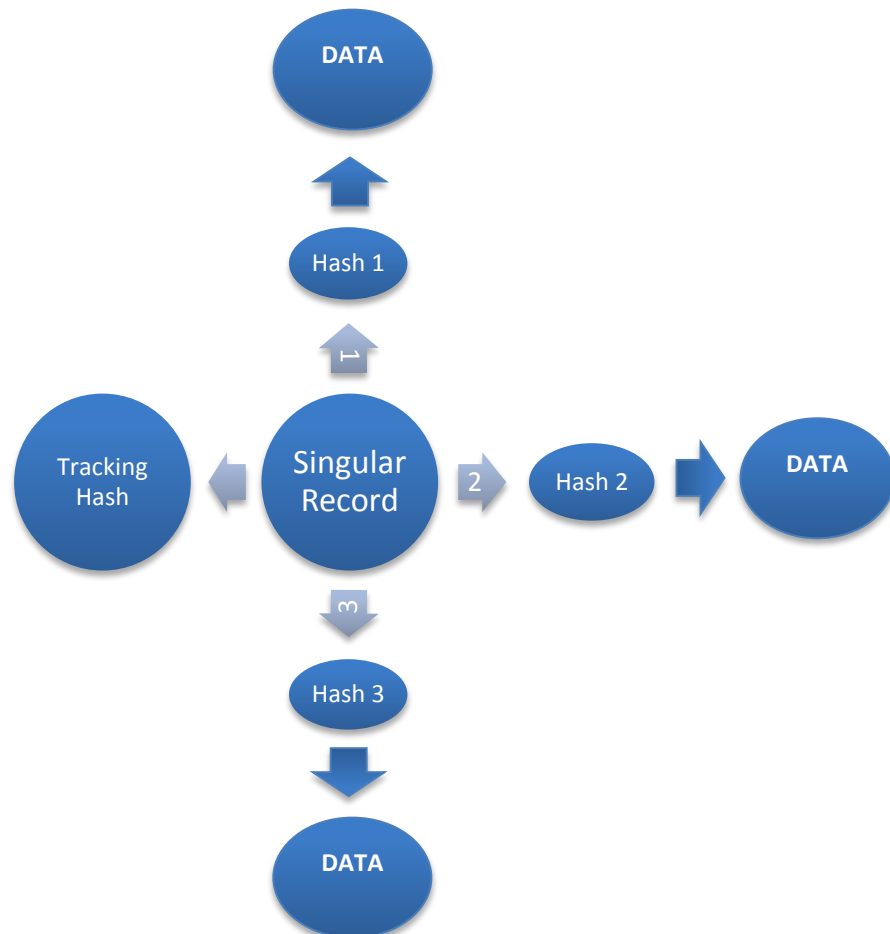
Simple Control Data Set

work.control		
Name	City	Income
0	1	0
1	1	0
0	1	1

Copies of the records from the data set referenced in the HashedDataSet parameter are loaded into separate SAS hash objects in memory, with each hash object corresponding to one of the particular keys supplied in the control data set.



The keys and pointers are read into the hash objects. Rather than obtaining the observation data from the hashes, only pointers are retrieved and used to find data in the hashed data set.



The running of the algorithm proceeds largely in a similar vein as the MultiHashJoin macro, reading in singular observations from the other data set. Only now there is the added step of using the pointers to retrieve data from the hashed data set.

1. The singular record is compared against Hash 1 via the Hash 1 key. The pointers are retrieved from the hash and then used to retrieve the resulting observations from the hashed data set. Combinations of records consisting of the singular record combined with the data retrieved via the pointers are output to the data set referenced by the outdata parameter. Before output however, the pointer is hashed from each record and added to the tracking hash.
2. The singular record is now compared against Hash 2 via the Hash 2 key. The pointer is retrieved. However, the resulting observations from following those pointers are not automatically retrieved from the hashed data set at this stage. First the pointers retrieved from Hash 2 must be checked against the pointers already stored in the tracking hash. If the pointer from an observation is not found in the tracking hash, its key is added to the tracking hash. The data is retrieved using the pointer from the hashed data set, and a new record consisting of the singular record and the retrieved data shall be output to outdata. If the pointer from an observation is already found within the tracking hash, then that observation is ignored. No data is retrieved and no observation is output.
3. The singular record is compared against Hash 3 via the Hash 3 key. A similar process takes place as in step 2, but now with Hash 3.

With the singular record having been compared to all keys in all three original hashes, the tracking hash is wiped clean and a new singular record from the other data set is read in. The process continues again from step one, and finishes when step 3 for the last record obtained from the other data set has been completed.

Parameters

HashedDataSet: This parameter is a reference to the data set from which the keys and pointers will be loaded into multiple hash objects. It is also necessarily the data set from which the eventual data referenced in this hash objects shall be sourced.

HashedDataSetVars: This parameter determines the variables that shall be included from the hashed data set in the output data set. Variables are supplied via a space delimited list of variables. If the user does not supply this parameter, it will default to include every variable in the hashed data set.

PrefixHashedDataSet: A prefix that is applied to all of the variables requested to be kept from the hashed data set on the output data set.

OtherDataSet: This parameter is a reference to the other data set that will be joined with the hashed data set via the multiple equijoins signified in the control data set.

OtherDataSetVars: This parameter determines the variables that shall be included from the other data set in the output data set. Variables are supplied via a space delimited list. If the user does not supply this parameter, the macro will include every variable in the other data set.

PrefixOtherDataSet: A prefix that is applied to all of the variables requested to be kept from the other data set on the output data set.

ControlDataSet: The multihashpointjoin macro requires the provision of a Control Data Set reference. To understand how a Control Data Set behaves and how it must be formatted, please read the documentation to the icarusindexdset macro.

Indexviewroot: The multihashpointjoin macro creates data step views with the requisite key values and pointer information for loading into the various hash objects. This parameter determines where/what those data step views will be called, by following the pattern &Indexviewroot.&i for each data step view required.

Outdata: The name of the main output data set from the multihashpointjoin operation. If the user does not supply this parameter, it defaults to work.mhpjoined.

DorV: This parameter can be set to D or V, and is used to determine whether the output data set is a physical data set or a view. Defaults to V if the user does not supply it, which is not at all surprising, because such comparison spaces are often far greater in size than any available storage medium.

ExcludeMissings: Determines whether records with missing variables in the various keys are included as candidates within the comparison space/output data set.

Exp: A parameter between 1 and 20 inclusive that is used by the SAS hash object to determine the number of buckets in a hash object as a power of 2. If the user does not supply this parameter, it defaults to 12.

SNHOOD

```
%snhood(  
  
DataSet = ,  
  
SortVar = ,  
  
Order = ,  
  
Outdata = work.SNHood,  
  
VorD = V,  
  
Window = ,  
  
Forevar = ,  
  
Aftvar = ,  
  
PrefixA = a_,  
  
PrefixB = b_,  
  
Denialvar = ,  
  
Rollover = N,  
  
Tagsort = N  
);
```

Description

Snhood implements an alternative method of blocking known as sorted neighbourhood.

The general idea is to sort a data set and slide a metaphorical window of a specific size from the top of the data set down to the bottom. As the window moves, records within the window are compared to each other.

I have not implemented this technique based on any other person's particular interpretation or specification. This was due to being unable to find any generally accepted answers to certain problems I encountered when thinking about the implications of implementing such a method.

As is my general tendency when tackled with a dearth of information and lack of apparent acknowledgement of such issues in much of the literature, I said “to hell with it” and made up my own.

Thus, this documentation should be taken as the definitive guide of how the Icarus implementation of sorted neighbourhood works, and it should not be confused with any alternate implementations or techniques bearing this name, whether they may differ or be similar in any shape or form.

Sorted Neighbourhood Theory

The main premise behind the idea of a sorted neighbourhood operation first involves a sorted data set and an intellectual object we will call the “window”. To illustrate the application of the sorted neighbourhood, I will use the following data set:

Example data set for sorted neighbourhood theory

work.example			
ID_Var	Name	City	Income
6	Carlos	Brisbane	183
5	Sophie	Chicago	456
3	Xiu	Honolulu	785
8	Akiko	Chicago	1000
7	Marcel	Chicago	1212
2	Enoch	Brisbane	1500
1	Charlie	Honolulu	1900

Window of size
N=2 centred on
Xiu

Window of size
N=1 centred on
Enoch

Window of size
N=3 centred on
Akiko

In this example the data set has been sorted, via an ascending sequence, on the Income variable.

The concept of a window is illustrated in the above diagram with windows centred on three different records, and with three different dimensions.

The window where $N=1$ is trivial, since only a singular record can reside in such a window, and so there are no other records we can compare it to. So we will begin our discussion by tackling the notion of a stationary window where $N=2$, centred on Xiu. There are three records that reside within this window: Sophie, Xiu, and Akiko. Possible combinations within this stationary

window are therefore: (Sophie, Xiu) and (Xiu, Akiko). I do not make any claims as to whether Xiu must take up the first position or the second position in any such comparisons.

Note that Sophie and Akiko are not possible combinations. If we moved a window of the same dimensions up and centred on Sophie, or moved down and centred on Akiko, then Akiko would not be within Sophie's window, and Sophie would not be in Akiko's.

If we now slide this window down one spot, so that it was centred on Akiko, we would have two possible comparisons for Akiko: (Xiu, Akiko) and (Akiko, Marcel). However, we have already compared Xiu and Akiko when the window was centred on Xiu, and so we do not need to duplicate the comparison again.

Sliding the window down like this, we output observations to a data set, such that each observation is of the form (Person_a, Person_b), and all necessary combinations of persons within a window of a specified dimension will exist, but we make no claims as to whether a particular record will hold the position of Person_a or Person_b, only that the requisite comparisons will be contained in the output data set. This is the basic idea of a sorted neighbourhood: sliding a window from the first record to the last record, and making comparisons between records that fit within the window.

The astute almost instantly see potential issues with this algorithm. What happens when we window is on either Carlos or Charlie? In the case of Carlos, the metaphorical "head" of the window "dangles"⁵ over non-existing records, and in the case of Charlie, it is the "tail" that dangles. In both cases, there is a distinct lack of records to compare them to. None of the potential solutions create a universally beautiful solution.

Dangling Windows of N=2

Dangling Head	6	Carlos	Brisbane	183
	5	Sophie	Chicago	456
	3	Xiu	Honolulu	785
	8	Akiko	Chicago	1000
	7	Marcel	Chicago	1212
	2	Enoch	Brisbane	1500
	1	Charlie	Honolulu	1900

Dangling Tail

In the Icarus implementation, there are potential ways of dealing with this issue.

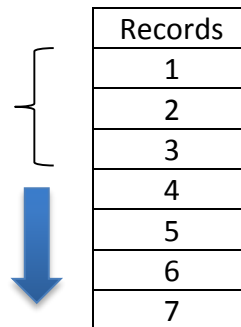
⁵ "Dangle" of course being a highly technical term...

The first option is to accept that the number of comparisons between records at the beginning and end of a sorted data set will be fewer than those in the middle. In this case, Carlos will only be compared with Sophie, and Charlie will only be compared with Enoch, whereas all other records will be compared to two other records.

A second option is to set the rollover parameter to “Y”. There are a number of ways to visualise or think about how this option actually operates. When the rollover parameter is set to “Y”, we can think of the data set as no longer constituting a “linear array” of records where the window moves from top to bottom. It now constitutes a “circular array” where the window rotates around the circle.

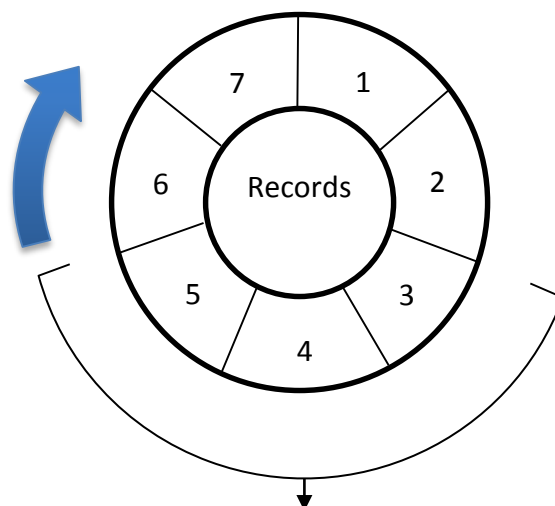
Linear Array

Window of
N=2 moves
down the
array



Records
1
2
3
4
5
6
7

Circular Array



Window of N=2 rotates around the array

Now the data set has no physical beginning or end, and although we start our window centred on record 1 and rotate it around until it centres on record 7, the effective window size has maintained its constant nature and each record is compared with two other records. Of course, record 1 might not have very much in common with record 7, but that issue is for the user to think about when deciding upon how to call the `snhood` macro and their particular domain of application.

There is another issue to consider when selecting the rollover option. The algorithm written by the non-rollover option ensures that there is no duplicate comparisons by ensuring that Sophie is not re-compared with Carlos after the window has moved from Carlos to Sophie. The roll-over option does not ensure this behaviour. This could either be desired or undesired. With the rollover option, it can be assured that there will be a constant number of records for each observation that takes up the `person_a` slot of a (`person_a`, `person_b`) record pair, and that each record that has had the window centred on them will take up a requisite position in the `person_a` slot. However, several records will be logical duplicates, in the sense that (Carlos, Sophie) is logically the same as (Sophie, Carlos), yet both will be found on the output data set.

Rollover has one more feature that I am not going to illustrate (for reasons that will soon become clear), but which I will attempt to describe. With the roll-over option set, we can start to play some multi-dimensional games by setting the window size greater than the actual data set. Now an observation can be compared to itself, because if you walk around a globe for a distance that is further than the circumference, you eventually arrive back where you started. It is a peculiar behaviour that the user is free to experiment with for their own nefarious ends.

A sliding window of fixed dimension is all very well and good, but it is also incredibly boring and not necessarily of optimal flexibility. Independent, orderly and uniformly distributed variables are the exception rather than the rule.

The user has the option of appending two numeric variables containing integers to the original data set that I will call the “Forevar” and “Aftvar”. When the window centres on a particular record, it looks at these two variables and says that the window for that record begins `forevar` records before the current record, and `aftvar` records after the current variable. When using these options, the data set can be thought of as a circular array, meaning one record before the first record on the data set is the last record on the data set, and the record one record after the last record on the data is the first record on the data set. The actual behaviour will be dependent on what numbers the user has entered into the `forevar` and `aftvar` variables for each observation. Indeed, by entering various numbers into the respective `forevar` and `aftvar` for each variable, any of the algorithms and behaviours mentioned so far in this documentation are fully reproducible, and in addition, a new dynamic-window-size behaviour may be induced.

The relationship between the forevar and aftvar dimensions, and the dimensions used to describe the size of the default static window is such that:

$$Window(N) = Forevar(N - 1) \text{ AND } Aftvar(N - 1)$$

One last theoretical concept in the snhood macro is the application of one additional variable: the denialvar.

The user has the option of adding a denial variable to the data set to be targeted, and then specifying it in the denialvar parameter. When the denialvar is set, the algorithm looks at the denialvar values between any two record comparisons that are achieved using the sorted neighbourhood macro. If the denialvar is the same for two records, that comparison is not output to the final data set. This property can be extremely helpful in allowing the user to run an algorithm that only compares records across two data sets (if they have been merged into one continuous data set and their original data set is identified in the denialvar), denying records from being compared to themselves (by including a unique ID in the denialvar), or any other criteria or phenomenon that can be expressed in this way.

Because the snhood macro requires the data set on which it is working to be sorted, the user should realise that it modifies (via sorting) the data that is referenced in the DataSet parameter.

Parameters

DataSet: The reference to the data set that will have the sorted neighbourhood method applied to it.

SortVar: The variables, supplied in a space delimited list, which will be used to sort the data set.

Order: An array of space delimited letters, consisting of values A or D. Must have the same number of elements as variables supplied in the SortVar parameter. Each item determines whether the SortVar in the same corresponding position is sorted in ascending or descending order.

Outdata: The name of the data set that shall be output from the operation of the snhood macro. If the user does not supply this parameter, it defaults to work.snhood.

Word: This parameter can take on the value V or D, determining whether the output data is in the form of a physical data set or a data step view.

Window: This parameter can be supplied to create a fixed dimension symmetrical window that moves through the sorted data set in accordance with sorted neighbourhood theory. The number supplied is the dimension of the window, as referenced in the documentation. You cannot supply the window parameter while simultaneously using the forevar and aftvar parameters.

Forevar: A parameter that names a variable on the sorted data set which contains integer values. Each value is used to determine how many previous records on the data set will be included in the window once the window has centred on that record. The forevar and the aftvar parameters must be supplied together, and neither can be supplied with the window parameter.

Aftvar: A parameter that names a variable on the sorted data set which contains integer values. Each value is used to determine how many following records on the data set will be included in the window once the window has centred on that record. The forevar and the aftvar parameters must be supplied together, and neither can be supplied with the window parameter.

PrefixA: A prefix that is appended to the beginning of variables on the output data set so that a singular record can meaningfully be brought together with other records from the same data set in one observation. If the user does not supply this argument, it defaults to a_.

PrefixB: A prefix that is appended to the beginning of variables on the output data set so that a singular record can meaningfully be brought together with other records from the same data set in one observation. If the user does not supply this argument, it defaults to b_.

Denialvar: An optional parameter that references a variable on the input data set. Record comparisons that share the same denialvar value will not be included on the output data set.

Rollover: A parameter that can take the value of Y or N which determines whether the data set is treated as a circular array as per the documentation. If the user supplied the forevar and aftvar parameters instead of the window parameter, this value will automatically take the value of Y. Otherwise, if the user does not supply this parameter, it will default to the value of N.

Tagsort: This parameter can take the value of N or Y. It defaults to N if not supplied. If set to Y, then the original data set is sorted using the tagsort option in PROC SORT. The tagsort option is particularly useful to limit resource use when sorting large, wide data sets. However, it is generally not advisable to use this option if the resources to sort the data set without it are available.

Example 1: A sorted neighbourhood on the work.example data set when sorted by Income (ascending) with a window of type N=2.

Example data set

work.example			
ID_Var	Name	City	Income
6	Carlos	Brisbane	183
5	Sophie	Chicago	456
3	Xiu	Honolulu	785
8	Akiko	Chicago	1000
7	Marcel	Chicago	1212
2	Enoch	Brisbane	1500
1	Charlie	Honolulu	1900

```
%snhood(
DataSet = work.example,
SortVar = Income,
Order = A,
Window = 2
);
```



work.snhood							
a_ID_Var	a_Name	a_City	a_Income	b_ID_Var	b_Name	b_City	b_Income
6	Carlos	Brisbane	183	5	Sophie	Chicago	456
5	Sophie	Chicago	456	3	Xiu	Honolulu	785
3	Xiu	Honolulu	785	8	Akiko	Chicago	1000
8	Akiko	Chicago	1000	7	Marcel	Chicago	1212
7	Marcel	Chicago	1212	2	Enoch	Brisbane	1500
2	Enoch	Brisbane	1500	1	Charlie	Honolulu	1900

Example 2: Perform a sorted neighbourhood on the work.example data set when sorted by Income (ascending). Rather than use a symmetrical window, apply a forevar named “f” and an aftvar named “a” to the original data set, and use those to set the dimensions of the window instead. Supply explicit prefixes.

Example data set with forevars and aftvars added

work.example					
ID_Var	Name	City	Income	f	a
6	Carlos	Brisbane	183	2	1
5	Sophie	Chicago	456	2	1
3	Xiu	Honolulu	785	2	1
8	Akiko	Chicago	1000	2	1
7	Marcel	Chicago	1212	2	1
2	Enoch	Brisbane	1500	2	1
1	Charlie	Honolulu	1900	2	1

```
%snhood(
DataSet = work.newexample,
SortVar = Income,
Order = A,
forevar=f,
aftvar=a,
prefixA = ,prefixB = b_);
```



work.snhood											
ID_Var	Name	City	Income	f	a	b_ID_Var	b_Name	b_City	b_Income	b_f	b_a
6	Carlos	Brisbane	183	2	1	2	Enoch	Brisbane	1500	2	1
6	Carlos	Brisbane	183	2	1	1	Charlie	Honolulu	1900	2	1
6	Carlos	Brisbane	183	2	1	5	Sophie	Chicago	456	2	1
5	Sophie	Chicago	456	2	1	1	Charlie	Honolulu	1900	2	1
5	Sophie	Chicago	456	2	1	6	Carlos	Brisbane	183	2	1
5	Sophie	Chicago	456	2	1	3	Xiu	Honolulu	785	2	1
3	Xiu	Honolulu	785	2	1	6	Carlos	Brisbane	183	2	1
3	Xiu	Honolulu	785	2	1	5	Sophie	Chicago	456	2	1
3	Xiu	Honolulu	785	2	1	8	Akiko	Chicago	1000	2	1
8	Akiko	Chicago	1000	2	1	5	Sophie	Chicago	456	2	1
8	Akiko	Chicago	1000	2	1	3	Xiu	Honolulu	785	2	1
8	Akiko	Chicago	1000	2	1	7	Marcel	Chicago	1212	2	1
7	Marcel	Chicago	1212	2	1	3	Xiu	Honolulu	785	2	1
7	Marcel	Chicago	1212	2	1	8	Akiko	Chicago	1000	2	1
7	Marcel	Chicago	1212	2	1	2	Enoch	Brisbane	1500	2	1
2	Enoch	Brisbane	1500	2	1	8	Akiko	Chicago	1000	2	1
2	Enoch	Brisbane	1500	2	1	7	Marcel	Chicago	1212	2	1
2	Enoch	Brisbane	1500	2	1	1	Charlie	Honolulu	1900	2	1
1	Charlie	Honolulu	1900	2	1	7	Marcel	Chicago	1212	2	1
1	Charlie	Honolulu	1900	2	1	2	Enoch	Brisbane	1500	2	1
1	Charlie	Honolulu	1900	2	1	6	Carlos	Brisbane	183	2	1

Example 3: A default sorted neighbourhood on work.example. Sorted by Income. Window of N=2. Use denialvar. Explicit prefixes.

