Using Lisp Macro-Facilities for Transferable Statistical Tests

Kay Hamacher
Dept. of Computer Science, Dept. of Physics, Dept. of Biology
Schnittspahnstr. 10
64287 Darmstadt, Germany
hamacher@bio.tu-darmstadt.de

ABSTRACT

Model-free statistical tests are purely data-driven approaches to assess correlations and other interdependencies between observable quantities. The few, distinct patterns how to perform these tests on the myriad of potentially different interdependence measures prompted us to use (Common) Lisp's macro capabilities for the development of a general, domain-specific language (DSL) of expectation values under so-called resampling techniques. Herein, we give an introduction into this statistical approach to big data, describe our solution, and report on application as well as on further research opportunities in statistical DSLs. We illustrate the results based on a toy example.

CCS Concepts

•**Computing methodologies** → **Symbolic and algebraic manipulation;** Shared memory algorithms; •**Software and its engineering** → **Domain specific languages;** •**Mathematics of computing** → *Nonparametric statistics; Statistical software;*

Keywords

Macros; Domain Specific Language; Statistical Modeling; Permutation Test

1. INTRODUCTION

Data analysis relies heavily on statistical tests. For example, in the traditional frequentist approach such procedures test, e.g., the significance under a null hypothesis (shortened to "the null"). In some cases, one cannot or does not want to formulate a (rather involved) null, but rather relies on a completely data-driven, model-free approach – so called parameter-free statistics[1]

[1]In contrast to parameter statistics: here, an explicit model and its respective parameters are fitted to the data and the significance is judged on the fitting outcome.

9th ELS May 9–10, 2016, Kraków, Poland

Then the test is often carried out under so-called *resampling*. This avoids the need for knowledge of the underlying probability distribution of any test statistic. Resampling techniques can be distinguished based on the sample of the available data that they create and thus by the underlying question to be answered:

Jackknifing is capable of estimating variance and bias in a test statistic [15]. Here, we take the original data and delete d many entries from it, recompute the test statistic. Upon repetition we get a quantification of the sensitivity of the test statistic under finite-size effects. The parameter d characterizes the procedure.

Bootstraping creates an approximate distribution of the test statistic [6]. To this end, one draws random samples from the test data repeatedly with replacement.

Permutation Tests have a long tradition [5]. Here, one shuffles different data columns randomly in order to destroy potential correlations. This procedure is therefore able to assess correlation in the original data on the background of uncorrelated data via permutations/shuffles.

Typically, one is interested in the distribution of expectation values over a (resampled) data set (under any of the above procedures). The resampling procedure then computes a function – the test statistic – of the entries in the original data set first. Then, within a loop, we compute the same function over and over again over the resampled data set and obtain a collection of function values. This collection is what we are interested in: a computational approximation of the underlying, real distribution of function values. With respect to this distribution, we can then assess the "relevance"[2] of the function value for the original, un-sampled data set. Note, that the three methods above are the "basic" variants with a lot of variations in domain-specific applications; thus, a general framework to allow for easy implementation of variants is desirable – begging for a DSL.

1.1 Notation

In the subsequent part, we will deal with real-valued matrices $\mathcal{X}$ where the columns contain observations or measurements of different variables and each row $\vec{x_i}$ is one measurement of all the variables simultaneously. Note, that time

[2]significance in statistical parlance

series analysis for, e.g., auto-correlation is possible by copying a time-lagged version from one column of the matrix into another.

A frequently employed notation for expectation values over a sample is

$$\left\langle f\left(\tilde{\mathcal{X}}\right)\right\rangle = \frac{1}{N_\sigma}\sum_{i=1}^{N_\sigma} f\left(\vec{\tilde{x}}_{\sigma(i)}\right) \quad (1)$$

The sampling function contained in the functions $\sigma(i) \to j$ and $\mathcal{X} \to \tilde{\mathcal{X}}$ implements the three procedures (permutation test, jackknife, bootstrap) from above. Some procedures – such as the jackknife – might change the number of data points from N to N_σ.

