4.1 What the UNF Does

The "Universal Numeric Fingerprint" (UNF) solves the data interpretation problems just described, and does so in a way that is effortless to implement and check.

A UNF is created by rounding data values (or truncating strings) to a known number of digits (characters), representing those values in standard form (as 32bit Unicode-formatted strings), and applying a fingerprinting method (such as cryptographic hashing function) to this representation. UNFś are computed from data values provided by the statistical package, so they directly reflect the internal representation of the data - the data as the statistical package interprets it.

Thus, A UNF differs from an ordinary file checksum in several important ways:

1. *A UNF is detects substantively significant misinterpretation of the data by the statistical software.* If the statistical software misreads the file, the resulting UNF will not match the original, but the file checksums may match.

2. *A UNF is format independent.* The UNF for the data will be the same regardless of whether the data is saved as a R binary format, SAS formatted file, Stata formatted file, etc., but file checksums will differ.

3. *A UNF is robust to substantively unimportant rounding error.* A UNF will also be the same if the data differs in non-significant digits, a file checksum not.

4. *A UNF is strongly tamper resistant.* Any accidental or intentional changes to the data values will change the resulting UNF. Most file checksums and descriptive statistics detect only certain types of changes.

4.2 How a UNF Works

The method is generalizable to any digital object. In the abstract, it works as follows: an approximation algorithm is used to compute the approximated semantic content of the digital object. This approximated content is then put into a normalized form. A hash function is used to compute a unique fingerprint for the resulting normalized, approximated object, and the hash is stored. When the object is reloaded into the same or another application, this process is repeated, and the value generated at load time is compared to the stored value.

The first part of what is needed is a normalization function $f()$ that maps each sequence of numbers (or more commonly, blocks of bits) to a single value:

$$f\{i_0, i_1, ..., i_n\} \rightarrow c \tag{2}$$

To verify the data, we would need to compute f once when initially creating the matrix. Then recompute it *after* the data has been read into our statistics package. For robust verification, we should choose $f()$ such that small changes to the sequence are likely to yield different values in $f()$.

Normalization of objects alone, while it can be used as a basis of establishing identity across formats in limited cases, is inapplicable when reformatting of the object changes the precision, accuracy, or level of detail of an object in trivial ways. This is a well known issue is video and audio formats, in reformatting complex text documents, and surprisingly occurs commonly even in reformatting purely numerical databases.

Thus, a type-specific approximation is used, such as decimation, spatial or frequency down-sampling, and or numerical cutoff filtering, in addition, substitution, or combination to truncation and rounding as described. (For examples of decimation, downsampling and cutoff algorithms see: IEEE [1979], Renze & Oliver [1996].)

Any type-specific approximation function $A()$, may be employed. This approximation process, $A()$, accepts as input a digital object, O, of specified type, and an approximation-level parameter, k. A() should satisfy two these conditions:

1. . For some measure of semantic distance, d, if $k > k'$ then $d(O, A(O, k)) <= d(O, A(O, k'))$.

2. if $k >= k'$ then $A(A(O, k), k') = A(O, k')$

The UNF module for R currently implements versions one through four of the UNF algorithm, which are designed for numeric data. Version 4 is recommended, and the first version to be widely used. This precedes as follows:

1.

2. Each element in the numeric vector is rounded to k significant digits , using the IEEE 754 'round toward zero' rounding mode. [4] The default value of k is 7, the maximum expressible in single-precision floating point calculations.

3. Each element is then converted to a character string. Unless the element is missing or not a finite value, it is represented in exponential notation, in which non-informational zeros are omitted. This notation comprises:

   (a) A sign character.

   (b) A single leading period.

   (c) A decimal point.

   (d) Up to k-1 digits following the decimal, comprised of the remaining k-1 digits of the number, omitting trailing zeros.

[4] Rounding toward zero ensures property 2 holds, but can produce more rounding error than rounding toward nearest. The maximum log relative error (LRE) for the former is *(digits-1)* while the maximum LRE for the latter is *digits*. Hence, you may wish to use one more significant digit when computing UNF's than when reporting rounding significant digits for presentation or storage.

(e) A lower case 'e'

(f) A sign character.

(g) The digits of the exponent, omitting trailing zeros.

(For example, the number pi, at five digits, is represented as "-3.1415e+" and the number 300 is represented as the string "+3.e+2")

4. If the element is missing it is represented as a string of three null characters. If the element is a IEEE 754 non-finite floating point special value, it is represented as the signed lower-case IEEE minimal printable equivalents (i.e., +inf,-inf, +nan).

5. Character strings representing non-missing values are terminated with a POSIX end-of-line character.

6. Each character string is encoded in the UTF32BE Unicode bit encoding. [see Unicode 2003] bit encoding.

7. The vector of character strings is combined into a single sequence, with each character string separated by a POSIX end-of-line characters and a null byte. A hash is computed on the resulting sequence. Version 4 uses SHA256 [NIST 2002] as the hashing algorithm. (Versions one through three use a 64-bit checksum, a 64-bit CRC, and MD5, respectively, as the hashing algorithm.)

8. The resulting hash is then base64 encode [see Josefson 2003] for printing.

Character values are treated in an identical way, except that they are truncated to k characters rather than rounded to k digits, and that the default k for characters is 128.