Example data set with a denialvar added

work.example				
ID_Var	Name	City	Income	Denial
6	Carlos	Brisbane	183	1
5	Sophie	Chicago	456	1
3	Xiu	Honolulu	785	0
8	Akiko	Chicago	1000	1
7	Marcel	Chicago	1212	0
2	Enoch	Brisbane	1500	0
1	Charlie	Honolulu	1900	1

```
%snhood(
DataSet = work.example,
SortVar = Income,
Order = A, Window = 2,
prefixA = ,prefixB = b_,
DenialVar = Denial
);
```



work.snhood									
ID_Var	Name	City	Income	Denial	b_ID_Var	b_Name	b_City	b_Income	b_Denial
5	Sophie	Chicago	456	1	3	Xiu	Honolulu	785	0
3	Xiu	Honolulu	785	0	8	Akiko	Chicago	1000	1
8	Akiko	Chicago	1000	1	7	Marcel	Chicago	1212	0
2	Enoch	Brisbane	1500	0	1	Charlie	Honolulu	1900	1

PLAINBLOCKING

```
%Plainblocking(  
  
DataSetA = ,  
  
DataSetB = ,  
  
  
BlockVarsA = ,  
BlockVarsB = ,  
  
VarsA = ,  
VarsB = ,  
  
  
Prefixa = a_,  
Prefixb = b_,  
  
Behaviour = 1,  
  
Exp = 12,  
  
Outdata = work.blocked,  
  
DorV = V,  
  
Excludemissings = Y  
);
```

Description

The plainblocking macro implements the simplest forms of blocking (or lack thereof) in the data linking world. In database parlance, it is equivalent to an equijoin between the two tables/data sets using the variables provided in BlockVarsA and BlockVarsB. In more common speech, it brings together those records which share the same values found in the blocking variables across two data sets.

Behind the scenes, the plainblocking macro uses the algorithms produced either by the hashjoin macro or by the SQL procedure. Blocking is essentially the equivalent of performing an inner join in both hashjoin and SQL terminology. An explicit choice between hashjoin and SQL options can be supplied by the user in the behaviour parameter, with 1 representing the hashjoin macro, and 2 representing the use of the SQL procedure. Hashjoin and SQL will then use their own methods to choose an algorithm to produce the tables specified.

There is one exception to this behaviour, which takes place when the user does not provide the BlockVarsA and BlockVarsB parameters. In this instance, the program will produce the Cartesian product, whereby every record in DataSetA is compared with every record from DataSetB. In doing so, plainblocking will use the algorithm defined by the SQL procedure irrespective of any setting supplied in the behaviour parameter. The ExcludeMissings parameter also becomes meaningless in such a situation.

Parameters

DataSetA: A reference to the first data set.

DataSetB: A reference to the second data set.

BlockVarsA: The variables in the first data set, supplied via a space delimited list, which will be used as the blocking variables. The variables should be of similar number, length and type as those supplied in BlockVarsB. If the user wishes to create a Cartesian product, they should not supply BlockVarsA or BlockVarsB parameters.

BlockVarsB: The variables in the second data set, supplied via a space delimited list, which will be used as the blocking variables. The variables must be of similar number, length and type as those supplied in BlockVarsA. If the user wishes to create a Cartesian product, they should not supply BlockVarsA or BlockVarsB parameters.

VarsA: The variables from the first data set, supplied via a space delimited list, which will be returned in the output data set. If the user does not supply this parameter, then the macro will return all of the variables from the first data set.

VarsB: The variables from the second data set, supplied via a space delimited list, which will be returned in the output data set. If the user does not supply this parameter, then the macro will return all of the variables from the second data set.

Prefixa: A prefix that is applied to the variables from the first data set on the output data set. Defaults to a_ if the user does not supply this parameter.

Prefixb: A prefix that is applied to the variables from the first data set on the output data set. Defaults to b_ if the user does not supply this parameter.

Behaviour: Accepts either the value of 1 or 2. Allows the user to determine whether plainblocking uses the hashjoin macro or the SQL procedure in the production of results. If not supplied, it defaults to 1. If the macro is being used to create a Cartesian product, then the macro will use the SQL procedure regardless.

Exp: A number between 1 and 20 inclusive. Used by the hashjoin macro, if it is the algorithm chosen by the behaviour parameter, to indicate how many hash buckets to be used by the hash object as a power of 2. If not supplied defaults to 12.

Outdata: The name of the output data set or view created from the resulting comparison of records. If not supplied, defaults to work.blocked.

DorV: Can be set to either D or V. Determines whether the output data is in the form of a data set or a view. If not supplied by the user, it defaults to V.

ExcludeMissings: Can be set to either Y or N, and determines whether records with missing values in the blocking variables are considered to be valid comparisons. Defaults to Y if the user does not supply this parameter. Has no effect if the user does not supply any blocking variables and thus creates the Cartesian product.

**Example: Perform plain blocking on two data sets, blocking on mob.
Exclude missing keys from comparison space.**

Example datasets

work.example1			
ID_VAR	NAME	MOB	YOB
1	Charlie	7	1945
2	Enoch	6	1932
3	Xiu	7	1973
4	Carlos	3	1982
5	Sophie	1	1991
7	Marcel	.	1983
8	Akiko	2	1991

work.example2			
ID_VAR	NAME	MOB	YOB
1	Janice	8	1973
2	Jesse	7	1991
3	Roger	.	1965
4	Ella	1	1918
5	Frank	1	1991
7	Sarah	.	.
8	Yan	12	1982


```
%plainblocking(
DataSetA=work.example1,
DataSetB=work.example2,
BlockVarsA=mob,BlockVarsB=mob,
Excludemissings=Y
);
```



work.blocked							
b_ID_VAR	b_NAME	b_MOB	b_YOB	a_ID_VAR	a_NAME	a_MOB	a_YOB
2	Jesse	7	1991	1	Charlie	7	1945
2	Jesse	7	1991	3	Xiu	7	1973
4	Ella	1	1918	5	Sophie	1	1991
5	Frank	1	1991	5	Sophie	1	1991

Icarus Data Linking Macros

The programs documented in this section constitute the remaining macros in Icarus I have been unable or unwilling to categorise into other specific and specialised categories.

There is hence a greater variation in what they do and how they do it compared to the macros, functions and programs that appear in other sections.

Although originally designed to tackle problems and implement techniques for data linking, many can be used for purposes far outside their original scope.

Some are simple and straight forward. Others are considerable pieces of work in their own right.

APDERIVE

```
%apderive(  
  
DataSet = ,  
  
LinkVarsA = ,  
  
LinkVarsB = ,  
  
OutVars = ,  
  
Outdata =work.AgreementPattern,  
  
AdditionalKeepVars =,  
  
DorV = V,  
  
Case = 3,  
  
Comptypes =,  
  
Compvals =  
  
);
```

Description

The blocking macros in Icarus let the user operate on the comparison space between two data sets as though they were physical data sets. Compared to many other data linking programs, this is itself an incredibly powerful and flexible option.

But there are also a set of relatively common comparisons and conditions that can allow one to convert these blocked data sets into an output of simple agreement patterns, and barring specialised skills and abilities in data linking, it is unlikely that the user will want to take advantage of the additional power offered by further custom programming.

The production of agreement patterns could be achieved by using a custom written SAS data step or PROC SQL on the comparison space created via the blocking macros, but it will almost universally be preferable to use the automation features of the apderive macro if the agreement patterns are relatively uncomplicated and common.

Additionally, the program written by apderive has been designed to be generally efficient. A program written by the user to produce the same data set and which would be equally efficient is likely to be relatively irritating to produce.

The apderive macro's output may additionally be summarised and fed into the like of Icarus_EM to produce M and U probabilities for the general application of probabilistic data linking techniques.

Apderive Theory

The apderive macro is typically used to convert comparison spaces into common forms of agreement patterns.

More technically, it compares pairs of variables within an observation, and creates a new variable based upon the result of this comparison. Typically the input variables are not what we are interested in, and so they are not (by default) kept on the output observation. Only the variable that holds the result of their comparison is kept.

As Icarus already has a set of programs designed to bring together records from disparate data sets so that each record pair now constitutes one observation, the apderive macro lends itself to operation on such a comparison space. Since the output variable derived from the comparison of two input variables is typically a value of 0, 1 or missing, and because the code generated is generally efficient at arriving at the requested result, it is well suited to producing common arrays of agreement patterns from an input data set representing a comparison space without very much work on the user's behalf.

While all parameters fed to the macro are important, it is the relationship between five of these parameters that are the most fundamental in terms of producing the desired result. These parameters are:

LinkVarsA = $a_1 a_2 \dots a_n$

LinkVarsB = $b_1 b_2 \dots b_n$

OutVars = $c_1 c_2 \dots c_n$

Comptypes = $d_1 d_2 \dots d_n$

Compvals = $e_1 e_2 \dots e_n$

Each of these parameters consist of space delimited lists, and each must possess the same number of elements.

LinkVarsA, LinkVarsB, and OutVars must all be lists of variable names, and intellectually the macro goes through these three lists and says:

Apply a comparison function to variable \mathbf{a}_n and variable \mathbf{b}_n and place the result in variable \mathbf{c}_n

The exact nature of the comparison function is decided via the application of the elements in the parameter Comptypes. The elements in Compvals are optionally used by the function described in Comptypes to provide an additional argument. Not all comparison functions will meaningfully use the additional argument passed to them, but if they do not use it, they still require one to be present in the Compvals list. I have stylistically taken to placing the value of 0 into the Compvals list when the comparison function specified in Comptypes will not use the additional parameter passed to it.

We can abstractly think of the entire operation, written in C-like syntax as:

$$c_n = d_n(\text{value from } a_n, \text{value from } b_n, e_n)$$

Where:

\mathbf{c}_n is the name of the output variable

\mathbf{d}_n is the identifier of the kind of function/operation applied to the values derived from \mathbf{a}_n , \mathbf{b}_n , and \mathbf{e}_n

\mathbf{a}_n is the variable from which the value of the first argument to the function shall be passed

\mathbf{b}_n is the variable from which the value of the second argument to the function shall be passed

\mathbf{e}_n is the value of the third argument that shall be passed to the function

The macro will produce code for each operation designated by the combined elements found in these five parameters.

There are a set of predetermined symbols that can be used as elements in the Comptypes list. A description of them and their respective behaviour follows. I refer to them as “operations” in the descriptions to distinguish them from the functions that share the same names in Icarus and SAS and which half of them also use internally.

In addition to the general descriptions of each option, additional universal behaviour is determined by the value of the Case parameter. The Case parameter can take the value of 2 or 3, which determines how each of the operations treat missing values in the variables supplied to them. If Case is set to 2, then any missing value supplied in argument one or argument two of an operation will result in a returned value of 0. If Case is set to 3, then any missing value supplied to one of the operations results in a returned value of missing. For those wondering, the Case parameter takes the value of 2 or 3 because it results in outputting agreement patterns where variables can take either binary or ternary states as have classically been used in data linking practice.

Equality

Symbol: **E**

The equality operation returns a value of 1 when its first two arguments are equal, and a value of 0 when they are not.

It ignores the value of any third argument.

Winkler

Symbol: **WI**

The winkler operation uses the winkler function to compare its first two arguments. It uses the default option of 0.1 for the weight required by the original winkler function. This results in a floating point number between 0 and 1. This number is then compared to the third argument. If it is less than the third argument, then the winkler operation returns a value of 0. If it is equal to or greater than the third argument, it returns a value of 1.

Jaro

Symbol: **JA**

The jaro operation uses the jaro function to compare its first two arguments. This results in a floating point number between 0 and 1. This number is then compared to the third argument. If it is less than the third argument, then the winkler operation returns a value of 0. If it is equal to or greater than the third argument, it returns a value of 1.

Highfuzz

Symbol: **HF**

The highfuzz operation uses the highfuzz function to compare the first argument to the second argument. The third argument can be represented by the symbol N. If the second argument is between 0 and N larger than the first argument, the highfuzz operation returns a value of 1. Otherwise, it returns a value of 0.

Genfuzz

Symbol: **GF**

The genfuzz operation uses the genfuzz function to compare the first argument to the second argument. The third argument can be represented by the symbol N. If the second argument is within N units of the first argument, the genfuzz operation returns a value of 1. Otherwise, it returns a value of 0.

Lowfuzz

Symbol: **LF**

The lowfuzz operation uses the lowfuzz function to compare the first argument to the second argument. The third argument can be represented by the symbol N. If the second argument is between 0 and N less than the first argument, the lowfuzz operation returns a value of 1. Otherwise, it returns a value of 0.

Complex/Levenshtein spelling distance

Symbol: **CL**

The complex operation uses the SAS complex function to generate the levenshtein spelling distance between argument one and argument two. If this distance is less than the third argument, then the complex operation returns a value of 1. If it is greater than or equal to the third argument, then it returns a value of 0.

Parameters

DataSet: The parameter is a reference to the input data set. Typically, it holds the observations typically representing the record pairs.

LinkVarsA: This parameter is a space delimited list of variables on the input data set that will be compared against the variables supplied in LinkVarsB.

LinkVarsB: This parameter is a space delimited list of variables on the input data set that will be compared against the variables supplied in LinkVarsA.

OutVars: This parameter is a space delimited list of variables. It defines the variables that will hold the result of comparisons between LinkVarsA and LinkVarsB variables. By default, these are the only variables that are kept on the output data set.

Outdata: The name of the output dataset.

AdditionalKeepVars: By default the only variables that are kept on the output data set are those that are listed in the OutVars parameter. However, there may be instances where the user wishes to perform some additional processing, or otherwise keep some of the original variables from the original dataset. This parameter allows the user to supply a list of variables in a space delimited list that will also be kept on the output data set, irrespective of whether there was any processing performed using them.

DorV: This parameter can take on the value D or V and dictates whether the output data set will be a physical data set or a data step view. If the user does not supply this parameter, it defaults to V.

Case: This parameter determines the treatment of missing values. It can be set either to 2 or 3, and by default it is set to the value 3. If it is set to 2, then in the instance of a missing value for one of the LinkVarsA or LinkVarsB variables, the respective OutVar is set to 0, irrespective of whether it is one or both variables that are missing. If it is set to 3, then rather than setting missing records to 0, it will output a missing value. Produced code is also optimised relative to the requested case option.

Comptypes: Is a string of space delimited symbols. Valid symbols are listed in the documentation of the apderive theory section. Each symbol determines the operation that is conducted on the variables in LinkVarsA and LinkVarsB to produce a value placed into the respective OutVar variable.

Compvals: Is a string of space delimited numbers, fed to the respective Comptypes function/operation.

Example: Create a data set view of ternary agreement patterns from the data set work.apderiveexample.

Example Data Set

work.apderiveexample							
A_Month	B_Month	A_Name	B_Name	A_Age	B_Age	A_Day	B_Day
7	7	David	Davey	45	45	10	10
7	8	David	David	45	44	10	11
7	6	David	Dave	45	46	10	.
.	.	David	Roger	45	.	10	10

Agreement will be represented by the value 1. Disagreement will be represented by the value 0. If any variable is missing, the comparison will be coded as missing.

A_Month is considered to agree with B_Month if B_Month is up to 1 less than A_Month.

A_Name is considered to agree with B_Name if the winkler score from a comparison between the two variables is equal to or greater than 0.85.

A_Age is considered to agree with B_Age if B_Age is up to 1 more than A_Age.

A_Day is considered to agree with B_Day only if the two fields agree exactly.

```
%apderive(
DataSet = work.apderiveexample,
LinkVarsA = A_Month A_Name A_Age A_Day,
LinkVarsB = B_month B_name B_age B_day,
OutVars = Month Name Age Day,
Outdata = work.AgreementPattern,
DorV = V,
Case = 3,
Comptypes = LF WI HF E,
Compvals = 1 0.85 1 0
);
```



work.AgreementPattern			
Month	Name	Age	Day
1	0	1	1
0	1	0	0
1	0	1	.
.	0	.	1

APP

```
%app(  
  
Vars = ,  
  
Nums = 2,  
  
Outdata = work.APP,  
  
DorV = V,  
  
Missings = N  
  
);
```

Description

In data linking practice, it is very common to represent the comparison between two records as an array of binary or ternary states.

App, the agreement pattern producer, is a macro that was originally written to produce a data set to represent all possible combinations of states for a set of variables, but it can also be used to create combinations in general.

The number of states a variable can take is supplied via the Nums parameter, and the elements start counting from 0. The variables that will constitute the data set are supplied via a space delimited list. Thus, all possible binary arrays may be created for a three variable case by supplying three variable names in the Vars parameter, and setting the Nums parameter to 2.

There is an additional parameter, Missings, which may be set to “N” or “Y”, which determines whether to add a missing value into the mix of possible states for each variable. Thus when Nums = 2, and Missings = Y, the macro will produce a data set holding all possible ternary arrays, where the three possible values are 0, 1, and missing. Alternatively, a ternary array consisting of the states 0, 1 and 2 may be produced by setting N = 3 and Missings = N.

App is fantastic for working with the abstract notions of agreement patterns rather than the observations themselves. You can see how various agreement patterns behave when an evidence function or weight is applied to them, or it can be used as a starting point to create

control data sets. Alternatively, it can just be used as a way to create a data set or view representing combinations.

Combinations can become large extremely quickly, so the default behaviour of the macro is to produce a view, rather than a physical data set that resides on disk or takes up explicit space in memory.

Parameters

Vars: The list of variables to be contained in the output data set.

Nums: A parameter representing the number of states each variable in the combinations on the output data set can take. An extra parameter exists to enable the addition of a missing variable to this set.

Outdata: The reference that names the output data set. If not supplied by the user, defaults to work.app.

DorV: Determines whether the output data set will be a physical data set or a view. Can take the values of D or V. If the user does not supply this argument, it defaults to V.

Missings: This parameter can take the value of Y or N and defaults to N if not supplied by the user. If the user sets this parameter to Y, then the total number of states that each variable can take in the output combinations shall be the number supplied in the Nums parameter plus a missing value.

Example: Produce a data set view with all possible binary combinations for the variables Name, Age, Income and Address.

```
%app(  
Vars=Name Age Income Address,  
nums=2  
);
```



work.app			
Name	Age	Income	Address
0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1
1	0	0	0
1	0	0	1
1	0	1	0
1	0	1	1
1	1	0	0
1	1	0	1
1	1	1	0
1	1	1	1

Example 2: Produce a data set view with all possible ternary combinations for the variables Name, Age where one of the possible values is missing.

```
%app(  
Vars=Name Age,  
nums=2,  
missings = Y  
);
```



work.app	
Name	Age
0	0
0	1
0	.
1	0
1	1
1	.
.	0
.	1
.	.

APSUMMARY

```
%apsummary(  
  
Dataset = ,  
  
Outdata = work.apsummary,  
  
DropVars = ,  
  
CountVar = count,  
  
DorV = D,  
  
Exp = 12  
);
```

Description

Apsummary is a simple macro that can be used on an agreement pattern data set to create a summary count of the agreement patterns contained therein.

Technically, it consists of a simple front-end for a call to the hashcount macro, whereby all distinct combinations of the variables on the data set shall be counted.

If there are additional variables on the data set that the user does not wish to include, then they may be excluded from the summary by including them as a space delimited list fed into the optional DropVars parameter.

Parameters

Dataset: A reference to the data set that holds the agreement patterns.

Outdata: The reference to the output data set.

DropVars: If there are additional variables on the input data set that the user does not wish to be included, and hence summarized, they can be designated in this parameter via a space delimited list.

CountVar: The name of the variable on the output data set that will contain the counts of the observed patterns.

DorV: This parameter can take on the value of either “V” or “D”, which determines whether the output is in the form of a data step view, or a physical data set. If the user does not supply this parameter, it defaults to D.

Exp: This parameter can take a number between 1 and 20, and determines the number of buckets, via operating as a power of two, in the hash object that summarizes the agreement patterns. If the user does not supply this variable, it defaults to a value of 12.

Example: Generate a summary count of the agreement patterns in the data set work.AP.

Example Data Set

work.AP	
Name	Age
0	0
0	1
0	.
1	0
1	1
1	.
.	0
.	1
.	.
0	0
0	1
0	.
1	0
1	1
1	.
.	0
.	1
.	.


```
%apsummary(  
dataset=work.AP,  
outdata=work.APSummary,  
DorV=D);
```



work.APSummary		
Name	Age	Count
.	.	2
.	1	2
1	0	2
1	.	2
1	1	2
0	0	2
0	.	2
0	1	2
.	0	2

DJMASSIGNMENT

```
%djmassignment(  
  
  Dset = ,  
  
  Outdata = work.FINAL_ASSIGNMENT,  
  
  Qualstatsout = work.QUALITYSTATS,  
  
  Ida = ,  
  
  Idb = ,  
  
  Weightvar = ,  
  
  Stopafter = C,  
  
  Addgrade = N,  
  
  Gradevar = Grade,  
  
  Qualstats = Y,  
  
  Exp = 12,  
  
  Sasfileoption = N  
  
);
```

Description

DJM Assignment⁶ is a macro that implements a one to one assignment procedure and as a relatively convenient side effect, can also output a simple group of statistics that can be used to inform on the quality/performance of a linkage.

The current standard method of one to one assignment in data linkage community typically views the problem as best tackled by a form of linear sum assignment. An implementation

⁶ DJM merely stands for my own initials, which were included in the file name of the original macro implementation. It doesn't stand for any particular technique or principal.

known as the “auction algorithm” has commonly been used for this task. I have deliberately chosen not to implement one to one assignment in this way.