1.2 Related Work

First, there was a package for Lisp implementing general statistical methods called `XLISP-STAT`. This system was, however, abandoned by the proponents [4]. The main reason as discussed in the cited paper lies in the fact that a shift in the user community was recognized; eventually favoring `R`. Although we also employ `R` frequently, it has become apparent since the 90s that it is too much a compromise; on the surface you can be object-orientated, functional, even macros are (somewhat) possible. But in the end, neither of these traits is implemented thoroughly, not to speak about performance issues. Thus, the motivation [4] to abandon `XLISP-STAT` might have been in the light of the 90's a good one, our experience tells us otherwise in the meantime. Especially for prototyping computational procedures by (semi-)experts a new, more expressive and computational efficient technology needs to be employed.

Then, there exists some code [2] that implements statistical procedures and methods or interfaces to external systems like `R` [13]. Note, however, that this package implements concrete procedures, rather than a macro-based, general framework that automatically creates (Lisp-)code for *any conceivable* test statistic. Another attempt on parameter statistics [12] seems to have been abandoned.

Furthermore, several software packages outside the Lisp ecosystems are available, that follow the same, traditional, procedure-orientated lines: implementing resampling techniques but without any notion of domain-specific language that allows for the automatic generation of resampling for a novel or user-defined measure to work with. The most prominent ones are `R` [11], `Julia` [3] and `(I)python` [10]. While these languages are capable of self-introspection of code and `Julia` has a macro facility similar to Lisp-like languages they are all, however, *not homoiconic* to the extent of Common Lisp. `Julia` is the only language converging to the capabilities of Lisp. Thus, almost all these language lack the ability to mix code fragments of a domain specific language (DSL) for resampling as well as standard mathematical expressions of that language.

Work on DSLs in general has a long history [1] – eventually being strongly interwoven with the history of Lisp [7, 9] A full review is beyond the scope of the present work but it is fair to say, that quite a lot of work in the realm of software engineering has be done using DSLs. Most of this focus on one special question and thus is barely related to the concrete question on resampling procedures we address.

1.3 Our Contribution

We have implemented a general framework for resampling techniques. The data randomization and sampling procedure can be easily replaced by any function coherent to the interface of existing procedures. We will illustrate this in Sec. 2.1.

Our framework consists mainly of two macros that expand nested expectation values of expressions which implement any conceivable, real-valued function f in Eq. 1. By this, we can now easily write "statistical formulas" that contain results from resampling procedures.

We illustrate this by implementing and applying a traditional measure (covariance) to a multi-dimensional, dynamic system that produces synthetic data (cmp. Sec. 3.1) for test purposes, see Sec. 3.

2. LISP TO THE RESCUE: MACROS FOR PERMUTATION TESTS

Above, we have described how permutation tests show a repeated pattern: iterating several times over (partially) mixed choices from an underlying data set, each time recalculating a particular measure.

This pattern could potentially be applied to any code that implements such a measure f of Eq. 1 – as long as this pattern can access and introspect the code of the measure and "program the program code" to rewrite itself to implement this pattern again and again for each user-/programmer-supplied pattern – the realm in which Lisp excels due to this inherent macro capabilities. To this end, we define here a domain specific language that describes how a code fragment - namely a function uses data – and thus makes the measure accessible to a macro implementing any of the above discusses resampling approaches.

Formulas for a user-provided f in Eq. 1 need to refer to data elements contained in the (resampled) data set to perform their computations. We introduce a notation to this end that is motivated by the dataframe syntax of the statistical language `R`. In our DSL we make the data entry of any row i of the full data set $\tilde{\mathcal{X}}$ available as `D$`i where i is a string or a number serving as a "name" for a particular column.