UNF's can be combined to fingerprint higher-level data objects. For example, the UNF's for multiple variables can be combined to form a UNF for the entire data frame, and UNF's for a set of data frames can be combined to form a single UNF representing an entire research study. The `summary()` function performs such combinations automatically, using the following algorithm:

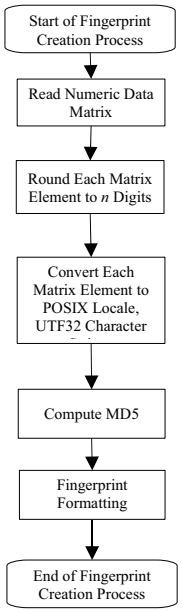
1. Calculate the UNF for each lower level data object.

2. Sort the base-64 representation of UNF's in POSIX locale sort order.

3. Apply the UNF algorithm to the resulting vector of character strings using k at least as large as the length of the underlying character string.

This requires only that you use a consistent version and level of precision across the individual UNF's being combined.

Figure 1: Stages of UNF Generation



4.3 Using UNF's

Here we describe usage of UNF library in **R**.

The 'unf' function returns a UNF object which can be converted using 'as.character' to a signature string. For example:

```
> library(UNF)
> v = 1:100/10 + 0.0111
> print(unf(v, ndigits = 7))

[1] "UNF:4:7,128:6kK46s059g5dswiRGBM7yVvo3gwyBVvuBzioK/df72o="
```

This representation is self-identifying – it identifies the string as a fingerprint ('UNF') , the version of the algorithm (4), and the number of significant digits uses for numeric and character values. (The number of significant digits can be omitted if the default for that version of the algorithm is used.) The segment following the final colon is the actual fingerprint in base64 encoded format (the equal signs are part of the encoding, and represent padding to a 24-bit boundary).

To compare two UNF's, or sets of UNF's, one usually wants to compare only the base64 portion. Use 'unf2base64' for this, which will extract the base64 portion for comparizon. [5]

```
> vr = signifz(v, digits = 2)
> unf2base64(unf(v))

[1] "6kK46s059g5dswiRGBM7yVvo3gwyBVvuBzioK/df72o="

> unf2base64(unf(v)) == unf2base64(unf(vr))

[1] FALSE

> unf2base64(unf(v, digits = 2)) == unf2base64(unf(vr))

[1] TRUE
```

Use 'summary' to produce a single UNF from set of vectors, by computing a new UNF across the sorted base64 strings.

```
> data(longley)
> mf10 <- unf(longley)
> print(mf10)
```

[5] Note that in the example below, we supply a function `signifz()` to compute significant digits using 'round towards zero mode' , since the *R* function `signif()` does not use this rounding mode.

```
[1] "UNF:4:6,128:8jZLmPa5mIIVxfM6ymFuafh6oGh8pPFu0vkHlDA6xto="
[2] "UNF:4:6,128:2xyGCHarwYSklOdTCDEgji76jv0waNmhMUB0IBm7R9s="
[3] "UNF:4:6,128:kncfvYKelMyyeAGDAN2A4SbnCO8wbRlbFLUZ0TyWN/8="
[4] "UNF:4:6,128:Bhuo94R5otkKNuJzxl+Ee1RxTWIC4al8ajv33aiwLz8="
[5] "UNF:4:6,128:9PnzHkZCJeNk4YZBRHdZ/N+oHXnPWyWyPokZf5ad5UI="
[6] "UNF:4:6,128:UBJuBYYt+QOXjTBU5V+tSV2euyfthhR/qmDBM18IzCY="
[7] "UNF:4:6,128:vTjA4Xagiwfhz4HDhx0tGLJYBdK9judYZK5eXdRRNcQ="

> summary(mf10)

[1] "UNF:4:6,128:7zq5Q8/mP7z3m2E+mwoOJndVM8flQmmbuHvvqDK910E="
```

UNF libraries are also available for standalone use, for use in C++, and for use with other packages, such as **Stata**.

5 Other Tools for Accurate and Reliable Statistical Computing

In addition to the perturbation tests and universal numeric fingerprints described above, the *Accuracy* module provides a number of other tools for more accurate statistical computing. Among these is a set of functions to collect true random numbers, and a generalized cholesky method that can be used to extract information out of non-invertible Hessians.

5.1 True Random Numbers

'Random' numbers aren't. The numbers provided by routines such as runif() are not genuinely random. Instead, they are *pseudo-random number generators* (PRNGs), deterministic processes that create a sequence of numbers. Pseudo-random number generators start with a single "seed" value (specified by the user or left at defaults) and generate a repeating sequence with a certain fixed length, or period p. This sequence is statistically similar, in limited respects, to random draws from a uniform distribution.

The earliest PRNGs, still in use in some places, and used in early versions of R, is the Linear Congruential Generator (LCG), which is defined as:

$$\begin{aligned} LCG(a, m, s, c) \equiv & \\ & x_0 = s, \\ & x_n = (ax_{n-1} + c) \bmod m. \end{aligned} \tag{3}$$

(All parameters are integers, and in practice x is usually divided by m to yield numbers between zero and one.)