The word “optimal” is thrown around with extreme liberalism in the literature when discussing such methods. To be fair, it does sound impressive and in relation to linear sum assignment problems at least, it could technically said to be correct. But data linking is not the linear sum assignment problem. A curiosity is that despite a multitude of references, there are almost no concrete mechanical examples of the auction algorithm actually doing what it was professed to do: making optimal and improved one to one assignments⁷. There are plenty of examples of people saying that they used it, but few examples showing its results.

I authored this algorithm after making several observations:

- It can be trivially shown that there are “more-optimal” one to one assignments that do not maximise the global sum of evidentiary weights. Furthermore, these assignments can be found via alternative algorithms.
- The emotive connotations of the word “optimal” and the constant references in the literature resulted in rote application of linear sum assignment problem algorithms, which then created a positive feedback loop by creating more references in the literature. In the real world, a combination of rote learning and lack of reflection about the suitability of linear sum assignment algorithms for one to one assignment in data linking has potentially resulted in a degradation of the quality of the linkage projects rather than their improvement.
- A program was needed that could potentially handle large scale problems better than current assignment algorithms.

Not only does DJM Assignment make one to one assignments, but it comes with an additional benefit. I made the observation that the properties of the assignments made from the bipartite graph which the algorithm is asked to solve can be used as a simple commentary on the likely quality of the linkage.

DJM Assignment Theory

The first step to understanding the DJM Assignment macro is to gain an overview of the general assignment algorithm itself. It is NOT designed to solve the linear sum assignment problem.

⁷ The one example I ever witnessed whereby the assignment that the algorithm arrived at was actually illustrated in something like a realistic bipartite graph was by Peter Christen, in his book “Data Matching: Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection” which is very much to his credit. Unfortunately, it arrived at a solution that in my opinion actually displays that the algorithm commonly fails.

The macro accepts as input any data set that contains variables in the form of IDA, IDB, and weight. The general assignment problem can be thought of as a bipartite graph in which nodes/vertices⁸ of the graph are contained uniquely in sets A and B. IDA represents a unique record ID for a node in set A, and IDB represents a unique record ID for a node from set B. The weight variable represents the weighted value of the connection between IDA and IDB.

In the abstract, the algorithm runs as follows:

Step 0: Nodes and connections are established and start unassigned.

Step 1: Find unassigned nodes which are each other's uniquely highest weighted connection. Assign these nodes to each other and eliminate all alternative connections to these nodes. If there are no more potential assignments to be made, end the algorithm.

Step 2: Repeat Step 1 until all possible assignments are made. At this point, one of two conditions must be true. Either the number of additional potential nodes/connections remaining is 0, in which case the algorithm ends, or there are nodes that cannot be assigned using this method, because no pairs hold the quality of being each other's highest weighted unique connection.

Step 3: The algorithm determines which collections of nodes/connections constitute independent assignment problems. It aims to make one assignment amongst the nodes with the highest weights in each independent problem. One more piece of evidence is collected to make this decision. The average weight of all remaining potential connections to each candidate node are calculated. When potential assignment between the highest weighted nodes are checked, the two average weight numbers of the nodes that constitute a potential assignment are compared via:

$$\begin{aligned} & \text{Node A average remaining weight} + \text{Node B average remaining weight} \\ & = \text{Combined Average Weight Statistic} \end{aligned}$$

The pair of nodes with the lowest combined average weight statistic but still with the highest weight are given preference for assignment⁹. If there are no unique best candidates with the

⁸ I will be using the terminology of nodes/connections in this document to refer to the likes of bipartite graphs, rather than vertices/edges. I believe this will be more intuitive for non-mathematicians, given that the common use of the words like "vertices/edges" is likely to cause confusion.

⁹ This is a relatively simple heuristic, but its operation might still be a bit unintuitive. Why give preference to the pair with the lowest average alternative weights when we've been running on the basis that highest weights

highest weight for assignment within an assignment problem, a record is chosen randomly¹⁰ from amongst those that share highest preference for assignment. All alternative connections to the nodes that have been now been assigned are eliminated.

Step 4: Step 1 through Step 3 will now be repeated until all possible assignments have been made.

Two important things to note about this process are:

- It does not claim to maximise the global weights.
- It does not guarantee all nodes in either of the sets fed to the algorithm will be assigned¹¹

The first point generally does not matter, because maximising the total weight of all possible assignments is not what we should usually be interested in when data linking.

The second point generally does not matter because the records that will not be assigned by the algorithm will tend to be those you have practically no chance of assigning correctly.

I made the further observation that when trying to solve an assignment problem in this way, there is a property of the final assignments that can be used to comment on the quality of the data linkage.

determine truth for the rest of this algorithm? Firstly, we have to accept that if the algorithm reaches this point, your data linkage is already of relatively low quality for these remaining assignments and that the individual weights cannot distinguish between assignments. The logic goes that if a node comes from a large cluster of similarly highly weighted nodes, the odds that this highly weighted connection happened by chance is increasingly likely. Meanwhile, if two nodes have a connection that is highly weighted, but their alternative connections are all lowly weighted, odds are the reason for this connection is a signal of genuine information, and should therefore be given preference over the connections which are likely to be statistical noise. The odds of this actually happening are quite slim, as equal weights usually mean equal agreement patterns. But it does depend on the user's linking/weighting method, and it should be pointed out that given the diversity of possible weighting schemes it's impossible to come up with a perfectly "optimal" solution for all possible schemes.

¹⁰ Technically, I just rely on the order of the nodes as dictated in the algorithm by the use of a particular hash function in the macro, rather than call any pseudo-random number stream with any particular property or distribution.

¹¹ This is intentional. A major problem with the use of the linear sum assignment algorithms often sees practitioners supplying a subset of nodes and connections to the algorithm, usually by specifying some arbitrary cut-off on the weight variable, because the algorithms or storage mediums can't realistically handle problems involving all possible nodes and connections in the total comparison space. It can then be possible for "bad nodes" to enter this subset with only one or a few potential partners, all of which are incorrect. When this happens, the bad node often replaces the good assignments. This is because the particular linear sum assignment problem can commonly be defined as necessitating the finding of an assignment for all nodes fed to it. The only option is for the bad node to kick the good node out of its rightful place, and the good node is likely to go on to be included in another bad assignment. This effect cascades down linking runs often with the practitioner blithely unaware.

As a simple mnemonic, I have named these “A”, “B” and “C” grade assignments, though just because an assignment is “A” grade this does not necessarily mean that the linkage is necessarily correct, and all of this assumes the implementer of any data linking project operation possesses sufficient skill in the application of a weighting function to return a reasonable weight representing the preference for linking two nodes together.

The definition of these types of assignments are as follows:

A Grade: These assignments are those that were made in the first application of step 1 of the algorithm. They are the assignments where both nodes initially had one candidate connection with a unique highest weight, and the connection which satisfied this condition for each of the nodes was the connection to each other.

B Grade: These assignments are those that were assigned in the first pass of the step 2 algorithm. They couldn’t initially be assigned to a unique first preferences like those in the A Grade assignments, but once earlier high quality assignments had been made and potential alternatives were eliminated, they now had a unique best match to which they could be assigned.

C Grade: After the initial run through step 1 and step 2 algorithms, all further assignments¹² are classed as C grade links. These are assignments for which you typically have even less faith than B grade links, because they must all be based upon assignments that did not meet the criteria for A or B grade assignment.

In this way, metrics derived from the very action of one to one assignment can tell you something about the likely success and quality of your linkage¹³.

In a beautiful and ideal one to one linkage project, the candidate record pairs across the two sets separate out such that each one has an obvious best assignment (A Grade assignments are numerous).

As the quality degrades and the difficulty increases, more candidate nodes are unable to find or be linked to a first best preference (B Grade and C Grade assignments increase).

And as the quality gets obscenely bad, the function which assigned the weights to candidate nodes is unable to successfully differentiate in any way between good and bad candidates (C grade links increase).

¹² Regardless of whether they be made in later operations of step 1, step 2 or step 3...

¹³ Assuming your linkage actually requires you to perform a one to one assignment.

DJM Assignment: Resource Use and Practicalities

One of the additional limitations of assignment algorithms has been the size of the problems that can be realistically tackled. DJM Assignment can also require large amounts of memory and hard disk space.

Obviously, it requires disk space to hold the final output of assignments, but there are three “generalised caveats”.

Caveat one: Above and beyond the relatively fixed costs of operation, the macro requires sufficient memory to hold individual nodes of set A and set B, with accompanying weight information for each.¹⁴

Caveat two: The first set of data output to the disk comes during step 1, via the outputting of assigned records and the subsequent creation of a subset working data set representing remaining candidate nodes. DJM Assignment can accept as initial input a view that is intellectually much greater than the size of memory or disk storage¹⁵, but the work directory must be able to hold the currently assigned nodes and the resulting candidate node/connection information that results after the first run of step 1.

Caveat three: The algorithm involved in step 1 and step 2 performs relatively well. The algorithm used for step 3 can quickly become overwhelmed. How many potential connections are left at this stage depends on your nodes/connections and how many were removed during step 1 and step 2. The option exists to stop the algorithm before step 3 is reached if this is an issue, but a generalised worst case scenario would be those situations where all potential assignments share similar weights, in which there are no independent assignment problems, and whereby the relationships tying all nodes into one assignment problem are relatively deep and difficult to find.

This macro also prints incremental notes to the log informing the user how many assignments have been made, and how many nodes remain for assigning. They are interspersed somewhat with notes on other aspects of operation. These notes could be used to judge the performance of the operation and how close it is to completion if the user wishes to terminate the macro before it has completed.

¹⁴ Not to be confused with holding the input data, which would have multiple references to nodes in set A and set B. Specifically, the input data contains one reference for each connection, whereas DJM Assignment approximately requires memory to hold references for each individual node.

¹⁵ This version of DJM assignment is technically the second version of this macro. As long as time and other bottlenecks are not an issue, it can theoretically operate on input streams bigger than memory or disk, or problems of such a size that sorting in general would be problematic, but for which we still believe a workable solution still exists.

Parameters

Dset: This is a reference to the data set or view that contains the starting definitions of IDA, IDB, and weight information.

Outdata: This is a reference to a data set that will contain the final assignments. If the user does not supply this parameter, it defaults to `work.Final_assignments`.

Qualstatsout: If the quality statistics are requested, this is a reference to the data set in which they will be contained. If the user does not supply this parameter, it defaults to `work.QualityStats`.

Ida: The identifier of the variable on the input data set that contains the unique ID's of those nodes in Set A.

Idb: The identifier of the variable on the input data set that contains the unique ID's of those nodes in Set B.

Weightvar: The identifier of the variable on the input data set that defines the weight of the connection between the nodes in IDA and IDB.

Stopafter: DJM Assignment assigns three grades to the assignments that it creates, based upon where in the algorithm these assignments were made. The Stopafter parameter accepts the value of either A, B or C. The value determines where the algorithm will stop operation. If it is not supplied, it defaults to C. If you wish the algorithm to stop after it has found only the A grade links, enter the value A. If you wish the algorithm to stop after it has found the A and B grade links, enter the value B.

Addgrade: This parameter can take the value of Y or N. If the user does not supply this parameter, it defaults to N. If it is set to Y, then a variable representing grade information is added to the output data set of final assignments.

Gradevar: If the Addgrade parameter is set to Y, then the value of this parameter determines the name of the variable that holds the grade information on the output data set. Its default value is Grade if the user does not supply it.

Qualstats: This parameter is a reference to the data set that will hold the quality stat summary information for the entire assignment procedure.

Exp: This parameter determines the number of buckets, as a power of 2, which are used in the hash objects being internally created and used by the entire `djmassignment` macro. If the user

does not supply this parameter, it defaults to the value of 12. Acceptable values are between 1 and 20 inclusive.

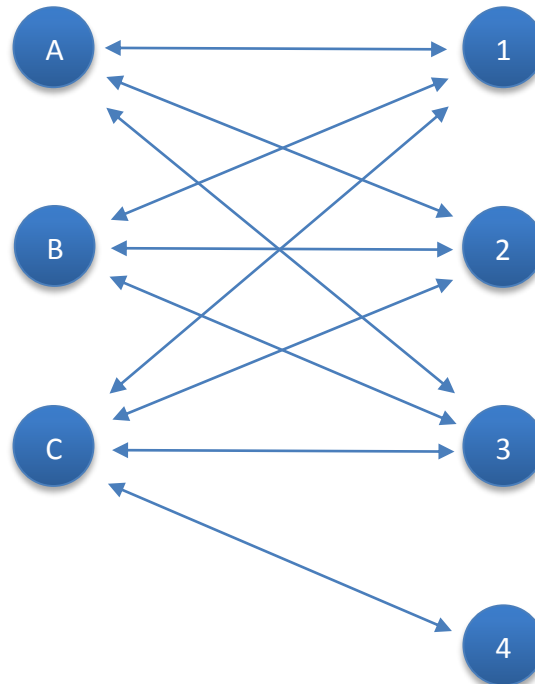
Sasfileoption: This parameter may be set to either Y or N, and if the user does not supply it, it defaults to N. The algorithm that runs in Step 3 of DJM Assignment has the option of loading its working data set and holding it in memory in order to speed up the process by which it can find the independent assignment problems. If you set this option to Y, you will need to ensure you actually have the additional memory required to hold such data. This option is not loading the entire assignment problem into memory, but only those candidate assignments that are left over after the A grade and B grade assignments have been made, so the feasibility and need to use this option depends somewhat on the properties of the bipartite graph that is being solved, and on the resources available in the user's environment.

Example: An extremely simple 1 to 1 assignment performed on the data set named `work.onetooneexample`

Example Data Set

work.onetooneexample		
id_a	id_b	score
A	2	20
A	1	10
A	3	5
B	2	15
B	1	14
B	3	10
C	1	4
C	2	4
C	3	4
C	4	4

Simple diagram of bipartite graph described in work.onetooneexample (score information not pictured)



Code

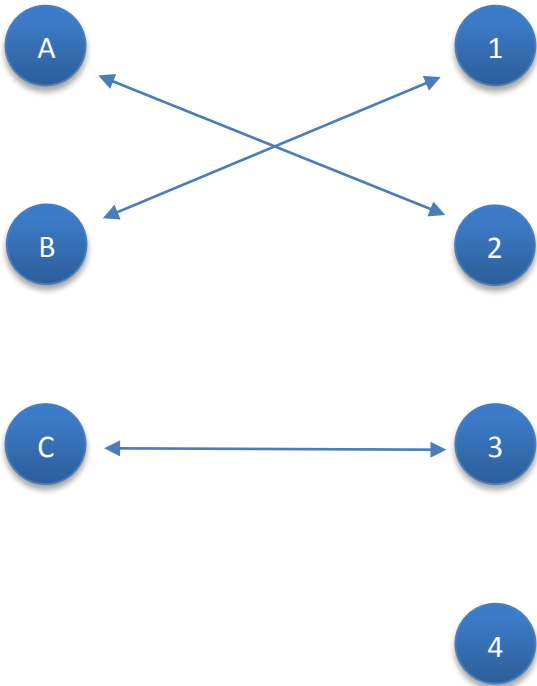
```
%djmassignment(  
dset=work.onetooneexample,  
    ida=id_a,  
    idb=id_b,  
    weightvar=score,  
    addgrade=Y,  
    gradevar=grade,  
    qualstats=Y  
);
```

Resulting Data Sets

work.FINAL_ASSIGNMENT			
id_a	id_b	score	grade
A	2	20	A
B	1	14	B
C	3	4	C

work.QUALITYSTATS	
Info	Number
A Grade	1
B Grade	1
C Grade	1
A Grade Percent	33.3333
B Grade Percent	33.3333
C Grade Percent	33.3333
Total Records Assigned	3

Resulting Graph - 1 to 1 Assigned



DSETSLICER

```
%dsetslicer(  
  
Dataset = ,  
  
N = ,  
  
Sequential = N,  
  
Partitions = ,  
  
DataSetRoot = work.Slice,  
  
DorV = D,  
  
Report = N,  
  
ReportDSet = work.Datasets,  
  
Deloriginal = N  
);
```

Description

The dsetslicer macro is designed to help decompose a larger data set in to a series of smaller data sets.

For example, work.bigdata may be split into several smaller data sets: work.smalldata1, work.smalldata2, and work.smalldata3. The particular root that determines the names and locations of the smaller data sets is supplied via DataSetRoot.

The operational inverse of dsetslicer is the dsetsmoosh macro, responsible for automatically returning many sequentially named data sets back into one, and thus both macros naturally complement each other's activities.

Either the N parameter or the Partition parameter can be supplied to the macro to designate the desired behaviour, but if both are supplied, then the N parameter takes precedence.

N determines how many equally sized¹⁶ data sets to produce from the original. It is supplied as an integer. The earlier example using `work.bigdata` would have been achieved with `N = 3`. This operation can be thought of as designating roughly equidistant "cut points" in the original data set required to provide the number of subsets requested. The observations between these cut points, and the records between the cut points and the beginning/end of the file, are then output as the respective smaller data sets. Logically, one cannot segment a larger data set into more segments than there are observations in the data set.

An alternative behaviour may be achieved when using the N option by changing the sequential parameter to Y, but such a behaviour inhibits the macro from outputting the new data sets as views. This alternative behaviour can be thought of as iterating through the original dataset one record at a time, and outputting one record in turn to each of the smaller data sets until the end of the original file is reached. Depending on the nature of the original data, this may ensure a more uniform distribution of characteristics/groups amongst the smaller data sets.

Alternatively again, the partitions parameter can be supplied instead of the N parameter with a list of space delimited integers. These numbers represent the "cut points" in the original data set, allowing the user to slice the original data set in a relatively arbitrary and non-uniform fashion if that is their wont. Logically, the integers must represent actual observation numbers within the original data set.

Some additional options can be turned on in the macro. If the Report parameter is set to Y, the macro will output an additional dataset supplied by the ReportDSet parameter with the names of the data sets that were created by the slicing operation.

The Deloriginal flag can be set to Y, which will delete the original data set after the slicing, leaving only the constituent smaller data sets remaining. It is not recommended that a regular user do this if they wish to output views instead of physical data sets, because the views will then have nothing to point to, but I have deliberately left this ability enabled, under the assumption that someone who chooses to do it knows what they are doing.

Parameters

Dataset: This is a reference to the original data set that will be split into the smaller constituent data sets.

¹⁶ Approximately equally sized that is. Obviously it is impossible to split a data set with 19 observations into 3 equally sized subsets.

N: This parameter, an integer smaller than the number of observations in the original data set, automatically enables the N behaviour if supplied. The macro splits the original data set into the number of data sets represented by the integer supplied via this parameter.

Sequential: If this parameter is set to Y, then the original data set is split not via "cut-points", but by passing each record in the original data set sequentially off to one of the respective smaller data sets in turn until there are no more observations in the original data set to distribute. If the user does not supply this parameter, it defaults to N.

Partitions: This parameter allows a string of integers to manually set the cut points of the dsetslicer macro. It should be supplied without supplying the N parameter. If the user supplied both, the N parameter takes precedence of the partition parameter behaviour.

DataSetRoot: This parameter represents the root of the output smaller datasets, which are referenced with an increasing integer appended to the root. If the user does not supply this parameter, it defaults to work.slice. In the small example included in the description earlier, the root parameter would have been set to work.smalldata.

DorV: A flag that can take the value of D or V, representing whether the smaller data sets will be physical data sets or views. If the user does not supply this parameter, it defaults to D. The sequential flag disallows the use of the V option, while the use of the V option combined with the Deloriginal flag can be problematic (but is not disabled).

Report: This flag can take the value of N or Y. If the user does not supply this parameter, it defaults to a value of N. If set to Y, an additional data set will be output containing the names of the smaller constituent data sets that were produced by the macro. The macro obtains the names by assuming that other data sets in the library do not share the same root as the data sets that were created, so keep that property in mind when using this option.

ReportDSet: This parameter should be a data set reference that determines the name of the additional output data set if the report option is set to Y. If the user does not supply this parameter, it defaults to work.DataSets.

Deloriginal: This parameter can take on the value of N or Y. If the user does not supply this parameter, it defaults to N. If it is set to Y, then the original data set is deleted once it has been sliced into smaller constituent data sets.

Example: Split the data set work.example into three smaller data sets

Example data set

work.example	
record	number
1	2
2	4
3	9
4	7
5	3
6	1
7	1
8	7
9	2
10	4

```
%dsetslicer(  
Dataset = work.example,  
    N = 3,  
DataSetRoot = work.slice  
);
```

Output Data Sets

work.slice1	
record	number
1	2
2	4
3	9

work.slice2	
record	number
4	7
5	3
6	1

work.slice3	
record	number
7	1
8	7
9	2
10	4

DSETSMOOSH

```
%dsetsmoosh(  
  
  Outdata = ,  
  
  N = ,  
  
  DataSetRoot = work.Slice,  
  
  Deloriginal = N  
  
);
```

Description

The dsetsmoosh macro can be considered the inverse operation of the dsetslicer macro. Its task is to write a program that constructs one large data set by combining smaller constituent data sets.

Unlike the dsetslicer macro, there are relatively few options for dsetsmoosh.

The Outdata parameter must be supplied, as it is the name of the output combined data set, and a root reference for the smaller data sets must also be supplied via the DataSetRoot parameter.

There are two more parameters that can be supplied, and both are optional.

The Deloriginal parameter is straight forward: it is either N or Y, and it determines whether the smaller data sets are deleted after they have been combined into one larger data set.

The N parameter can be supplied to let the macro know that the constituent data sets take the form of &root.1 through to &root.&n. However, the user can also choose to not supply this parameter at all, and then the macro itself will simply combine all the data sets found that share the root reference supplied by DataSetRoot.

For example, if there are three data sets work.part1, work.part2, and work.part3, the user can supply the parameter N=3 and DataSetRoot = work.part. But they could alternatively also choose to supply only DataSetRoot = work.part. The macro will then be responsible for finding all data sets in the work library whose names begin with "part". The macro will then automatically use these data sets and combine them into the final larger data set.