Thus, we are able to write a measure, e.g., $f = x_a \cdot x_b$ as a form `(* D$a D$b)`. The macro needs then to expand this to a combination of columns a and b. Note, that in practice we are always interested in all pairings (a, b) of columns in the data set. Therefore, `D$1` does *not* refer to the first column of the data, but rather represents the first column index currently under investigation.

Following Peter Seibel's suggestion [14] to first write down what a macro needs to achieve, we illustrate here how the resampling of a covariance $\langle(D\$a - \langle D\$a\rangle)\cdot(D\$b - \langle D\$b\rangle)\rangle$ between any pair of columns should be implemented:

```
(with-resampling test-dataframe 100
      #'permutation-test
      expectation-value
       (*
         (- D$a (expectation-value D$a))
         (- D$b (expectation-value D$b))))
```

Here `test-dataframe` is an array with test data for the model in Sec. 3.1. The function `permutation-test` is described in Sec. 2.1.

We achieve this by the macro `with-resampling` (shown in Algorithm 1) that first 1) extracts all `D$x` symbols, 2) sets up

iterators over the combination of columns to be combined in the abstract syntax tree (AST) implementing the user's f, and 3) returns an array with the values of f for the original data and for the resampled ones.

While we could have implemented `with-resampling` as a function we took the deliberate design decision to implement it as a macro: we hope to accommodate future extension such as user-provided aggregation function (histogram building, for example) beyond the simple expectation value with this step and make the procedure more widely applicable.

Within the resampling procedure we need to parse the AST of f and insert appropriate code for expectation value computations.

To this end, we implement `parse-ev-calls` as a macro to make the current, looped-over `indices` of the sampled data set available to any formula internal to its respective macro invocation. We show in Algorithm 2 the concrete implementation for the expectation value. Thus a code fragment like above can be converted to an expression in which `D$`a and `D$`b are replaced by respective `aref`s to the resampled data.

Note, that we cannot naively use `subst`, `subst-if`, and similar facilities to walk the AST tree[3] : we must not substitute `D$`x symbols at different levels of nesting of `expectation-value` occurrences. One could achieve this by a rather involved predicate definition, but we decided to achieve the same result by an appropriate base case in the recursive definition of our `parse-ev-calls` macro.

2.1 Implementing Sampling Variants

We show in Algorithms 3, 4, 5, and 6 the implementations of the permutation test, the jackknife, and the bootstrap with a shared interface.

Furthermore, we have implemented the `identity-sample` function, which is a `permutation-test` without shuffling at all – thus we are able to compute the test statistic for the original data using `identity-sample`. Later on, we will illustrate the usefulness of this detail and the necessary redundancy in `identity-sample`.

Note, the subtle differences in the argument list between `identity-sample`, `bootstraping`, `identity-sample` on the one hand and `jackknifing` on the other: `jackknifing` needs an additional parameter d. We can, however, easily have a concrete jackknife for a fixed value of d conforming to the common parameter pattern by employing a lexical closure as in, e.g.,

```
1 (defun jack2 (idxs n)
2   (let ((d 2))
3     (jackknifing idxs n d)))
```

3. RESULTS

All tests were run on a machine under Linux Kernel version 4.3.3, SBCL 1.3.1.

3.1 Test Data

To apply our package we created synthetic data[4] for a system of three coupled dynamic variables with complex dynamics:

$$
\begin{aligned}
f(x) &= 4 \cdot x \cdot (1-x) && (2)\\
a_{i+1} &= 0.5 \cdot f(a_i) + f(a_{i-3}) \\
x^{(1)}_{i+1} &= f\left(x^{(1)}_i\right) \\
x^{(2)}_{i+1} &= 0.8 \cdot f\left(x^{(2)}_i\right) + 0.2 \cdot f\left(x^{(1)}_i\right) \\
x^{(3)}_{i+1} &= 0.5 \cdot f\left(x^{(3)}_i\right) + 0.25 \cdot f\left(x^{(2)}_i\right) + 0.25 \cdot f\left(x^{(1)}_i\right) && (3)
\end{aligned}
$$