Parameters

Outdata: This parameter is a reference to the output final large dataset.

N: This parameter is optional. If supplied, it must be a number representing a number that is appended to the end of the Data Set Root, in the form of 1 to N, to designate the data sets that will be combined into the larger data set. If it is not supplied, the macro is responsible for finding and combining all data sets that share the Data Set Root. Please be careful when using the automatic option combined with the Deloriginal option.

DataSetRoot: This parameter determines the Data Set Root. If the user does not supply this parameter, it defaults to work.Slice.

Deloriginal: This parameter determines whether the smaller constituent data sets are deleted after they have been combined into the final large output data set. It may take values of Y or N, and if the user does not supply this parameter, it defaults to N.

Example: Combine the data set work.slice1, work.slice2, work.slice3 together into one bigger data set called work.example

Input Example Data Sets

work.slice1	
record	number
1	2
2	4
3	9

work.slice2	
record	number
4	7
5	3
6	1

work.slice3	
record	number
7	1
8	7
9	2
10	4

```
%dsetsmoosh(  
Outdata= work.example,  
DataSetRoot=work.Slice  
);
```

Output Data Sets

work.example	
record	number
1	2
2	4
3	9
4	7
5	3
6	1
7	1
8	7
9	2
10	4

GENPROB

```
%genprob(  
  
Dataset = ,  
  
ProbVars = ,  
  
Outdata = work.Probability,  
  
ProbMax = 0.99999999,  
  
ProbMin = 0.00000001,  
  
Positivevalue = 1,  
  
WeightVar = _DJM_AP_weight,  
  
Exp = 12  
);
```

Description

Genprob (Generate Probabilities) is a relatively simple macro designed to produce basic empirical U probability estimates in the standard practice of probabilistic data linking. These are roughly the odds of a particular variable within a comparison space possessing a certain agreement status due to chance.

The comparison space is represented by a file of weighted agreement patterns where the weight variable defines how many times a particular agreement pattern has been observed. The macro apsummary can be used on the output of apderive or any other stream of agreement patterns to generate such a file.

Genprob Theory

The phrase "empirical U probabilities" probably needs a little bit of an explanation.

In the contemporary language of the Fellegi/Sunter model of probabilistic data linkage, we generally talk about M and U probabilities for specific variables. M probabilities have a quality

of being generally difficult to observe and their use is theoretically problematic, but this doesn't seem to have slowed statisticians down.

U probabilities on the other hand can be easily estimated. In an informal, simplified and slightly incorrect definition, they can be thought of as the odds of any two random records achieving a certain agreement state when brought together for comparison.

The main thing about U probabilities is how easily we can generate good estimates for them. Because the problem of data linking scales quadratically, for all non-trivial data linking exercises the set that comprises the "real links" quickly shrinks into insignificance when compared to the scale of the set of "not real links". It does so at such a pace that for most sets of agreement patterns you can cheekily pretend it isn't there at all, meaning the set of all agreement patterns are approximately equal to a set of agreement patterns representing chance alone¹⁷.

The Genprob macro can then be used to generate a U probability estimate for any state within these agreement patterns, by seeing what percentage of observations within the entire set of agreement patterns are of a particular state, and then comparing this to the total number of agreement patterns/comparisons. It does this for all variables (except the weight variable), and outputs the result to a data set typically as a number between 0 and 1 for each variable.

As per theory, if you did this for all states in the agreement patterns, and added the U probabilities for each variable together from each of the output data sets, you should arrive at a total of 1 for each variable.

There are of course many uses for U probabilities. They can for instance be directly used in a weighting scheme for probabilistic linking. Alternatively, if you are so inclined, you can also use them as starting values for variables when operating an EM algorithm (see `icarus_em` macro).

Parameters

Dataset: A reference to the agreement pattern summary dataset (or other data set if you're being creative) that will be used in this calculation.

¹⁷ Or typically chance conditional upon some property if you're blocking. It is for the reason of relative scale between the "matched set" and the "unmatched set" that we can simplify the calculation of these stats down to agreement by chance. If we really wanted to be technically correct, we can talk about U probabilities as agreement/disagreement by chance on record pairs in the set of record pairs that aren't "real links". Of course, if we want to be even more specific, there is no existential demarcation of information on real world data sets into "matched sets" and "unmatched sets", and things are mainly thought of that way purely because of historical orthodoxy and tradition.

ProbVars: The variables from that data set for which probabilities are to be calculated. If the user does not supply this parameter, it defaults to all variables on the input dataset except for the weight variable.

Outdata: A reference to the output data set. If the user does not supply this parameter, it defaults to work.Probability.

ProbMax: A value that represents the maximum possible probability value. Since values of 1 are often both undesirable and unrealistic, it may be preferable to supply a probability that is arbitrarily close but not equal to 1. If the probability returned from the operation for any variable is greater than this number, then it is replaced with this number. Defaults to 0.99999999 if parameter is not supplied.

ProbMin: As the Probmax parameter sets a maximum valid value for a probability estimate, so too does the Probmin parameter set a minimum allowable value. Since values of 0 are often both undesirable and unrealistic, it is often preferable to supply a probability that is arbitrarily close but not equal to 0. If the probability returned from the operation for any variable is less than this number, then it is replaced with this number. Defaults to 0.00000001 if not supplied.

Positivevalue: Designates the value whose observation is to be counted within each variable, which will then be divided by the total number of observations implied by the value of weight. In the case of Agreement U probabilities in the standard Icarus agreement patterns, this value is 1. Disagreement U Probabilities are typically 0. If the user does not supply this parameter, it defaults to 1.

WeightVar: The variable on the input agreement pattern data set which represents the number of times each agreement pattern is observed. Defaults to _djm_ap_weight if the user does not supply this parameter, but this will cause the macro to abort if such a variable isn't actually on the input dataset.

Exp: This parameter represents the number of buckets, as a power of 2, which are used in the hash objects created by this macro. Valid values are 1 through 20. If the user does not supply a value for this parameter, it defaults to 12.

Example: Produce probabilities of each variable from the agreement pattern summary data set work.apsummary

Example data set

work.apsummary				
pcode	surname	sex	firstinitial	count
1	1	0	1	565
1	1	1	1	173403
0	1	0	0	1290
1	0	1	0	192
0	1	0	1	68
1	0	1	1	3
1	1	0	1	261
0	0	1	0	642346
1	0	0	0	475
0	0	1	1	3724
1	1	1	0	3152
0	0	0	0	502959
0	0	1	0	238
1	1	1	1	289
0	1	1	0	149
0	0	0	1	14658
0	0	1	1	9410
1	1	0	0	2584
1	1	1	0	39912
0	1	1	1	60

```
%genprob(Dataset=work.apsummary,  
          Outdata=work.Probability,  
          positivevalue=1,  
          WeightVar=count,  
          exp=12);
```



work.Probability			
pcode	surname	sex	firstinitial
0.15822167	0.15886434	0.62538815	0.14504226

GENWEIGHT

```
%genweight(  
  
MData = ,  
  
UData = ,  
  
MissMData = ,  
  
MissUData = ,  
  
Outdata = work.Weightfile,  
  
Weighttype = 1  
  
);
```

Description

In traditional probabilistic data linking, probability estimates are typically translated into weighting values for observations of particular states in an agreement pattern. They are then usually transformed via some log-esque operation that allows them to be summated together.

Genweight can be used to automatically create a data set of weights from input data sets of M Probabilities, U Probabilities, Missing M probabilities, and Missing M probabilities. Due to the fact that Genweight produces several different types of standard weights, not all sets are needed to produce all types of weights.

Genweight Theory

The data set that is output by the genweight macro is a 3 x N dataset, where there are 3 rows and N is the number of variables for which weights are being produced. The weights are produced assuming the variables are on all the input data sets, and all in the same order. The variable names from the input data sets will then be replicated on the output data set.

The first row on the output data set represents the weights that can be attributed to an observation when variables disagree.

The second row represents the weights that can be attributed to an observation when variables agree.

The third row represents the weights that can be attributed to an observation when a variable in the comparison is missing.

The user is responsible for ensuring that there is a meaningful interpretation of the probability files they are inputting to the Genweight macro with respect to the weighting structure they have requested. They should be particularly cautious if they are supplying probabilities of 0 or 1, and may instead wish to replace these with another number arbitrarily close to one or zero.

The 4 kinds of weights generated by genweight are as follows:

Weight Type 1: Missing variables are given a weight of zero

Required Data Sets: MData, UData

Weight Type 1 General Equations

$$\text{Disagree Weight} = \log((1 - MProb)/(1 - UProb))/\log(2)$$

$$\text{Agree Weight} = \log(MProb/UProb)/\log(2)$$

$$\text{Missing Weight} = 0$$

Weight Type 2: Missing variables are the equivalent of disagreement

Required Data Sets: MData, UData

Weight Type 2 General Equations

$$\text{Disagree Weight} = \log((1 - MProb)/(1 - UProb))/\log(2)$$

$$\text{Agree Weight} = \log(MProb/UProb)/\log(2)$$

$$\text{Missing Weight} = \log((1 - MProb)/(1 - UProb))/\log(2)$$

Weight Type 3: Missing weights calculated based upon all probabilities supplied

Required Data Sets: MData, UData, MissMData, MissUData

Weight Type 3 General Equations

$$\text{Disagree Weight} = \log((1 - M\text{Prob} - \text{Miss}M\text{Prob})/(1 - U\text{Prob} - \text{Miss}U\text{Prob}))/\log(2)$$

$$\text{Agree Weight} = \log(M\text{Prob}/U\text{Prob})/\log(2)$$

$$\text{Missing Weight} = \log(\text{Miss}M\text{Prob}/\text{Miss}U\text{Prob})/\log(2)$$

Weight Type 4: Weights are generated using only U probabilities. Non-agreement weights equal 0

Required Data Sets: UData

Weight Type 4 General Equations

$$\text{Disagree Weight} = 0$$

$$\text{Agree Weight} = \log(1/U\text{Prob})$$

$$\text{Missing Weight} = 0$$

Parameters

MData: This parameter consists of a reference to any required M probability dataset

UData: This parameter consists of a reference to any required U probability dataset

MissMData: This parameter consists of a reference to any required missing M probability dataset

MissUData: This parameter consists of a reference to any required missing U probability dataset

Outdata: This parameter references the weight data set that will be output by the genweight macro.

Weighttype: This parameter can be set to the number 1, 2, 3 or 4. It determines which of the 4 weight types in the documentation are produced.

Example 1: Create a weightfile of type 2 from the data sets work.MProbs and work.UProbs

Example Data Sets

work.MProbs				
author	postcode	surname	gender	initial
0.98853389	0.99999585	0.99999613	0.9937438	0.92063353

work.UProbs				
author	postcode	surname	gender	initial
0.00514981	0.00470026	0.00358255	0.4970014	0.06618961

```
%genweight(  
MData=work.mprobs,  
UData=work.uprobs,  
outdata=work.Weightfile,  
Weighttype=2  
);
```



work.Weightfile				
author	postcode	surname	gender	initial
-6.43903073	-17.87083069	-17.97346113	-6.329130822	-3.556528123
7.584627403	7.733038813	8.124791387	0.999624001	3.797950413
-6.43903073	-17.87083069	-17.97346113	-6.329130822	-3.556528123

HASHMAPPER

```
%hashmapper(  
IDA = ,  
IDB = ,  
DataSetA = ,  
DataSetB = ,  
OutdataA = work.HashMappedA,  
OutdataB = work.HashMappedB,  
Chars = abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789,  
VarRearrange = Y,  
Exp = 12,  
Hashprefix =  
);
```

Description

The hashmapper¹⁸ macro is an interesting beast I developed due to experiments on various memory and index design in SAS. It has no connection to data linking theory per se, but I include it both for my own purposes, and so anyone else may creatively put it to use.

Hashmapper takes two data sets with identical variables. It uses the characters specified in the Chars parameter to generate coded combinations which replace the original values in these data sets. It does this such that each variable's values maintain the equijoin relationships across the data sets, with each value replaced with a combination of characters. Each variable is then assigned a near minimum length while maintaining equijoin behaviour.

¹⁸ Hashmapper should not be confused with the theory of hash tables themselves, or the class of a similar object in Java. I named it before I knew it might generate confusion, and I know practically no Java, so apologies.

Since the meaningful values of the variables are effectively replaced by an unintelligible combinations of characters, Hashmapper asks the user to identify ID variables contained in both data sets. These will not be recoded.

The recoded data sets have the quality that they:

- Are much smaller than the original data sets
- Can produce the same equijoin/missing agreement patterns as the original 2 data sets
- Can return a stream of record IDs identical to those in the original data sets using the same equijoin predicates.

This means that pointer keys from the hashmapped data sets can be stored in hash objects and used to access the records in the original data sets. With some creativity, the hashmapped data sets can effectively act as compressed in-memory indexes for equijoin operations.¹⁹

Parameters

IDA: This parameter is the ID variable on Data Set A. It will be unchanged on OutdataA.

IDB: This parameter is the ID variable on Data Set B. It will be unchanged on OutdataB.

DataSetA: This parameter is a reference to the first input data set.

DataSetB: This parameter is a reference to the second input data set.

OutdataA: This parameter is a reference to the first output data set.

OutdataB: This parameter is a reference to the second output data set.

Chars: This parameter signifies the characters used in the variable recodings. Characters can be fed via a string of characters with no blanks or spaces between the characters. Regular SAS rules apply, so if a regular semicolon is included without masking, it will result in a syntax error. If left blank, this field defaults to alphanumeric characters.

VarRearrange: If set to Y, the macro ensures the variables are in their correct positions by rearranging them on the data sets. This is helpful if the user has gone to the trouble of ensuring

¹⁹ But when I say compressed, unlike regular compression, there is no added CPU cost of trying to decompress particular records. They are compressed only in the sense that equality relationships between two data sets can be stored uncompressed in far smaller pieces of information than that which typically exists in all the bells and whistles of real world data. It is also worth pointing out that the user would have to program the use and operation of these indexes, as I have not included a preset program or technique in Icarus specifically designed to target the hashmapped data sets.

the variables are the same on two data sets, but they are not in the same positions (which is a requirement of the macro). If the user does not supply this parameter, it defaults to Y.

Exp: This parameter represents the number of buckets, as a power of 2, which are used in the hash objects created by this macro. Valid values are 1 through 20. If the user does not supply a value for this parameter, it defaults to 12.

Hashprefix: A prefix value to append to each of the variables on the output data sets.

Example: Apply Hashmapper to work.dataone and work.datatwo. Use the characters 'a', 'b', and 'c' in the replacement codes.

Example data sets

work.dataone		
ID	Number1	Number2
1	4	16
2	5	21
3	20	26
4	9	27
5	19	17
6	4	25
7	12	17
8	21	26
9	29	28
10	3	30
11	12	16
12	27	15
13	12	26
14	6	18
15	7	15
16	19	29
17	0	29
18	26	32
19	25	18
20	13	30

work.datatwo		
ID	Number1	Number2
1	17	19
2	21	10
3	5	21
4	9	19
5	15	9
6	22	-3
7	6	19
8	21	11
9	17	24
10	29	24
11	22	0
12	23	17
13	0	8
14	17	8
15	6	16
16	7	25
17	2	19
18	19	1
19	15	10
20	21	20

```
%HashMapper(IDA=ID,IDB=ID,
  DataSetA=work.dataone,
  DataSetB=work.datatwo,
  outdataA=work.HashMappedA,
  outdataB=work.HashMappedB,
  chars=abc,
  VarRearrange=Y,exp=12);
```



work.HashmappedA		
ID	Number1	Number2
1	acb	ccb
2	cba	cab
3	cca	aba
4	aaa	cba
5	bba	cac
6	acb	abc
7	aac	cac
8	aab	aba
9	bab	aab
10	cbb	acb
11	aac	ccb
12	aca	cbb
13	aac	aba
14	ccb	baa
15	bca	cbb
16	bba	abb
17	caa	abb
18	aba	bac
19	cac	baa
20	baa	acb

work.HashmappedB		
ID	Number1	Number2
1	bac	bba
2	aab	bca
3	cba	cab
4	aaa	bba
5	cab	aaa
6	abb	bab
7	ccb	bba
8	aab	bbb
9	bac	aac
10	bab	aac
11	abb	caa
12	bcb	cac
13	caa	bcb
14	bac	bcb
15	ccb	ccb
16	bca	abc
17	bbb	bba
18	bba	aca
19	cab	bca
20	aab	cca

ICARUS_EM

```
%icarus_em(  
  
Dset = ,  
  
LinkVars = ,  
  
CountVar = ,  
  
Mstart = ,  
  
Ustart = ,  
  
Mmstart = ,  
  
Mustart = ,  
  
P_hatinitial = ,  
  
Epsconverge = 0.001,  
  
Maxiter = 1000,  
  
Mdata = work.mprobs,  
  
Udata = work.uprobs,  
  
Mmdata = work.mmprobs,  
  
Mudata = work.muprobs,  
  
Model = 3  
  
);
```

Description

The use of an Expectation-Maximisation algorithm has become a common method of estimating M and U probabilities for probabilistic data linking practices. Icarus_EM is the Icarus implementation of a 2 or 3 agreement-state EM model for this purpose. It uses summarised

agreement pattern data sets as input, and attempts to output 2 or 4 data sets of probability estimates based upon the agreement patterns and starting settings of the algorithm.²⁰

Unfortunately, the nature of the Expectation-Maximisation algorithm and a lack of access to a commercial version of SAS for the production of this manual somewhat hinders my ability to produce meaningful examples of the algorithm/macro as I have managed with many of the other macros.

Icarus EM Theory

Fortunately for us all, I am not about to repeat the entirety of historical EM theory in this document, nor its entire history in the field of data linking. One of my colleagues at the Australian Bureau of Statistics authored a research paper on the topic²¹. If you are unable to restrain yourself, it includes general formulae, methodological notes, as well as the usual host of compulsory references which you can follow.

The Icarus implementation of a 2 state and 3 state EM model accepts an agreement pattern summary file as input. It is up to the user to ensure that the states on the input agreement pattern data set matches the states expected by the model in the Icarus EM macro. Icarus EM assumes states of 1 for agreement and 0 for missing/disagreement in a 2 state EM model, and states of 1 for agreement, 0 for disagreement, and SAS missing values for missing in a 3 state EM model.

If the user is selecting the 2 state model, then they only need to supply starting M and U probabilities. If they are using the 3 state EM model, then they need to supply starting M probabilities, U probabilities, Missing M probabilities, and Missing U probabilities. These

²⁰ In a bitter twist, I am not a particular fan of the EM algorithm, and yet working to improve its resource usage and computational abilities was some of the first work I ever did after moving to a methodology position in the Bureau of Statistics. It has followed me around ever since. Subsequently it would be a shame not to include a new version specifically for Icarus' architecture.

My negative opinion comes from a strong feeling that it is an example of the statistician's preference for consistency with pre-learned models instead of dealing with the complexity of the phenomenon in front of them. EM is very useful when the problem is so easy you don't need it, and practically useless when the problem is so difficult that it won't work. People remove variables from the linking problem just to get the assumptions of the EM model to hold, when the added information from those variables is more useful than any EM results. Not to mention, the notion of using a model that assigns probabilities to matches/non-matches to feed into another model that assigns probabilities to matches/non-matches grates on me because its circularity is something that screams rote learner. Nonetheless, someone would probably complain if I didn't include it, and I've already spent too much time on it...

²¹ Carrie Samuels was the author, publication number was 1352.0.55.120, and the title was "Using the EM Algorithm to Estimate the Parameters of the Fellegi-Sunter Model for Data Linking". It should be publically available on the Australian Bureau of Statistics website.

probabilities can be supplied in the form of space delimited lists of numbers, where each probability consists of a value between 0 and 1. It is also the user's responsibility to ensure that the numbers they enter actually have a reasonable interpretation (i.e. that requisite probabilities do not sum to a value more than 1 in accordance with general Fellegi/Sunter, EM and probability theory). Each number represents the requisite starting probability for the linking variable in the same relative position.

Alternatively, the user can instead include a data set reference to feed in starting probability estimates, containing probabilities in the same format as those generated from the likes of the `genprob` macro also found in `Icarus`. If this is the case, then the user will also need to ensure that the order of the variables on the data sets is the same as the order of the Linking variable entered via the `LinkVars` parameter.

A variable on the agreement pattern summary data set then needs to be identified which corresponds to the number of times an agreement pattern is observed.

Finally, the estimated ratio of links/non-links is set via the `P_HatInitial` parameter, the maximum number of iterations before an automatic abort command is set via the `Maxiter` parameter, and the "distance" that determines when conditions for convergence has been achieved is specified in the `EpsConverge` parameter.

With all the settings applied, the algorithm will write, precompile, and run the requisite expectation and maximisation phases of the algorithm. Upon completion/convergence, it should output results into a set of either 2 or 4 probability data sets.

Icarus EM Practicalities

You need to have a certain level of experience with probability and data linking theory to use the EM algorithm properly, as there is no general way to ensure that the answers it gives can be considered "correct", "optimal" or even useful.

To be fair, if you do have said knowledge, when it fails it usually fails spectacularly, which means that someone who possesses said prerequisites can usually tell at a glance when it has done so.

In addition, my colleagues have also found a second and unlikely boon to its use. If your data standardisation routines have failed at some point in your project, or inadvertently introduced a bug into the data comparison process that you weren't expecting, then the answers that the EM algorithm produces can actually be extremely loud signals that something is wrong. M and U probability estimates that are exceptionally wrong or heading towards 0 or 1 are often blaring sirens that something has gone catastrophically wrong with the formatting and

operations on the original data, and so the EM algorithm can actually operate as a kind of inadvertent data quality control step.

Regardless of how or why the user is wanting to use the EM algorithm, it will do well to always be on guard, and as such, I will include several pointers from myself as an author and experienced operator which I have found useful in practice:

Start the algorithm with sensible values: The EM algorithm requires starting values to get the operation off the ground. The closer these values are to notionally correct answers, the more likely it is that the algorithm will converge to generally acceptable results.²² U probabilities can be effectively estimated for variables empirically via the `genprob` macro within Icarus. M probabilities can be generally thought of as being a function of error rates within the two data sources that are being matched, but they are comparatively hard to estimate, and `guestimates` must often be used in their place. If you had a set of “real links”, you could also use `genprob` to generate empirical M probabilities from that data set as well...but then why would you use EM?

Does P Hat resolve to a sensible value?: The value of P hat represents the proportion of records in the comparison space which are being attributed to the matched set. Whenever the algorithm does converge and terminate, it does the user good to think about whether the resulting proportion is in any way sensible given the project at hand. It is often possible in data linking projects to have some idea of how many links are feasible before one tries to find them, and so using this to estimate an a priori expected P Hat can be a good signal of believability. An inherent P Hat that is too small can lead to the algorithm failing to converge.

Are the M and U probabilities resolving to sensible values?: Again, one can usually obtain some kind of apriori reasoning as to what acceptable M and U probabilities should be before any algorithmic work or formal analysis. The U probabilities can be empirically found with a very high degree of accuracy, and M probabilities may be roughly estimated by looking at the general qualities of the data at hand.

Beware of non-normally distributed variables and complex relationships: The EM algorithm, and most applications of Fellegi/Sunter type methodology tend to assume that variables are conditionally independent from one another. This wouldn't really be a problem if it weren't for the fact that almost every variable and relationship we have any interest in observing isn't conditionally independent. Such a model can still work, and still work relatively well, but caution must be taken and awareness must be maintained. Such relationships increase the likelihood of the model failing to converge on reasonable results.

²² Of course, my cynical side can't help but point that if you are able to supply accurate starting values, perhaps you don't need to use the EM algorithm in the first place...

Decimal Numbers, Scale and Floating Points: It often shocks mathematicians and statisticians to learn that there are precision issues with how numbers are represented and operated on in a computer, but it is an inescapable fact of life. SAS, and hence Icarus, uses double precision floating point to represent numbers in the computer's programs and memory. There are a distinctly finite set of numbers that can be accurately representing in this format, and there is no realistically simple way of changing this while still remaining in a SAS environment²³. At some point of scale, which can easily be reached by Icarus, problems and unexpected behaviours can start to emerge due to the limitations of numbers that can be accurately represented by double precision floating point numbers in the calculations being used. There is also, however, a chance the user won't realise this, and that there won't be a noticeable effect from this lack of precision even though the effect exists.

Even if the answers look reasonable, doesn't mean they are right or suitable: As a subjectivist, and a general philosopher aware of reference class problems and their implications, it's incredibly jarring for me to have to talk about probability estimates based on largely incorrect assumptions and models as "right" or "wrong". Regardless, it has been my experience that answers finitely close to those a person expects can act like a siren song to the ear of Odysseus. This bias can lead one to quickly accept the answers that confirm your own beliefs, causing nagging values or consequences to go unobserved. If you wish to be a half decent scientist or analyst, always be cynical and distrustful of your own results and motivations²⁴.

Parameters

Dset: This parameter is a reference to an agreement pattern summary data set that will be used as input.

LinkVars: This parameter is a space delimited list, listing the variables on the input data set for whom various probabilities will be estimated.

CountVar: This parameter is the name of the variable on the input agreement pattern data set that represents the count of observations of that particular agreement pattern.

Mstart: Supplied either as a space delimited list of numbers between 1 and 0, or as a genprob macro data set reference, this parameter determines the starting M probabilities.

²³ Though this is a problem generally, not just a SAS problem. Although there are environments and ways to extend beyond double-floating point limitations, they do not come without their costs...

²⁴ A decent analyst or scientist, of course, should never necessarily be confused with being a professional or employed one. If the latter is your primary goal, it might be more advisable to never be cynical or distrustful of one's employer's results and motivations.

Ustart: Supplied either as a space delimited list of numbers between 1 and 0, or as a genprob macro data set reference, this parameter determines the starting U probabilities.

Mmstart: Supplied either as a space delimited list of numbers between 1 and 0, or as a genprob macro data set reference, this parameter determines the starting missing M probabilities.

Mustart: Supplied either as a space delimited list of numbers between 1 and 0, or as a genprob macro data set reference, this parameter determines the starting missing U probabilities.

P_hatinitial: Specified as a number between 0 and 1, this is the ratio of observations within the set of agreement patterns that can be thought of as constituting the starting estimate of the ratio of true links.

Epsconverge: The epsconverge parameter designates a "distance" that determines when convergence is achieved by the EM algorithm. If the user does not supply this parameter, it defaults to 0.001.

Maxiter: This parameter determines how many times the EM algorithm can iterate before it is automatically aborted if convergence has not been achieved. If the user does not supply this parameter, it defaults to 1000.

Mdata: This parameter is a reference to the output M probability data set. If not supplied by the user, it defaults to work.Mprobs.

Udata: This parameter is a reference to the output U probability data set. If not supplied by the user, it defaults to work.Uprobs.

MMdata: This parameter is a reference to the output missing M probability data set. If not supplied by the user, it defaults to work.MMprobs.

MUdata: This parameter is a reference to the output missing U probability data set. If not supplied by the user, it defaults to work.MUprobs.

Model: This parameter can take on the value "2" or "3", and determines whether the user is requesting operation of the 2 state or 3 state EM model. If the user selects the 2 state, they only need supply parameters for Mstart and Ustart starting probability parameters. The user should ensure that the agreement pattern summary data set they are using as input to the Icarus EM macro does in fact match the type of model they have specified in this parameter.

LOCAL_FAM

```
%local_fam(  
  
  Indata = ,  
  
  Outdata = ,  
  
  Ida=,  
  
  Idb=,  
  
  Keepvars = ,  
  
  Locfamvar = ,  
  
  Exp = 12,  
  
  Sasfileoption = N  
);
```

Description

The local family macro takes a data set containing two variables whose observations represent links between nodes in a bipartite graph. Unlike the DJM Assignment macro, it doesn't need a weight variable, but additional variables to be kept on the output dataset can be assigned via the Keepvars parameter.

Local family is responsible for discovering which of the nodes in the graph can be contained in their own independent graphs, and assigns an id number to the nodes in each resulting independent graph.

Another way of looking at the problem can be through the lens of blocking in the practice of data linking. If the record IDs were output from a comparison space resulting from blocking on a variable, each block could be thought of as its own independent bipartite graph. Records are only compared with records in the same block. If the output from these comparisons contained nothing but record IDs and no information of the blocking operation or the blocks each record pair were in, there may be no immediate indicator that blocks exist, or which record pairs are grouped in common blocks. The local family macro can rediscover this information.

If the operation that created the stream of record IDs did not partake in simple/traditional blocking, there might nevertheless be implicit and unintentional blocks that may be of interest. Of course, the local family macro can't tell you what those blocks represent (post codes, suburbs, names, etc.), but it does supply a new identifier for each of the blocks inherent in the data set.

The name of the macro comes from my own description of the phenomenon of independent graphs or blocks in such a list of record ID comparisons: I noticed that nodes in such streams were often grouped collections of "localised families", rather than being widely or randomly distributed.

The problem of solving this puzzle is relatively computationally intensive, so it may serve the user well to not assume they can feed the local family problems on quite the same scale as other problems in Icarus. Nonetheless, the user should experiment to understand the limitations and performance of the local family macro if they wish to use it.

The DJM Assignment macro uses local family to discover the independent bipartite graphs in its operation.

Because of the nature of the algorithm used in local family, it is not designed for use with SAS views.

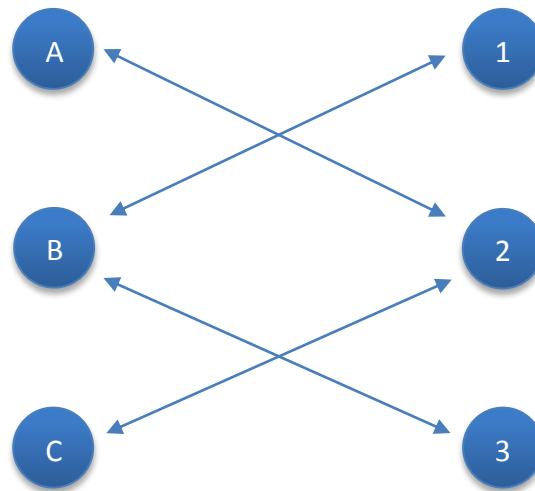
Local Family Theory

Local family is designed to break down a stream of nodes representable by bipartite graphs into identifiable and independent sub-graphs, and then append graph identifiers to the nodes within each independent sub-graph. One of the most trivial possible cases may be represented by the following data:

Data describing 2 trivially independent bipartite graphs

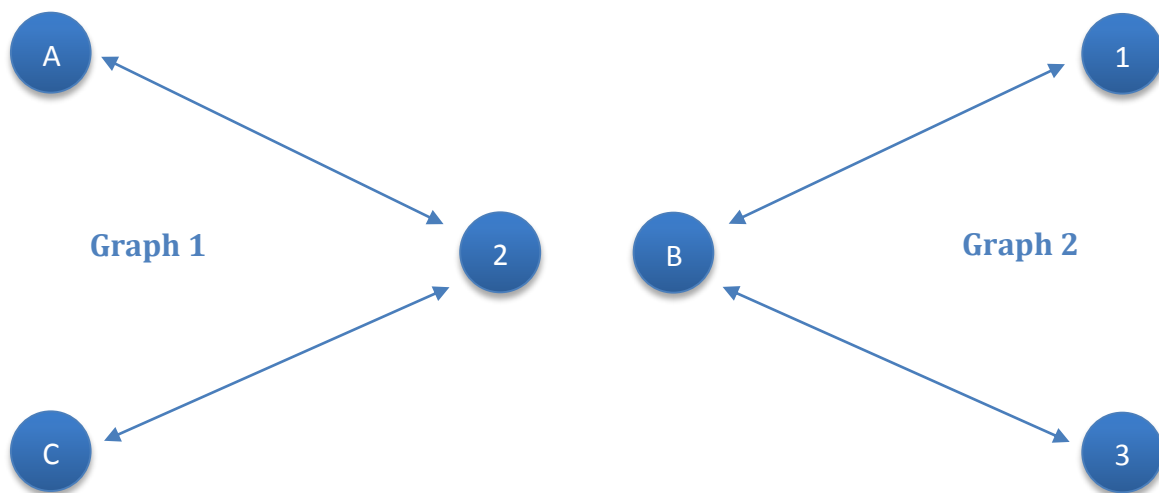
work.triviaexample	
ID_A	ID_B
A	2
C	2
B	3
B	1

Diagram of 2 trivially independent bipartite graphs



I have labelled the graphs as trivially independent because they are almost the simplest case possible whereby there is any actual obfuscation. Despite the triviality, the independent nature of the two graphs doesn't jump off the page at first glance.²⁵

The two graphs reveal themselves



²⁵ At least it doesn't jump off the page for me, but you might be smarter or more familiar with graphs than I am...

Of course, there is very little point in writing or using a computer algorithm for such a simple example, but it gives the reader an idea of how difficult even the simplest possible problems can be for a human.

Parameters

Indata: This parameter references the data set that contains the original bipartite graph as a series of IDs contained in two variables.

Outdata: This parameter references the output dataset that will contain the output data.

Ida: A reference to the first variable that contains Ids on the input data set.

Idb: A reference to the second variable that contains Ids on the input data set.

Keepvars: The user can optionally include a list of additional variables from the input data set that they wish to be kept on the output data set. This list is supplied via this space delimited parameter.

Locfamvar: The name of the variable that will contain the “local family identifier” for each particular record pair.

Exp: This parameter defines the number of buckets used by the hash object in the algorithm, in power of twos. Valid values are between 1 and 20. If the user does not supply this variable, it defaults to 12.

Sasfileoption: If the user has enough memory to hold the input data set while still retaining sufficient memory for the operation of the macro itself, they may supply this parameter and set it to Y to potentially quicken operation.

Example: Finding the local families of the trivial example data set

```
%local_fam(  
  indata=work.triviaexample,  
  outdata=work.locfamed,  
  ida=id_a,  
  idb=id_b,  
  locfamvar=localfamily  
);
```



work.locfamed		
Localfamily	ID_A	ID_B
1	A	2
1	C	2
3	B	3
3	B	1

NGRAMDSETLETTER

```
%ngramdsetletter(  
  
Dataset = ,  
  
Outdata = work.ngrammed,  
  
Var = ,  
  
N = 2,  
  
NgramVar = Ngram,  
  
DorV = V  
  
);
```

Description

Ngramdsetletter is a macro designed to apply ngram variables to a dataset. Ngrams are collections of N letters from a variable supplied in Var. Each ngram is placed in its own variable, named via the form &NgramVar.&i. The size of the ngrams are supplied via the N parameter.

An example of ngrams using the text 'boat' is as follows:

1-grams: b , o , a , t

2-grams: bo , oa , at

3-grams: boa , oat

4-grams: boat

The purpose of ngramdsetletter is not just to derive the ngrams, but to place them in variables for the same observation.

If all pieces of text are the same length then there will be the same number of ngrams for all observations. If this is not the case, some observations will contain missing variables in addition to their derived ngrams. You cannot use ngramdsetletter if the variable contains text that is shorter in length than the Ngram you requested.

Parameters

Dataset: This parameter is the data set that serves as input.

Outdata: This parameter is the data set that will output from the macro. If the user does not supply this parameter, it defaults to work.ngrammed.

Var: This parameter takes a variable name. Ngrams are derived from the text in this variable.

N: This parameter sets the dimension of the ngrams. If the user does not supply this parameter, it defaults to 2 (bi-grams).

NgramVar: This parameter represents the root of the ngram variable names that appear on the output data sets. The ngram variables take the form of &NgramVar.&i. If the user does not supply this parameter, it defaults to ngram.

DorV: This parameter can take the value of either D or V and determines whether the output data set is a physical data set or a view. If not supplied, it defaults to D.

Example: Add tri-gram variables from “string” in work.example. Output to work.trigrams. Ngram variables are called “NG”.

Example Data Set

work.example
string
appropriate
boxy
romeo
tomato
sparticus
buffalo
forest
rythmstick
funnybrian

Example Code

```
%ngramdsetletter(  
Dataset=work.example,  
Outdata=work.trigrams,  
Var=string,  
N=3,  
NgramVar=NG  
);
```

Output Data Set

work.trigrams									
string	ng1	ng2	ng3	ng4	ng5	ng6	ng7	ng8	ng9
appropriate	app	ppr	pro	rop	opr	pri	ria	iat	ate
boxy	box	oxy							
romeo	rom	ome	meo						
tomato	tom	oma	mat	ato					
sparticus	spa	par	art	rti	tic	icu	cus		
buffalo	buf	uff	ffa	fal	alo				
forest	for	ore	res	est					
rythmstick	ryt	yth	thm	hms	mst	sti	tic	ick	
funnybrian	fun	unn	nny	nyb	ybr	bri	ria	ian	

NGRAMDSETWORD

```
%ngramdsetword(  
  
Dataset = ,  
  
Outdata = work.ngrammed,  
  
Var = ,  
  
N = 2,  
  
NgramVar = Ngram,  
  
DorV = D,  
  
Delimiters = %STR( ),  
  
Modifiers = 0  
);
```

Description

Ngramdsetword is a macro designed to apply ngram variables to a dataset.

Ngrams are collections of N words from the variable supplied in Var. Each ngram is placed in its own variable, named via the form &Ngram.&i.

The size of the ngrams to be derived are supplied via the N parameter.

A simple example of ngrams using the text 'A long way home' is as follows:

1-grams: A, long, way, home

2-grams: A long, long way, way home

3-grams: A long way, long way home

4-grams: A long way home

The purpose of ngramdsetword is not just to derive ngrams, but also to place them in variables for the same observation.

If all pieces of text contain the same number of words then there will be the same number of ngrams for all observations. If this is not the case, some observations will contain missing variables in addition to their derived ngrams. You cannot use ngramdsetword if the variable contains text that has fewer words than the ngram you requested.

Parameters

Dataset: This parameter is the data set that serves as input.

Outdata: This parameter is the data set that will output from the macro. If the user does not supply this parameter, it defaults to work.ngrammed.

Var: This parameter takes a singular variable name. The ngrams will be derived from this variable.

N: This parameter sets the dimension of the ngrams. If the user does not supply this parameter, it defaults to 2 (bi-grams).

NgramVar: This parameter represents the root of the ngram variable names that appear on the output data set. The ngram variables take the form of &NgramVar.&i. If the user does not supply this variable, it defaults to Ngram.

DorV: This parameter can take the value of either D or V and determines whether the output is a physical data set or a data step view. If the user does not supply this variable, it defaults to D.

Delimiters: The macro uses the SAS scan function, and this parameter defines delimiters that define words. If the user does not supply this parameter, it defaults to %STR(), which is to say, words are delimited by blank spaces.

Modifiers: The macro uses the SAS scan function, and this parameter defines the use of modifiers for this function. If the user does not supply this parameter, it defaults to 'o' to increase the efficiency of the algorithm. Other modifiers can be included, and references are in the documentation of the SAS scan function. If the user is not familiar with these details, they should not supply the modifier argument, and accept the default operation.

Example: Add 1-gram variables from “newstring” in work.example and output to data set work.onegrams.

Example Data Set

work.example
newstring
Things fall apart
The center
cannot hold
Mere anarchy
is loosed
upon the world

Example Code

```
%ngramdsetword(  
  Dataset=work.example,  
  Outdata=work.onegrams,  
  Var=newstring,  
  N=1,  
  NgramVar=NG  
);
```



work.onegrams			
newstring	NG1	NG2	NG3
Things fall apart	Things	fall	apart
The center	The	center	
cannot hold	cannot	hold	
Mere anarchy	Mere	anarchy	
is loosed	is	loosed	
upon the world	upon	the	world

NGRAMLETTERSUMMARY

```
%ngramlettersummary(  
  
Dataset = ,  
  
Outdata = work.Ngramsummary,  
  
Var = ,  
  
Rollover = Y,  
  
N = 2,  
  
NgramVar = Ngram,  
  
Exp = 12  
);
```

Description

Ngramlettersummary produces a data set containing a count of ngrams found within a variable. In this case, the ngrams consist of those produced by the ngramdsetletter macro. The user must ensure that the variable fed to the ngramlettersummary macro does not contain fewer letters than the ngram selected.

The rollover option allows ngrams to be generated from text as though it continued on to the observation below it, allowing ngrams to be calculated for the entirety of a document if text must be spread across multiple observations. However, the restriction of the text length in each observation being greater than the length of the ngram still applies.

Parameters

Dataset: This parameter is the data set from which one will acquire Ngram statistics.

Outdata: This parameter is the output data set that will contain the Ngram statistics.

Var: This parameter is the name of the variable on which Ngram statistics are calculated.

Rollover: This parameter can take the value of Y or N. If set to Y, the macro conducts a count of ngrams produced from the text as though there was no distinction between individual observations. If set to N, ngrams are calculated within each observation.

N: This parameter sets the length of ngrams. If not supplied, it defaults to 2.

NgramVar: The name of the variable which holds the ngrams on the output data set.

Exp: This parameter sets the number of buckets, as a power of 2, in the hash objects used by this macro. Valid values are between 1 and 20 inclusive. If the user does not supply this parameter, it defaults to 12.

Example: Provide a summary count of one-grams from the variable “string” on work.example.

Example Data Set

work.example
String
Appropriate
Boxy
Romeo
Tomato
Sparticus
Buffalo
Forest
Rythmstick
funnybrian

Example Code

```
%ngramletterssummary(  
  Dataset=work.example,  
  Outdata=work.onegrams,  
    Var=string,  
  Rollover=N,  
    N=1,  
  NgramVar=Ngram,  
    exp=12  
  );
```

Output Data Set

work.onegrams	
Ngram	result
a	6
b	3
c	2
e	3
f	4
h	1
i	4
k	1
l	1
m	3
n	3
o	8
p	4
r	7
s	4
t	7
u	3
x	1
y	3

NGRAMWORDSUMMARY

```
%ngramwordsummary(  
  
Dataset = ,  
  
Outdata = work.Ngramsummary,  
  
Var = ,  
  
Rollover = Y,  
  
N = 2,  
  
NgramVar = Ngram,  
  
Delimiters = %STR( ),  
  
Modifiers = o,  
  
Exp = 12  
);
```

Description

Ngramwordsummary produces a data set that is a count of the ngrams found within the text of a variable. In this case, the ngrams consist of the words that constitute a text string. For examples, see the ngramdsetword macro documentation.

The user must ensure that the variable fed to the ngramwordsummary macro does not contain strings of text that contain fewer words than the length of the ngram selected.

The rollover option additionally exists such that ngrams will in fact be generated from each text string as though it continued on to the text string in the observation below it, allowing summary Ngrams to be calculated for the entirety of a document if text needs to be spread out across multiple observations. However, the practical issue of the text in each observation containing more words than the dimension of the Ngram still applies in this case.

Ngram summaries can be very helpful in the real world analysis of administrative data. For instance, when given a list of addresses, some of the most common Ngrams that can often be

found in such lists are those of name prefixes or street suffixes. This allows the user to single out such values for standardisation and recoding.

Parameters

Dataset: This parameter is a reference to the data set that will contain the variable on which one wants to acquire Ngram statistics.

Outdata: This parameter is a reference to the output data set that will contain the Ngram statistics.

Var: This parameter is the name of the variable that will contain the text on which Ngram statistics are calculated.