The function f is the logistic equation in the chaotic regime. Therefore, the series of $x^{(1)}$ and a values are independent, chaotic trajectories. At the same time, $x^{(2)}$ and $x^{(3)}$ are coupled instantaneously to the driving system $x^{(1)}$ and thus should show correlaction to $x^{(1)}$. We simulated the dynamics for $i \in [1 \ldots 500]$. A embedding plot of the trajectory for the three components $x^{(1)}$, $x^{(2)}$, $x^{(3)}$ is shown in Fig. 1. The trajectory of a serves as an example to which the other values $x^{(1)}$, $x^{(2)}$, $x^{(3)}$ cannot be correlated and thus any metric must vanish and/or be insignificant. Initial condition were chosen by a random generator.

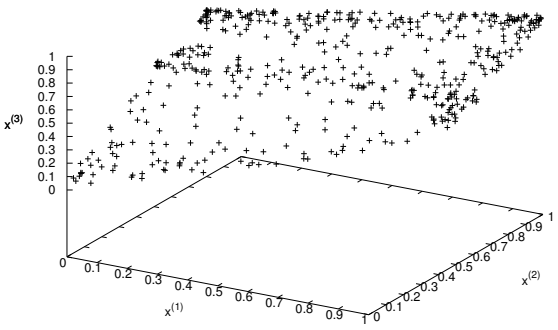

Figure 1: The 3D embedding of the time series $\left(x^{(1)}; x^{(2)}; x^{(3)}\right)$. Clearly each pair of variables covary, but also the 3-tupel of all three variables shows rich dynamics and interdependence.

3.2 An Example : Covariance

As an example we show the covariance between two data vectors $\vec{X} = (X_1, \ldots X_N)$ and $\vec{Y} = (Y_1, \ldots Y_N)$ defined as

$$
\begin{aligned}
\operatorname{covar}\left(\vec{X}, \vec{Y}\right) &:= \frac{1}{N} \sum_{i=1}^{N} \left(X_i - \bar{X}\right) \cdot \left(Y_i - \bar{Y}\right) && (4)\\
\text{with} \qquad & \bar{X} = \sum_i X_i \quad \text{and} \quad \bar{Y} = \sum_i Y_i.
\end{aligned}
$$

$\vec{X}$ and $\vec{Y}$ are any pairs of columns in the data set created in Sec. 3. We demanded above from our DSL this to be implementable with the code

```
1 (with-resampling test-dataframe 100
2       #'permutation-test
```

[3] as, e.g., suggested at listips.com webpage

[4] Note, that for illustration purposes the used data set is irrelevant; we have, however, opted for a system under our complete control to be able to distinguish, e.g., spurious correlations from real ones ($a \leftrightarrow x^{(1)}$ and e.g. $x^{(2)} \leftrightarrow x^{(1)}$, respectively).

Algorithm 1 The macro implementing the "frame" for calling general AST representing a measure f. `multi-dim-homogenous-iter` is a CLOS-class that iterates over a regular, multi-dimensional grid of column indices (eventually the ones represented be the a, b, c, ... in the `D$`x terminals present within the AST). This mapping of a, b, c, ... to real column numbers is done via a hash table that is modified by `modify-hash-table`. We omit several utility functions for brevity here, such as `repetition`, ...

```
1  (defmacro with-resampling(data N method &rest ast)
2    `(let* ((arr-dims (array-dimensions ,data))
3            (dimens (cadr arr-dims))
4            (idxs (from-zero-to (car arr-dims)))
5            (indices (identity-sample idxs dimens)))
6       (multiple-value-bind (Ds data-hash-table) (extract-ds ',ast)
7         (let* ((grid (repetition Ds dimens))
8                (z (make-array (append grid (list (1+ ,N))) :initial-element 0.0)))
9           (loop for nn from 0 to ,N do
10                (let ((idx-iterator (make-instance 'multi-dim-homogenous-iter
11                                                   :dimensionality Ds :N dimens)))
12                  (loop while (not (donep idx-iterator)) do
13                        (let ((idx (next idx-iterator)))
14                          (setf data-hash-table
15                                (modify-hash-table data-hash-table idx))
16                          (let ((w (parse-ev-calls data-hash-table indices ,data ,@ast)))
17                            (setf (apply #'aref z
18                                         (append (coerce idx 'list) (list nn)))
19                                  w)))))
20                (setf indices (funcall ,method idxs dimens)))
21           (values z)))))
```