Rollover: This parameter can take the value of either Y or N. If set to Y, then the macro conducts a count of the Ngrams produced from the text as though there was not a distinction between individual observations (which is to say, an Ngram can contain letters from adjacent observations). If set to N, Ngrams are only calculated within each observation. You must still ensure that each observation in a variable contains more words than the dimension of the Ngrams you are requesting.

N: This parameter sets the dimension of the Ngrams. If the user does not supply this parameter, it defaults to 2.

NgramVar: The name of the variable which holds the value of Ngrams on the output data set.

Delimiters: The macro uses the SAS scan function, and this parameter defines the use of the delimiters in this function that define words in a string of text. If the user does not supply this parameter, it defaults to %STR(), which is to say, words are delimited by blank spaces.

Modifiers: The macro uses the SAS scan function, and this parameter defines the use of modifiers in this function. If the user does not supply this parameter, it defaults to 'o' to increase the efficiency of the algorithm. Other modifiers can be included, and their exact reference can be investigated in the documentation of the SAS scan function. If the user is not familiar or comfortable with these details, they should not supply the modifier argument, and accept the default operation.

Exp: This parameter sets the number of buckets, as a power of 2, in the hash objects used by this macro. Valid values are between 1 and 20 inclusive. If the user does not supply this parameter, it defaults to 12.

Example: Provide a summary count of the one-grams contained within the variable “newstring” on the data set work.example.

Example Data Set

work.example
newstring
Things fall apart
The center
cannot hold
Mere anarchy
is loosed
upon the world
The blood-dimmed
tide is loosed
and everywhere
The ceremony
of innocence
is drowned

Code

```
%ngramwordsummary(  
Dataset=work.example,  
Outdata=work.onegrams,  
Var=newstring,  
N=1,  
NgramVar=Ngram,  
Delimiters=%STR( )  
);
```

Output Data Set

work.onegrams	
Ngram	result
tide	1
and	1
Mere	1
apart	1
fall	1
blood-dimmed	1
the	1
cannot	1
loosed	2
drowned	1
hold	1
innocence	1
anarchy	1
is	3
The	3
center	1
ceremony	1
world	1
Things	1
of	1
upon	1
everywhere	1

OPTIMISEAP

```
%optimiseap(  
  
Dset = ,  
  
Excludevars = ,  
  
Outdata = work.Optimised_AP  
  
);
```

Description

A set of agreement patterns can often be used as a control data set for other macros such as `icarusindexjoin`. These other macros will often be used to find record pairs that are in the union of sets by using each agreement pattern as a key for an equijoin operation.

It can be very easy to get carried away when the computer is writing the code for you, and rely on the strategy of just throwing lots of agreement patterns around, knowing that automation will do all the hard work.

Often when returning the union of these multiple sets, you do not in fact need to use all agreement patterns to get the same result. If you had a programmatic way of easily determining which of those agreement patterns you actually needed to include, your automatically generated program would be more efficient, and your memory use would be less.

`Optimiseap` takes a set of agreement patterns and returns another set of agreement patterns. The returned set represents the minimal agreement patterns you would need on inner join operations to ensure you return those records defined as the union of all records from inner joins on all the agreement patterns in the original set.

Optimise AP Theory

Let us pretend that we have a population consisting of males and females across two data sets that we're linking. Each person also has various birth date and age details. If we wanted to find those record pairs defined by multiple blocking criteria, a simple way to do so is to produce the agreement patterns defining each blocking structure, and use this file as the control data set for one of the complex joining macros contained in `Icarus`. Providing you had the computer

memory to store them all, and the time to allow these algorithms to check the hash of each blocking strategy, this would technically work.

But once you start using many blocking criteria and face memory constraints, you can often do better.

Let us take the blocking patterns (or inner join keys) of: SEX, SEX + MONTHOFBIRTH, and SEX + DAYOFBIRTH. A casual user may get the computer to join on these three agreement patterns. But a moment of thought gives us insight into an optimisation. If we are doing an inner join on the variable sex, then the set of records returned must be a superset of the records returned by an inner join on sex + monthofbirth, and also a superset of the inner join sex + dayofbirth. We are wasting our time and computer resources hashing, holding and searching for records with the latter two key combinations when we know those records will have already been returned as a subset of a singular join on sex.

Subsequently, if you run these three agreement patterns through the agreement pattern optimisation macro, it will reduce them down to one join on the sex variable, and only output the agreement pattern that represents that join.

This might appear trivial when dealing with 3 key combinations on three variables, but it becomes far from trivial when dealing with thousands of key combinations on several variables that have been programmatically generated.

Optimise AP comes to the fore when implementing the “Melksham method”²⁶ of blocking in the practice of data linking. This can intellectually be described as follows. First, we generate all possible agreement patterns between two data sets. If we have a weighting function which defines the weight of any two record pairs, there is a good chance we can also write some useful “minimum possible weight function” as a function of the agreement patterns rather than the records themselves. This lets us know which equality-join agreement patterns could ever possibly have a weight above a certain cut-off before we’ve brought the original two data sets together. Those agreement patterns can then be subset, run through the agreement pattern optimiser, and used in one of Icarus’ multi-key join macros.

The resulting output records can either be kept, or have the more expensive operations conducted on them.

²⁶ I have no idea how to actually refer to this technique since I presented its proof of concept to my colleagues at the Australian Bureau of Statistics. Since I independently invented it, I don’t know if it exists out there in the world already. My colleague Sean Buttsworth has taken to referring to this strategy as “The Melkinator”, and all I can say is I hope a more suitable name is quickly found.

The result, if you do it right, is returning all record pairs from the full Cartesian product potentially above a certain weight, without actually having to go through the process of calculating or running the evidentiary functions on the full Cartesian product. Since the full Cartesian product is usually so severely saturated by those records which have almost nothing in common with each other, this task can actually be quicker than one might otherwise imagine, since many possible comparisons are automatically excluded.

Nonetheless, it would often be highly inefficient to attempt this strategy without first running the multiple agreement patterns through the Optimise AP macro, and it still does not follow that such a strategy is in fact applicable or feasible in all cases.

Parameters

Dset: This parameter is a reference to a data set containing original input agreement patterns.

Excludevars: If there are any additional variables on the input data set that don't constitute agreement patterns, the user may exclude them from the calculations by providing them to the macro in a space delimited list with this parameter.

Outdata: This parameter is a reference to the data set that will be output from the operation of this macro.

Example: Optimise the agreement patterns in the data set work.example

Example Data Set

work.example		
SEX	DOB	MOB
1	0	0
1	0	1
1	1	0

Code

```
%optimiseap(  
dset=work.example,  
outdata=work.Optimised_AP);
```



work.optimised_AP		
SEX	DOB	MOB
1	0	0

POINTY

```
%pointy(  
  
  PointData = ,  
  
  PointVarA = ,  
  
  PointVarB = ,  
  
  DataSetA = ,  
  
  DataSetB = ,  
  
  VarsA = ,  
  
  VarsB = ,  
  
  Prefixa = ,  
  
  Prefixb = ,  
  
  Outdata = work.pointed,  
  
  DorV = V  
  
);
```

Description

Pointy is a macro designed to take input from 3 data sets: PointData, DataSetA and DataSetB.

PointData must contain two variables of integers (PointVarA and PointVarB) relating to observation numbers on DataSetA and DataSetB.

Pointy will then use the integers in the PointVar variables to obtain the observations from DataSetA and DataSetB that relate to the observation numbers found in the PointVarA and PointvarB variables respectively.

It places the observations from the two data sets together and outputs them to a new data set as one observation.

Parameters

PointData: This parameter is a reference to a data set that contains two variables, both of which contain integers referring to the position of records on DataSetA and DataSetB.

PointVarA: The name of the variable on PointData that contains integers referring to the position of records on DataSetA.

PointVarB: The name of the variable on PointData that contains integers referring to the position of records on DataSetB.

DataSetA: This parameter is a reference to a data set from which records will be obtained, designated by the integers in PointVarA on the dataset PointData.

DataSetB: This parameter is a reference to a data set from which records will be obtained, designated by the integers in PointVarB on the dataset PointData.

VarsA: This parameter determines the variables that will be kept from DataSetA when the two observations are brought together. If the user does not supply this parameter, it defaults to every variable on DataSetA.

VarsB: This parameter determines the variables that will be kept from DataSetA when the two observations are brought together. If the user does not supply this parameter, it defaults to every variable on DataSetB.

Prefixa: A prefix value that will be appended to the front of each variable name from DataSetA.

Prefixb: A prefix value that will be appended to the front of each variable name from DataSetB.

Outdata: This parameter is a reference to the data set that will be output from the macro. If the user does not supply this parameter, it defaults to the value work.pointed.

DorV: This parameter can take the value D or V. It determines whether the output data set is a physical data set or a view. If the user does not supply this parameter, it defaults to the value V.

Example: Using three data sets with the pointy macro

Example Data Sets

work.pointdata1	
first	second
1	7
2	3
6	2
8	5
9	9

work.group1
info
a
b
c
d
e
f
g
h
i
j

work.group2
things
q
r
s
t
u
v
w
x
y
z

Code

```
%pointy(  
  PointData=work.pointdata1,  
  PointVarA=first,  
  PointVarB=second,  
  DataSetA=work.group1,  
  DataSetB=work.group2,  
  outdata=work.pointed,  
  DorV=V);
```

Output Data Set

work.pointed	
info	things
a	w
b	s
f	r
h	u
i	y

ROYALSAMPLER

```
%royalsampler(  
  
DataSetA = ,  
  
DataSetB = ,  
  
VarsA = ,  
  
VarsB = ,  
  
Prefixa =,  
  
Prefixb =,  
  
Outdata = work.RoyalSampled,  
  
DorV = V,  
  
NumRecords = 2000000  
);
```

Description

Royal Sampler (which is named for the highly technical reason of being a Simpson's reference), is a macro designed to randomly bring together records from two datasets and place them into an output data set as a single observation. In this way, it is very handy for sampling and drawing empirical references about the comparison space and deducing the general properties of all record pairs in the Cartesian product without actually having to generate them.

RoyalSampler Theory

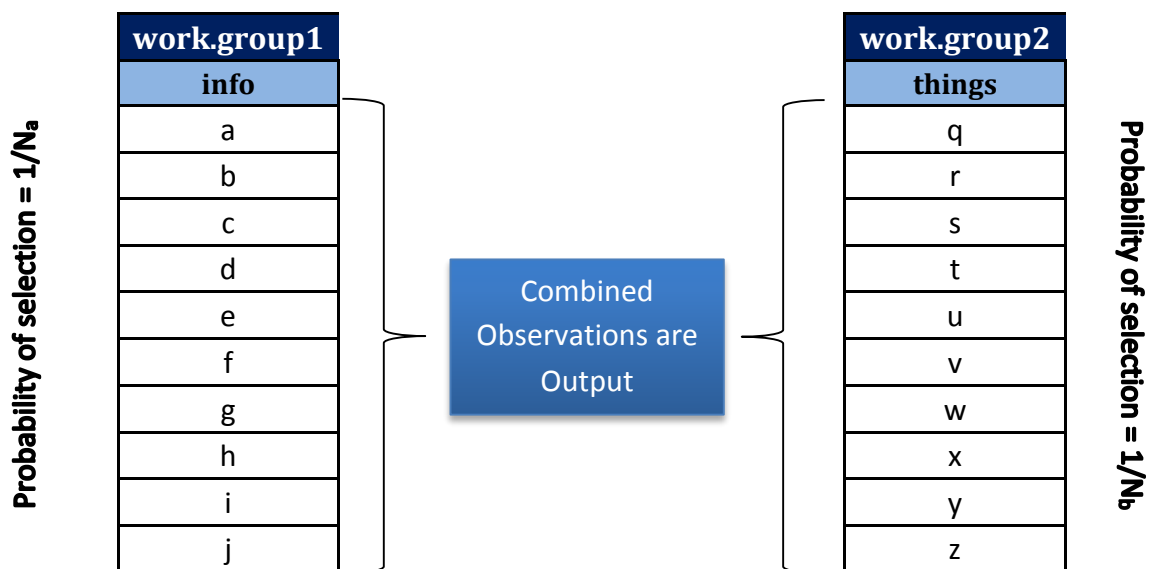
Let there be two data sets, labelled A and B. A consists of N_A observations, and B consists of N_B observations.

An integer for both data sets is randomly generated with uniform probability. On data set A, this integer can take a value between 1 and N_A inclusive, with the probability of any record being chosen being $1/N_A$. For B, the integer can take a value between 1 and N_B , with the probability of any record being chosen being $1/N_B$.

With a record from both of the data sets chosen in this way, they are then brought together and written to the output data set as a singular observation.

This process repeats itself as many times as is requested in the macro call. When an observation is selected from either of the data sets, it is not removed, thus the process running on both of the data sets when it comes to record selection can be thought of as a form of simple random sampling with replacement.

Take note: Because this macro relies upon pseudo-random numbers, running it twice will not produce an identical data set each time.



Parameters

DataSetA: This parameter is a reference to the first data set that will be sampled from.

DataSetB: This parameter is a reference to the second data set that will be sampled from.

VarsA: This parameter determines which variables from Data Set A will be kept on the output data set. It can be supplied via a space delimited list. If the user does not supply this parameter, it defaults to every variable in Data Set A.

VarsB: This parameter determines which variables from Data Set B will be kept on the output data set. It can be supplied via a space delimited list. If the user does not supply this parameter, it defaults to every variable in Data Set B.

Prefixa: This parameter allows the use of an optional prefix to apply to the variable names of all variables from Data Set A on the output data set.

Prefixb: This parameter allows the use of an optional prefix to apply to the variable names of all variables from Data Set B on the output data set.

Outdata: This parameter is a reference to the data set that will be output due to the operation of this macro. If the user does not supply this parameter, it defaults to the value of work.RoyalSampled.

DorV: This parameter can take on the value of D or V. It determines whether the output data set is a physical data set, or a data step view. If the user does not supply this parameter, it defaults to the value of V.

NumRecords: This parameter is a number which determines the number of samples taken from both of the data sets. If the user does not supply this parameter, it currently defaults to a value of 1000000.

SIMPLEENCRYPT

```
%simpleencrypt(  
  
DataSet = ,  
  
Outdata = work.encrypted,  
  
EncryptVars = ,  
  
SD = 12  
  
);
```

Description

The simple encrypt macro applies the MD5 hash to variables specified in the Encryptvars parameter. It then creates a new output dataset with variables from the original dataset, where any of those variables supplied in EncryptVars are now hashed, and those which were not supplied are in their original form.

Parameters

DataSet: A reference to the original input data set which contains variables that the user wishes to be hashed using the MD5 encryption function.

Outdata: A reference to the data set that will be output from this macro.

EncryptVars: A list of variables, supplied via a space delimited list that determines which variables will have the encryption function applied to them.

SD: The number of significant digits to take into account when hashing numeric variables. Numeric variables will be converted to character strings for hashing, using the SAS format of the form BEST&SD... Numbers which contain more significant digits than this will be rounded, and hence there is the chance that if the user has multiple variable with many significant digits, they may be hashed to the same value if they are in fact arbitrarily close to one another. If the user does not supply this parameter, it defaults to 12, meaning SAS will use the BEST12. format to convert numbers into strings so they can have the MD5 function applied to them.

Example: Use the simple encryption macro to run the variable “name” on the data set work.example through the MD5 hash.

Example Data Set

work.example			
ID_Var	name	mob	yob
1	Charlie	7	1945
2	Enoch	6	1932
3	Xiu	7	1973
4	Carlos	3	1982
5	Sophie	1	1991
7	Marcel	.	1983
8	Akiko	2	1991

Code

```
%simpleencrypt(  
  DataSet=work.example,  
  Outdata=work.encrypted,  
  EncryptVars=name,  
  sd=12  
);
```

Output Data Set

work.encrypted			
name	ID_Var	mob	yob
EA2BFA79CD221F1E88A9BF39EA155CD8	1	7	1945
49B99FD93D3078A2019BC0E15072E67F	2	6	1932
226DF0A58FD1512010DBECC4DA085A54	3	7	1973
E3D780AE6B064B0F38EADAF9090B8FCF	4	3	1982
0B12404EE7E2719FD26DC68ED97299D1	5	1	1991
7F4A6730FEA32D3B2349B2C6878D7C46	7	.	1983
52553B7F95FFF48EF387E993822E6500	8	2	1991

SIMPLEEVIDENCE

```
%simpleevidence(  
  
DataSet = ,  
  
IDA = ,  
  
IDB = ,  
  
WeightData = ,  
  
Outdata = work.evidenced,  
  
DorV = V,  
  
Prefixa = ,  
  
Prefixb = ,  
  
Comptypes = ,  
  
Compvals = ,  
  
SumVar = TotalWeight,  
  
KeepNonSumVars = N  
  
);
```

Description

The simpleevidence macro is designed to apply weights to a comparison space to produce the most common comparisons, weightings, and outputs in the practice of probabilistic data linking.

Comparison spaces can be created by the likes of the hashjoin, multihashjoin, multihashpointjoin, icarusindexjoin, and plainblocking macros.

The output data sets or views can then be combined with a weight data set created by the genweight macro to weight record pair comparisons and output the record pair weights to a new data set or view.

Simple Evidence Theory

The Simple Evidence macro exists to create weighted record pair output. It requires two data sets as input: the Dataset parameter representing the comparison space of record pairs and the WeightData parameter representing the agree/disagree/missing weights for variable comparison states.

The Dataset parameter must be a data set of record pair comparisons with a requisite ID variable for both records in the record pair. These ID variables must be provided in the IDA and IDB parameters as they literally appear on the comparison data set. They do not have a prefix applied to them like the other variables will. Thus, if they have a prefix, it must be included in arguments to the IDA and/or IDB parameters.

Additional variables for each record in the record pair must be named the same as the variables found on the weight file. The additional variables can differ from the variables in the Weightfile if they have an optional prefix appended to the front of them, but are otherwise identical (and indeed, at least one variable will have to be prefixed since you cannot have two variables named the same in the one observation). The optional prefixes are supplied in the Prefixa and Prefixb parameters.

Once the macro knows the location of the IDs, the comparison data set, and the weight data file, it goes to work coding up a comparison that will turn each observation into an output form of the three variables: IDA, IDB and SumVar. The SumVar is the total weight that will be given to that particular record pair comparison. Basic additional options can be supplied via the Comptypes and Compvals parameters in a similar fashion to the apderive macro.

Indeed, these options are practically identical in their operation as those in the apderive macro, only now the agreement pattern states that the apderive macro would have produced are instead replaced with the weights that correspond to those states. The user can think of the operation as applying weights to the agreement patterns that would be produced by the likes of the apderive macro and then having them summed into a total weight.

If there are no parameters fed into Comptypes and Compvals, the macro will apply the weights as simple functions of agreement, disagreement and missing defined by the equality operator.

Parameters

DataSet: This parameter is a data set reference to an input data set of record pair comparisons.

IDA: This parameter is the name of the variable on the input data set of record pair comparisons that represents the ID of records from one data set.

IDB: This parameter is the name of the variable on the input data set of record pair comparisons that represents the ID of records from the other data set.

WeightData: This parameter is a data set reference to an input data set of weight information, such as the output provided by the genweight macro.

Outdata: This parameter is a data set reference that determines the name of the output data set from the simpleevidence macro. If the user does not supply this parameter, it defaults to work.simpleevidenced.

DorV: This parameter can take the value of "D" or "V" and determines whether the output from this macro is in the form of a physical data set, or a data step view. If the user does not supply this parameter, it defaults to "V".

Prefixa: If the variables from one data set in a record pair comparison have a prefix appended to them which stops them from matching the variables in the weight data set, this parameter lets you set this prefix value so that they are now treated correctly.

Prefixb: If the variables from one data set in a record pair comparison have a prefix appended to them which stops them from matching the variables in the weight data set, this parameter lets you set this prefix value so that they are now treated correctly.

Comptypes: This parameter accepts a space delimited string, which determines additional comparison operations/functions that determine how the weights are applied to each record pair comparison. If the user supplies this parameter, it must have the same number of members as there are variables in the weightfile. For the specific values and treatment of this parameter, see the simple evidence theory section. If the user does not supply this parameter, all comparisons between variables determine their state via equality.

Compvals: This parameter accepts a space delimited string, which determines additional comparison operations/functions that determine how the weights are applied to each record pair comparison. If the user supplies this parameter, it must have the same number of members as there are variables in the weightfile. For the specific values and treatment of this parameter, see the simple evidence theory section. If the user does not supply this parameter, it is treated as though the user entered the value of 0 for every required member.

SumVar: This parameter sets the name of the variable on the final output data set that will contain the total weight of the record pairs.

KeepNonSumVars: This parameter can be set to Y or N. If the user does not supply this parameter, it defaults to N. If it is set to Y, the output data set will also contain the individual weights generated from each variable comparison calculation in addition to their total sum in the SumVar.

Example: Use the weight file `work.exampleweight` combined with the comparison file `work.examplecompare` to produce a simple output file of weighted record comparisons.

Example Data Sets

work.exampleweight		
name	age	sex
-5	-1	-3
3	2	1
0	0	0

work.examplecompare							
ID1	a_name	a_age	a_sex	ID2	b_name	b_age	b_sex
1	Ishmael	27	M	5	Ishmael	26	F
1	Ishmael	27	M	4	Ishmal	27	M
1	Ishmael	27	M	3		27	M
1	Ishmael	27	M	2		.	F

Code

```
%simpleevidence(  
DataSet = work.examplecompare,  
    IDA = ID1,  
    IDB = ID2,  
WeightData = work.exampleweight ,  
    Prefixa = a_,  
    Prefixb = b_  
);
```



work.evidenced		
ID1	ID2	TotalWeight
1	5	-1
1	4	-2
1	3	3
1	2	-3

TOPN

```
%topn(  
  
DataSet = ,  
  
Outdata = ,  
  
IDVar = ,  
  
WeightVar = ,  
  
N = ,  
  
WeightorID = ID,  
  
Performance = 1  
  
);
```

Description

The topn macro is a very simple macro designed to take a data set as input. Taking an ID variable and a weight variable on the referenced data set, it sorts and then outputs the top observations that contain a given ID.

TOPN allows two different methods of defining the top N records: ID or Weight.

If the ID option is used, then the macro returns the top N observations for a given ID, sorted by weight. If there are a set of observations with identical weights, it does not have any guarantee which will be kept or which will be discarded if there is an ambiguous number of observations given the number of requested top records. In this operation, you can state for a fact that N records or less will be returned for each ID in a data set.

If the weight option is used, then the macro returns observations with the top N weights for a given ID. Under such an operation, you cannot state a general minimum²⁷ number of records

²⁷ Beyond obvious conclusions such as “less than or equal to the total number of observations for that ID on the given data set”.

that will be returned for each ID on the input data set. The number of observations for each ID returned depends on how many have one of the top N weights for that ID.

A performance option is suppliable in order to add some options to the sorting procedure used to sort the input data set to find the top N records.

Parameters

DataSet: This parameter is a reference to the input data set.

Outdata: This parameter is a reference to the data set that is output from the macro.

IDVar: This parameter is the name of the variable on the input data set that contains the identifiers.

WeightVar: This parameter is the name of the variable on the input data set that contains the weight that is used to sort the data set.

N: N represents the parameter fed into the macro to determine how many of the top observations shall be selected. If the ID behaviour is selected, then it represents the maximum number of observations for each ID that will be output. If the weight behaviour is selected, it represents the maximum number of unique weight values that will be output for each ID.

WeightorID: This parameter determines whether the macro partakes in the “Weight” or “ID” behaviour, which also happens to correspond with the valid inputs for this parameter. If the user does not supply this parameter, it defaults to “ID”.

Performance: This parameter may take the value of 1, 2, 3, or 4. If the user does not supply this parameter, it defaults to 1. The number contained in this parameter determines the added options that will be used in the sort procedure used by the macro:

- 1: Applies the SAS noequals option to the sort procedure.
- 2: Uses the regular proc sort procedure with no options.
- 3: Applies both the noequals and tagsort options to the sort procedure.
- 4: Applies only the tagsort option to the sort procedure.

For the details of these options, the user is directed to the SAS documentation for the SAS sort procedure.

Example: Apply the topn macro to the data set work.example. Use the ID operation and output the top 2 records for each ID value.

Example Data Set

work.example		
IDA	IDB	weight
1	2	10
1	3	15
1	5	5
2	3	12
2	3	10
3	8	10
3	9	12

Code

```
%topn(  
  DataSet=work.example,  
  Outdata=work.topned,  
  IDVar=IDA,  
  WeightVar=weight,  
  N=2,  
  WeightorID=ID  
);
```

Output Data Set

work.topned		
IDA	IDB	weight
1	3	15
1	2	10
2	3	12
2	3	10
3	9	12
3	8	10

Distributed Computing Macros

The majority of users will probably think of using Icarus with an installation of Base SAS on a single computer.

Impressive performance is possible using Icarus in this way, and Icarus has been written so as to get the most power possible out of a Base SAS installation. Processing performance and flexibility compared to some of the other “high performance” data linking solutions offered is favourable, to put it mildly.

However, the user may also have licenced the SAS/Connect product, and may have access to multiple computers running SAS software, or may explicitly wish to distribute various tasks across cores on a multi-core CPU.

I have authored several additional macros that find their utility when no longer thinking in terms of singular Icarus or SAS sessions.

They are designed to help the user focus on distributing large arbitrary tasks across multiple computers using SAS/Connect, largely irrespective of whether the user is partaking in data linking or not.

ICARUS_ADDNODE

%icarus_addnode(

ControlDataSet = ,

Alias = ,

RWork = ,

IcarusGerminate = ,

AuthDomain = ,

CMacVar= ,

ConnectRemote = ,

ConnectStatus = ,

ConnectWait = ,

CScript = ,

CSysRPutSync = ,

InheritLib = ,

Log = ,

Output = ,

NOCScript = ,

Notify = ,

Password = ,

Sascmd = ,

Server = ,

Serverv = ,

SignonWait = ,

Subject = ,

TBufSize =

);

Description

Icarus_addnode exists to create distributed computing control data sets. That is to say, a SAS data set that contains information on the remote sessions used and the settings to connect to each remote session.

Although the parameter list for the macro can appear relatively intimidating, and there is a subtle interaction between the options available, the vast majority of these options will never need to be supplied, and constitute little more than optional signals to the SAS/Connect signon statement.

Icarus_addnode Theory

One of the best ways to understand icarus_addnode is to study the SAS/Connect documentation for the "signon" statement. The majority of the parameters for the icarus_addnode macro are directly lifted as options to be applied to this statement. Indeed, all parameters except for ControlDataSet, Alias, RWork, and IcarusGerminate can loosely be thought of as being optional parameters that are fed through to a signon statement as referenced in the SAS/Connect documentation. They will only be used as options to this statement if the user actually supplies them, and subsequently, they largely follow the same rules and exceptions that determine correct operations of these options in the SAS/Connect signon statement documentation.

Four additional macro parameters are not direct commands for the signon statement, and thus require some additional explanation.

ControlDataSet is simplest, in that it is merely a data set reference to the output Control Data Set, but its operation is subtle, and we will return to see exactly how it is used by the icarus_addnode macro after explaining the other options.

Alias represents an alias used to refer to a remote session in a SAS rsubmit command.

RWork represents a local library reference that will refer to the work library of a remote session.

IcarusGerminate is a flag parameter. If the user sets it to Y, then Icarus will be germinated²⁸ on the remote session.

The `icarus_addnode` macro is tasked with creating a Control Data Set, which consists of all the connection settings for the remote connections that the user may wish to use in a distributed computing programming session. This means that its method of outputting and adjusting its arguments is slightly different to most of the other macros that output data sets.

The user needs to understand two principals about `icarus_addnode` to grasp this behaviour:

1. It can create a Control Data Set or append records to an already existing Control Data Set
2. It adjusts its parameters and input arguments to automatically adapt to the relative position of a new node/remote process within a Control Data Set.

The first principal is relatively easy to appreciate. If the data set referenced by the Control Data Set parameter does not exist, it will be created, and whatever settings have been fed into the macro will be the first observation in that data set.

However, if the Control Data Set already exists, then `icarus_addnode` appends the settings into a new observation on the end of the data set. SAS users need not concern themselves with the settings of lengths for variables on the old data set; they will be automatically adapted by the macro to accept the input of any new observations and will avoid truncation of values.

In this way, the user easily constructs a data set with the `icarus_addnode` macro that contains connection settings for multiple remote sessions to be used in any one distributed computing program. If the user is working on a multi-core CPU, and settings correspond with starting parallel sessions on the same machine, then the macro can be run several times with the same settings to set up the Control Data Set for such a program.

Of course, the discerning programmer will realise this creates a bit of a problem, because you can't "literally" have the same settings for multiple sessions. How will you refer to each remote session? How will the remote work libraries all be assigned if they are carry the same reference?

This brings us to the second principal. Some of the input settings are appended with the relative observation number that will be produced when the macro runs. If the macro is creating a data set, this observation number will be 1. In all other cases, it depends on the

²⁸ See the documentation for `icarus_germinate` macro. In short, Icarus can upload its own source code, run and install itself temporarily on remote sessions.

number of observations already contained in the particular Control Data Set. The number appended will be equal to that number plus one.

The parameters that principal 2 changes include:

- **Alias**
- **RWork**
- **CMacVar** - which is also additionally appended with the value of `_status`

Thus, if an Alias value of Icarus, an RWork value of Rmote and a CMacVar value of Icarus are all chosen, when the macro creates a new Control Data Set the actual values for those settings will become Icarus1, Rmote1 and Icarus1_status.

If the same settings are appended as the third remote connection in an already existing Control Data Set, then the values would become Icarus3, Rmote3 and Icarus3_status.

The reason for doing this is that it allows the macro to be run multiple times with identical settings, which then creates multiple valid connection settings from the same input. This is very helpful if you are operating on a multicore CPU, or want to set up several connections in one program or via your own macro.

Now it must be said that I have not in fact used all, or even half of the options that are available in the signon command with SAS/Connect, so I cannot make any comment as to what is definitely required in the user's situation and what is not. Nor can I truthfully comment on whether the Icarus distributed computing settings will work or will suffice in such a case. In this macro, the value of Alias is itself used as a macro variable to store the value of the ConnectRemote option, which holds the actual name/location of the computer you wish to connect to when establishing a remote session. This macro variable is then used directly in the signon statement without any macro variable token signifier. This behaviour in SAS is weirdly inconsistent and screams "hack" to me, but it seems to be what's required in my experience to actually get SAS/Connect sessions to work.

The macro itself is also relatively "dumb". It merely creates the Control Data Sets with the settings specified. It is only when you try to use the generated data set with the likes of `icarus_connect` that you will discover whether the settings you have entered will in fact work with Icarus or your environment.

Parameters

Given the host of parameters which serve no distinct purpose other than to feed options to the signon command, I will not be documenting all these additional parameters. Their behaviours

and documentations are covered in the actual SAS documentation. I will, however, cover the additional command parameters or those which are changed by principal 1 and/or principal 2 stated earlier in the documentation:

ControlDataSet: This parameter is a reference to the Control Data Set that will either be created by the macro, or a Control Data Set that will have an extra observation appended to it by the macro.

Alias: This parameter represents the name of a remote session. It will be appended with the number of the resulting observation in the Control Data Set.

RWork: This parameter represents the name of a library reference in the local session that will be assigned to the remote session's work directory. Subsequently, the rules of general library names apply, and it will be appended by a number that represents the resulting observation in the Control Data Set. The length of the library reference must take this into account, as the total length of the library in SAS is not allowed to be more than 8 characters. If the user does not wish to set a library reference to the remote work directory, do not supply this parameter.

IcarusGerminate: If the user supplies this parameter and sets it to Y, this is telling operations using this control data set that the user wishes to germinate Icarus on the remote session upon connection. For more information on Icarus germination, see the documentation for the `icarus_germinate` macro. If the user does not wish to germinate Icarus on the remote session, do not supply this parameter.

ConnectRemote: This parameter sets the actual reference to the computer that you wish to establish a remote session to. See the SAS documentation for this option in relation to the SAS/Connect signon command for more information.

CMacvar: The value of this parameter determines a macro variable which monitors the status of the remote session signon. It is appended not only with the observation number, but also with the string `"_status"`. If the user does not care about such a variable, they are free to not supply this parameter.

ICARUS_CONNECT

```
%icarus_connect(
```

```
ControlDataSet=work.nodes
```

```
);
```

Description

Icarus_connect is designed to automate and lift the heavy burden of writing huge amounts of "administrative" code when connecting to multiple SAS/Connect sessions. It references a Control Data Set that has been created with icarus_addnode, which details the connections to be used in the distributed computing program.

Its exact actions depend upon the details contained in the Control Data Set, but a summary of the three main tasks I would expect it to typically perform if I had set up the Control Data Set are as follows:

1. Connect to each session listed in the control data set. An alias is set for each remote session, and a local session macro variable is established to monitor the status of each remote session.
2. Automatically assign a local library reference to the work directory of the remote session.
3. Automatically germinate Icarus on the remote session so that Icarus commands and functions are fully accessible on the remote session as well as the local session.

More specifically, the macro can be thought of as iterating through the Control Data Set, and for each record, writing code to run the following algorithm:

1. Check the control settings for the SAS/Connect signon commands in the Control Data Set. Implement a signon procedure using the settings specified.
2. Check whether there is a specified remote work option in the Control Data Set. If there is, automatically establish a local library reference to the remote work directory.
3. Check whether the option to germinate²⁹ Icarus is specified. If it is, germinate Icarus on the remote session.

²⁹ See documentation on Icarus_germinate macro if this terminology confuses you.

Icarus_connect exists to implement the settings that have been established in the Control Data Sets that are written by icarus_addnode.

Subsequently, the user may find the purpose of icarus_connect clearer after reading the documentation of the icarus_addnode macro and creating a few icarus_addnode data sets.

If the user is struggling to operate the combination of icarus_addnode and icarus_connect, a possible solution may be to switch on the MPRINT system option in your sas environment and see what signon statements are being generated by using the icarus_connect macro based on the settings contained within the Control Data Set.

Parameters

ControlDataSet: This parameter is a reference to a control data set that is typically produced via the icarus_addnode macro.

ICARUS_DISTCODE

```
%icarus_distcode(  
  
ControlDset = ,  
  
Codedir = ,  
  
Codename = ,  
  
Wait = Y,  
  
Timeout = 30  
  
);
```

Description

Icarus_distcode uploads a sas code file to all remote sessions in a Control Data Set. The code is written to the same file in the remote session's work directory, so it is only temporarily unless the user takes additional action to move it to some other location. After uploading, the file is then run via an %include command on each remote session. The Wait and Timeout parameters can be used to control whether the macro waits for completion of the upload and code running on all remote sessions before it returns control to the local session.

Parameters

ControlDset: This parameter is a reference to a distributed computing control data set.

Codedir: This is the system path to the directory that holds the sas file to be uploaded.

Codename: This is the name of the file (i.e. code.sas) that holds the code to be uploaded to each remote session and executed.

Wait: This parameter can be set to Y or N, and determines whether icarus_distcode waits for the completion of uploading/code running on all the remote sessions before it returns control to the local session.

Timeout: If icarus_distcode is waiting for uploading/code running to complete, this parameter determines, as a number of seconds, the maximum amount of time the macro will wait before determining that something has gone awry and returning control to the local session.

ICARUS_DISTSLICE

```
%icarus_distslice(  
  
ControlDataSet = ,  
  
DataSet = ,  
  
DataSetNames = _ic_slice,  
  
Deloriginal = N,  
  
Includelocal = N  
  
);
```

Description

Icarus_distslice's purpose is to split an initial data set into the same number of smaller data sets as there are observations in a Control Data Set, and then distribute these partitions to each of the remote sessions.

The data set partitions will be placed in the work library of each of the remote sessions, and operation of this macro assumes that a remote library reference has been assigned for these work libraries, and that this was achieved via the settings in the Control Data Set. It then uses these remote work library references to distribute the partitioned data sets.

Parameters

ControlDataSet: This is a reference to the Control Data Set that specifies the remote sessions to which the data set's partitions will be distributed.

DataSet: This parameter is a reference to the data set that you wish to be split into pieces and sent to the requisite remote sessions.

DataSetNames: This parameter is not a full data set reference, since the macro will append the value of the requisite work library to the front of this parameter. It will also append a number to the end of this parameter, representing the relative position of the remote session within the Control Data Set. If the user does not supply this parameter, it defaults to _ic_slice. If there were three observations in the Control Data Set, each with a remote work library reference of

RWork1, Rwork2, and RWork3 respectively, then the original data set will be partitioned into RWork1._ic_slice1, RWork2._ic_slice2, and RWork3._ic_slice3.

Deloriginal: This parameter can take the value of Y or N. If the user does not supply this parameter, it defaults to N. If it is set to Y, then the data set referred to in the DataSet parameter shall be deleted.

Includelocal: This parameter can take the value of Y or N. If the user does not supply this parameter, then it defaults to N. If it is set to Y, then the original data set will actually be split into partitions equal to the number of observations in the Control Data Set + 1. This last partition then remains in the work directory of the local session. Continuing the example from the DataSetNames parameter, this data set would be called work._ic_slice4.

Example: Distribute work.example amongst the remote sessions

Example DataSet

work.example	
record	number
1	2
2	4
3	9
4	7
5	3
6	1
7	1
8	7
9	2
10	4

Simplified Representation of a Control Data Set

work.control				
alias	rwork	icarusgerminate	cmacvar	connectremote
IC1	RWORK1	Y	IC1_status	localhost
IC2	RWORK2	Y	IC2_status	localhost
IC3	RWORK3	Y	IC3_status	localhost

```
%icarus_distslice(  
ControlDataSet = work.control,  
DataSet = work.example,  
DataSetNames = _ic_slice,  
Deloriginal = N,  
Includelocal = N  
);
```



work.example

RWork1

_ic_slice1	
record	number
1	2
2	4
3	9

RWork2

_ic_slice2	
record	number
4	7
5	3
6	1

RWork3

_ic_slice3	
record	number
7	1
8	7
9	2
10	4

ICARUS_DISTSMOOSH

```
%icarus_distsmoosh(  
  
ControlDataSet = ,  
  
OutData = ,  
  
DataSetNames = _ic_slice,  
  
Deloriginal = N,  
  
Includelocal = N  
  
);
```

Description

Icarus_distsmoosh's purpose is to reconstruct split data sets that have been distributed across remote sessions.

The data set partitions are searched for in the work library of each of the remote sessions, and operation of this macro assumes that a remote library reference has been assigned for these work libraries, and that this was achieved via the settings in the Control Data Set. It then uses these remote work library references to return data sets which correspond to the pattern of &DataSetNames.&i in the remote work libraries, where i is the number of the observation in the Control Data Set that references that remote session.

Parameters

ControlDataSet: This parameter is a reference to the Control Data Set.

OutData: This parameter determines the location of the output data set that will be constructed from the partitioned data sets obtained from each of the remote sessions.

DataSetNames: This parameter determines the names of the data sets, but it is not a regular data set reference, because the macro assumes that the data sets are all held in the work libraries of the remote sessions. The data sets that will be retrieved will be of the form &Remotework.&DataSetNames.&i, where i is the number of the observation in the Control

Data Set that references that remote session, and Remotework is a macro variable designating the reference for the remote work library for a remote session in the Control Data Set.

Deloriginal: This parameter can be set to either Y or N. If the user does not supply this parameter, it defaults to N. If it is set to Y, then the partitioned data sets will be deleted from the remote sessions.

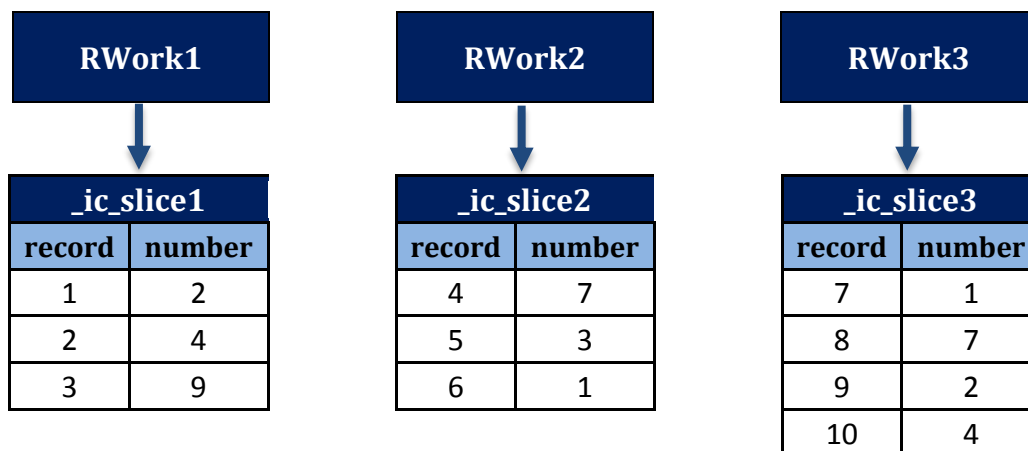
Includelocal: This parameter can be set to either Y or N. If the user does not supply this parameter, it defaults to N. If it is set to Y, then the macro assumes there is also a data set partition by the name of work.&DataSetNames.&N+1 where N is the number of observations in the Control Data Set. This data set will also be merged into the final OutData file.

Example: Recombine the _ic_slice data sets from the remote sessions

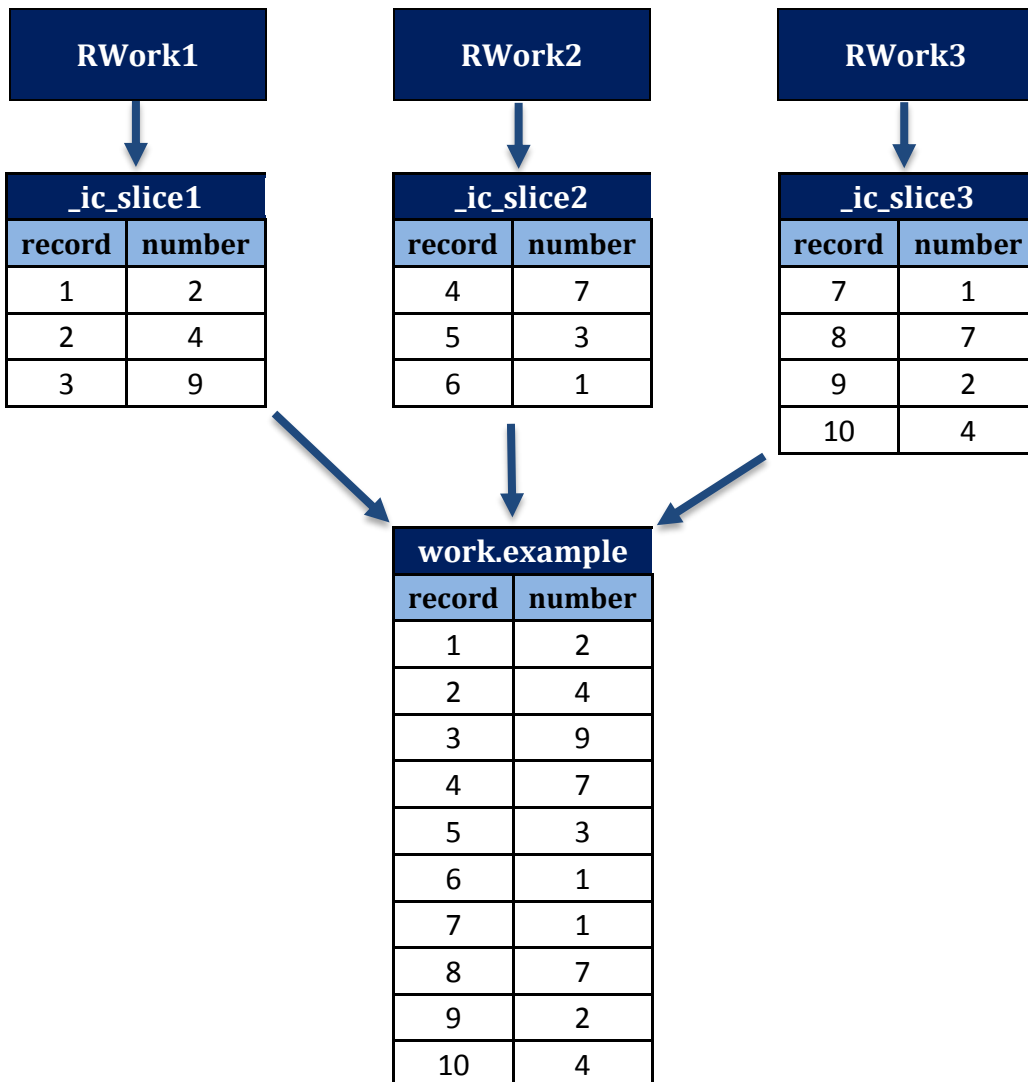
Simplified Representation of a Control Data Set

work.control				
alias	rwork	icarusgerminate	cmacvar	connectremote
IC1	RWORK1	Y	IC1_status	Blah.blah.blah
IC2	RWORK2	Y	IC2_status	Blah.blah.blah
IC3	RWORK3	Y	IC3_status	Blah.blah.blah

_ic_slice data sets and their respective locations



```
%icarus_distsmoosh(  
ControlDataSet=work.control,  
OutData=work.example,  
DataSetNames=_ic_slice,  
deloriginal=N,  
Includelocal=N);
```



ICARUS_DISTVAR

```
%icarus_distvar(  
  
ControlDataSet = ,  
  
DistVar = ,  
  
DistVarValue = ,  
  
LocalSessionVar = N  
  
);
```

Description

Icarus_distvar is a macro responsible for creating a macro variable on remote sessions.

The Control Data Set is of the type output by the icarus_addnode macro. The name of the remote variable is referenced by DistVar. It will be of global scope on all the remote sessions.

The value of the remote variable is specified via the DistVarValue parameter. If the user does not supply this parameter, then it will default to a rather curious behaviour. The variable will take a value on each of the remote sessions that is equal to their observation number in the Control Data Set.

The LocalSessionVar flag can be set to Y or N. If set to Y, a variable is also created in the local session. If there was not a value supplied for the variable to take, then the value on the local session becomes one more than the number of observations in the Control Data Set.

Parameters

ControlDataSet: This parameter is a reference to the control data set that will be used to identify all of the remote processes on which to set the macro variable.

DistVar: This parameter is the name of the variable that will be distributed to each of the remote processes.

DistVarValue: This parameter is optional. If the user chooses to supply it, it is the value that the macro variable will hold on each of the remote processes. If the user does not supply it, then the value that will be stored in the macro variable on each of the remote processes will be

equal to the observation number of that remote session as found in the original Control Data Set.

LocalSessionVar: This parameter can be set to Y or N. If it is set to Y, then a variable will also be created on the user's local session as per the earlier documentation. If the user does not supply this parameter, it defaults to N.

ICARUS_DISTVARDEL

```
%icarus_distvardel(  
  
ControlDataSet = ,  
  
DistVar = ,  
  
LocalSessionVar = N  
  
);
```

Description

This macro is related to and even simpler than `icarus_distvar`. It is responsible for deleting the global variables that you distribute with the `icarus_distvar` macro.

It too requires a Control Data Set as created by the `icarus_addnode` macro, and a specification of the global macro variable that will be deleted from each of the remote variables must be supplied in the `DistVar` parameter.

Parameters

ControlDataSet: This parameter references a remote processing Control Data Set, such as created by the `icarus_addnode` macro.

DistVar: This parameter contains the name of the global variable you wish to delete from each remote session.

LocalSessionVar: This parameter can take the value of either N or Y. If the user does not supply it, it takes the value of N. If it is set to Y, then `icarus_distvardel` also attempts to delete the variable designated by the `DistVar` parameter from the user's local session as well.

ICARUS_GERMINATE

```
%icarus_germinate(  
  
NodeAlias = ,  
  
Icaruslocation = &_Icarus_installation  
  
);
```

Description

Icarus_germinate is responsible for uploading and temporarily instantiating Icarus on a remote session.

The source code of Icarus is uploaded to the remote session's work directory, where the code is run, compiled and set up like a regular Icarus installation. The only exception is that this version writes all traces of itself to the remote work directory. Source code is copied to the work directory on the remote session, Icarus functions are compiled and stored in the work directory, and paths and function libraries are edited to point to files in the work directory. This has the effect of establishing files and settings as temporary, and they should all be removed when the SAS session on the remote computer ends as part of normal SAS behaviour.

Icarus germination lets you turn a remote session into a fully operational installation of Icarus, letting you use Icarus macros, functions and commands in remotely submitted code.

Parameters

NodeAlias: This parameter represents the alias used to represent the remote session in an rsubmit command.

Icaruslocation: This parameter represents the system path to the directory where Icarus is installed, which allows Icarus to identify code to be uploaded as part of the germination process. By default, the user does not need to submit this parameter unless there is some special reason to do so, as Icarus obtains this value from an automatically set global macro variable that is created when the Icarus installation command was called in the original session.

Closing comments, deliberate omissions and a word on further techniques

Data Repair and Standardisation

The reader may wonder why I haven't included additional repair and standardisation programs with Icarus.

Firstly, there is a product that is extremely powerful when it comes to operating on data to repair, edit, and transform information into its desired form. It is called "SAS".

Secondly, the repair and standardisation processes that need to be undertaken in a data linking process depends strongly on the actual nature of the data and the information being linked together. It is not very realistic or particularly productive to supply pre-prepared rules or standards for most kinds of data. It should be possible to easily come up with customised standardisation and repair rules that will quickly outperform any standardised or pre-packaged solution, especially in a product especially suited towards such a task like SAS.

There are of course several macros that can be used for such a purpose. Ngrams and frequencies are particularly useful for establishing valid values from the data itself, because they tend to appear with a far greater frequency than the erroneous values which tend to surround them.

Frequency based weights

Icarus attempts to be relatively technique neutral, while making it as easy as possible for the user to quickly program the technique which suits their needs. Therefore, there is no option which says "this is how you shall implement frequency based weights".

But this doesn't mean that you can't use frequency based weights. On the contrary, the Icarus hashcount macro allows one to easily produce frequencies from the data, as does the native SAS procedure PROC FREQ. Additionally, SAS formats and hash tables can be used as lookup techniques, so that values can be matched against the frequencies or weights in a program. The SAS data step language is especially suited to applying such calculations to data sets. The main choice the user faces is how they want to define and use frequencies when programming a project.

Why is there no cut-off/threshold setting macro?

There is no cut-off setting macro for two simple reasons:

1. “Cutting off” a data set conditional on being above or below a certain weight is one of the simplest possible commands in SAS.
2. Most of the simple theory-based methods used in the likes of probabilistic data linking to estimate cut-off points don’t actually work in any reliable way when dealing with real world projects.

Clerical Review Facilities

For those who haven’t been involved with the likes of probabilistic data linkage, this euphemism can be confusing. Translated, it means “get someone to look at it and decide upon link status manually”. Hopefully, your next thought was “I can’t believe they actually made up a phrase for that”.

SAS has many ways of summarising, graphing, presenting and looking at data, but this is not generally what is meant by the phrase “clerical review facility”, which typically describes a software process for viewing two records, and assigning such record pairs some status, usually via a mouse click or button press. Assigning status via programmatic means is of course possible in Icarus, and is infinitely preferable.

Forgive me for the essay that will now follow, but the reader must realise that I’m about to go against more than 50 years of probabilistic record linkage conditioning and practice.

I don’t include a clerical review facility because I don’t believe clerical review works. It is primarily a very expensive confidence trick and an historical hangover. Allow me to offer some explanation.

Clerical review³⁰ exists for three primary reasons:

1. **As a historical relative of modern data linking:** Before we had any computers, data linking had to be done by hand. As the sophistication of computers increased, so too did the amount of clerical review decrease. Yet it has always remained to some extent. When operating on data the size of which Icarus is capable, it is increasingly fair to say that we have reached the stage where the contribution financially realistic amounts of

³⁰ By which I mean the relatively standardized manual classification of many record pairs by some manual process of iterative visual inspection and classification. Looking at representations of data to conduct analysis or write/use programs addressing the actual act of classification or data linking is not traditionally considered as falling under the banner of clerical review.

clerical review can contribute to the overall results of a data linking project is tangentially approaching zero.

2. **Due to the influence of the 1969 paper by Fellegi and Sunter: A Theory for Record**

Linkage: The theory espoused in this paper became the predominant and most widely implementation of probabilistic data linking. The theory just happened to split record comparisons into three sets: linked records, unlinked records, and uncertain records. This third category has almost universally been interpreted as the group at which clerical review should predominantly be targeted. As it turns out, this is also typically the group for which human beings are least able to make any kind of rational decision. That hasn't typically mattered, because of the third reason...

3. **If given a choice between a process that is good but unquantifiable, versus a method that is inappropriate but quantifiable and repeatable, then statisticians, business, government, management and humans in general seem to have an almost universal tendency to prefer the later:**

For this reason clerical review sticks around. It provides metrics. Nice comfortable metrics. Metrics provide publications and marketing materials. The assumption is typically that the clerical reviewer is always right and this gives us a basis to make up all sorts of additional numbers. Such a phenomenon happens in programming with "lines of code". It happens in most 21st century science and statistics publications with confidence intervals and statistical significance. It happens in economics with anything that can or cannot be given a price. And it happens in data linking with clerical review. The assumption of unfettered truth in clerical review lets you make claims like "number of correct links", and "quality estimates" that sound nice and official. They give the illusion you can summarise a multi-stage and multi-faceted complex phenomenon down into a number or two. Such a notion is of course, absolute bollocks.

The clerical reviewer is not always right, and in fact there are very good reasons to believe they may be worse than a computer operated by a relatively competent programmer or analyst.

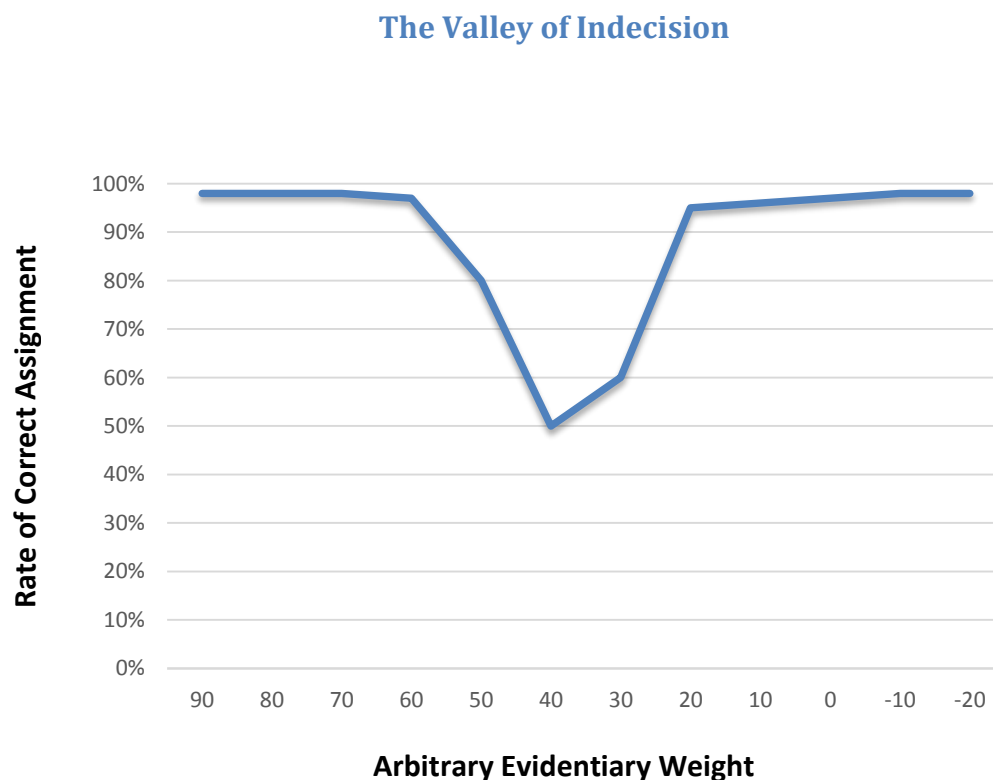
Let us first deal with some uncomfortable facts.

Humans respond to incentives, and these incentives in most professional workplaces will be measured by the number of records reviewed and/or the number of links found. In reality, humans will tend to link too many records. It is a natural consequence of our psychological biases, business metrics, and the operant conditioning inherent in the task.

Secondly, humans can only review an extremely finite amount of information at any one time. We cognitively misunderstand and misapply concepts of probability, and our brains just aren't suited or designed for this kind of problem. It is practically impossible for a human to optimise such problems with any global or higher order considerations. To make any kind of genuinely educated judgment on difficult records, a human would have to take an exorbitantly long amount of time investigating and weighing options, and their judgments usually reflect little more than their own personal experiences, biases, and more realistically, boredom.

This of course doesn't apply to all pieces of information. Some decisions are easy. But for those easy decisions, a computer makes the same decisions, and does so several orders of magnitude faster.

For the harder problems, clerical reviewers have very little on which to make any rational attribution. This leads to what I will call the "valley of indecision", and it is a curse to clerical reviewers in the same way it is to machines and algorithms.



It is a pattern that, if you performed analysis on clerical reviewers, classification algorithms, or programs, you would see in almost any real world situation. It represents the fact that people

and algorithms can make very good decisions at both ends of the evidentiary spectrum. When there is overwhelming evidence in favour of something, humans and computers are both very good at classifying such data. When there is overwhelming evidence to the contrary, humans and computers are very good at classifying such data.

And when there is borderline evidence either way, everyone struggles.

When humans struggle, their inherent biases usually take over and come to the fore, and tragically it's usually these situations at which clerical reviewers are targeted in great number.

Computers, though, have several main advantages over human beings even in this endeavour:

1. They unbiasedly do what they're told.³¹
2. They don't have that additional 0% to 5% rate of human error that seems to creep into just about every task that most humans naturally partake in.
3. Even though both humans and computers may make bad assignments, computers can make assignments equally bad much faster.

The second point is worthy of more respect than it typically earns, but I think as a point of ego we like to downplay it.

You don't have to actually get anywhere near perfection for an algorithm to outperform a group of humans in these kinds of repetitive classification tasks. Just get close enough to the level of performance imposed by the human error metric, and you'll probably do better overall.

And of course, there's always one last point that statistics so often likes to ignore. Often there isn't an objective metric, measurement, or definition of classification at which to aim in the first place.

Could we get a group of highly trained clerical reviewers to unbiasedly implement an extremely low rate of error with high rate of successful classification? Maybe. I have yet to see any evidence that you can easily train away basic human cognitive biases or conditions, or that training offers any significant adaption of human behaviour in this regard. In addition, anyone who has actually had any experience with large scale clerical review knows from first-hand experience the drudgery and frustrations from trying to actually do it, resulting again, in reversion to good old cognitive tricks and biases to try to get through a horribly boring task.

I do accept that it would be possible to come up with a very expensive and time consuming process that would result in a relatively accurate method of assignment.

³¹ Which is also their weakness...

But this process would come back to one of the main problems with clerical review. It wouldn't materially change the outcome of a large scale data linking project relative to the same performance achieved via fully programmatic and professionally analysed means given the size of the project in its entirety.

There is of course, one more problem with trying to implement the likes of a clerical review process in Icarus. Clerical review carries an additional traditional justification on the basis that there must be some kind of extra information available to the reviewers, even though in the real world this is not always the case. Icarus' tackles this issue by being as general as possible while being embedded in a general analytical system, and it would be incredibly difficult to design a software interface around any non-digitisable information, while maintaining Icarus' focus on general applicability.

My opinion is that one should choose a strategy that avoids at all costs the need for any kind of large scale clerical review. It will cost you money, will not materially improve your linkage, and although it gives you metrics if you assume your reviewers are never wrong, the use of such assumptions and metrics will make your linkage techniques and applications worse, rather than better³².

If the user truly wishes to partake in an expensive and large scale clerical review process, in spite of all these points, it is recommended that they design their own system with respect to the specifics of the projects on which they will be working and the environment in which they will be located.

Further work, development, and extensions to Icarus

There are admittedly always some areas in which any project can be extended, and I've learnt during the production and design of Icarus that programming is fighting the internal battle of what to eliminate, simplify and abandon. This battle is far more important and emotionally destructive than deciding what to include. There remains aspirational parts of Icarus I abandoned in its development to arrive at a timely finalised version. Truthfully, here at the end I don't feel that bad about excluding them.

The distributed computing section of Icarus was largely a natural and experimental extension to the individual base SAS section, and so could be systematised and generalised more. Much of the base part was a continuous cycle of natural evolution, planning, refactoring and reflection until I had a product I felt was usable without being trivial or impossible. I know I have been

³² Of course, the point of metrics is often not to improve a process, but to essentially improve its marketability, or to partake in some political or social process.

lazy in some parts of the code, as all honest programmers know, and that there are further optimisations and improvements that are always possible.

There are always additional and domain specific extensions that could be made: address parsers, and various databases of lookup information such as names, addresses and geographies. Yet to focus on such topics is to focus on one arbitrary domain in preference to all others, which is not what I was trying to achieve at this stage.

I do not know if there will be any demand for Icarus, or for extensions to its current structure and architecture. I leave any promises of future work and revision to the vicissitudes of fate. I wrote Icarus primarily for myself, and as with most projects of any scale amongst the intellectually curious, now that it is completed I have grown fickle and wonder what other adventures await. Further work will hence largely depend on what other people demand.

Towards the end of the programming and documentation of Icarus, I discovered the lisp family of languages, such that the final weeks have been spent between the laborious task of editing, documenting, bug checking and finalising the project, and the reinvigorating nature that comes from exploring a new language and seeing new possibilities.

But like any creative soul, beyond a small period of recuperation, I don't think stopping is ever a serious consideration.

For now, I believe I might take a bit of a rest.

Damien John Melksham 29/05/2013

Thanks and Acknowledgements

It occurs to me that in the typical composition of these kinds of endeavours, the author always gives thanks to the wrong things³³.

I've been lifted from this burden because there's been no one paying me to work on this project. I actually took leave to put it all together. In the same fashion, I didn't announce it in any substantial way until it was already completed, so there's no one I've been collaborating with.

I don't have academic servants or wage slaves to go through the laborious task of checking my work or the other backbreaking tasks involved in putting Icarus together. I've been unable to steal the ideas of my students and employees and pass them off as my own as is the usual way of the world, which could be considered a trifle annoying. The flipside of all that is that I've also had no supervisors, managers or executives to steal my ideas and take credit for them, so I suppose we can be thankful for small graces.

But before you think my head is getting too big, which I'm sure it always does to some extent, there are of course people deserving of recognition.

To my beautiful partner Erin Anderson, I offer sincere gratitude and thanks. She managed to put up with the click of my keyboard and my random incomprehensible outbursts of impotent verbiage. But most importantly, her effervescent personality and partnership has kept me from becoming a cantankerous and destitute shut-in. She keeps in perspective what I value in the world and reminds me there is always time to leave something and come to bed.

To all of the SAS technical paper authors, users who have ever written advice or answered people's questions online, and the general internet community of hackers and programmers: I learnt to program from your instructions. You do not know me, nor have I ever talked to you, and yet you all played an important part in the creation of Icarus.

To the Rancillio Silvia, whose constant supply of golden sweet nectar provides everlasting solace in an otherwise indifferent world.

Some thanks must go to the creators of SAS. I've been using it almost every day for the last few years, and SAS On Demand for Professionals ensured I understood any extra extensions of the

³³ Usually because social graces tend to flow in proportion to financial holdings and status rather than the far greater wealth of ability and actual contribution...

syntax/environment, and that all algorithms and macros work and perform like I thought they would when I designed them.

Thanks go to the creators of the Vim text editor, and especially the person responsible for the SAS syntax highlighting. Life was made much easier when I could write out all the code using my own resources. Even though I am not a heavy user, Vim was responsible for the first version of practically all code. Which I suppose brings me to some anonymous notepad producers as well, and some random pen producers, since several pieces of the code were in fact scribbled out on a few pages of paper when I didn't have access to a computer and was otherwise generally bored.

Thanks go to the authors of Apophysis, who are responsible for the program I used to produce the cover art of the Icarus manual. It seems appropriate that a project consisting of programs that write programs should have some cover art also generated by programmatic means. In addition, with its undeniable likeness to the arrogant showiness of the peacock's tail, I feel it may be an apt metaphor for the creation of Icarus itself.

And lastly, the authors of Emacs, SLIME, and Steel Bank Common Lisp, along with David Touretzky, Peter Seibel, and Paul Graham. Not only did your writings keep me entertained for the dreariest of these last few weeks when administrative tasks, bug hunting and documentation were necessary, but the final construction and installation macro of Icarus was stitched and edited together from the constituent source code by a program I wrote while learning Common Lisp, so kudos to you.

Contacting the Author, Licensing and Use

If you wish to contact the author, at the time of authorship you may do so by sending emails to either of the following addresses:

Damien.Melksham+icarus@gmail.com

Damien.Melksham+datalinking@gmail.com

Damien.Melksham+recordlinkage@gmail.com

As for licensing and use, the situation is as follows.

If you wish to use Icarus, you need to contact the author to negotiate access, prices, and terms of use. I am currently reserving all rights with regards to Icarus.