Algorithm 2 A macro implementing the DSL-specific keyword `expectation-value`. Note, how we access the data set elements via an array of (sampled) indices and craft an S-expression in line numbers 7-9 to access those elements. We traverse the abstract syntax tree (AST) recursively.

```
1  (defmacro parse-ev-calls(Dsht indices data &body ast)
2    (labels ((walk (Dsht i d en runID trafo-ast)
3               (cond ((null en) trafo-ast)
4                     ((atom en) (if (and (symbolp en)
5                                         (start-with-D$p en))
6                                    (let ((IDX (gensym)))
7                                      (append `(let ((,IDX (gethash ',en ,Dsht)))
8                                                 (aref ,d (aref ,i ,runID ,IDX) ,IDX))
9                                              trafo-ast))
10                                   (if (eq en 'quote)
11                                       trafo-ast
12                                       (cons-non-nil en trafo-ast))))
13                    ((listp en)
14                     (if (and
15                          (symbolp (car en))
16                          (eq (car en) 'expectation-value) )
17                         (let* ((runID2 (gensym))
18                                (cont (walk Dsht i d (cdr en) runID2 trafo-ast)))
19                           (concatenate 'list `(let((r 0.0)
20                                                    (II (car (array-dimensions ,indices))))
21                                                 (dotimes (,runID2 II) (incf r ,@cont))
22                                                 (/ r II) ) trafo-ast))
23                         (mapcar #'(lambda(x)
24                                     (walk Dsht i d x runID nil)) en))))))
25      (let ((runID (gensym)))
26        (walk Dsht indices data ast runID nil))))
```

Algorithm 3 The identity mapping of indices – here, redundancy is necessary to easily implement the permutation test later on.

```
(defun identity-sample (idxs n)
  (let* ((rowMax (coerce (length idxs) 'number))
         (dims (list rowMax n))
         (current (coerce idxs 'vector))
         (z (make-array dims :initial-element 0)))
    (loop for j from 0 to (1- rowMax) do
         (loop for i from 0 to (1- n) do
              (setf (aref z j i) (aref current j))))
    (values z)))
```

Algorithm 4 Mapping of indices under the permutation test – each column is shuffled individually to destroy potential correlations. `nshuffle` implements Knuth's shuffling procedure.

```
(defun permutation-test (idxs n)
  (let* ((z (identity-sample idxs n)) ; initialize with identity
         (rowMax (coerce (length idxs) 'number))
         (current (coerce idxs 'vector)))
    (loop for i from 1 to (1- n) do ; shuffle all but the first column
         (setf current (nshuffle current))
         (loop for j from 0 to (1- rowMax) do
              (setf (aref z j i) (aref current j)))
         finally (return z))))
```

Algorithm 5 The bootstrap – indices are drawn randomly and might occur several times in the created index array.

```
(defun bootstraping (idxs n)
  (let* ((rowMax (coerce (length idxs) 'number))
         (dims (list rowMax n))
         (current  (coerce idxs 'vector))
         (z (make-array dims :initial-element 0))
         (w 0))
    (loop for j from 0 to (1- rowMax) do
         (setf w (random rowMax))
         (loop for i from 0 to (1- n) do
              (setf (aref z j i) (aref current w)))
       finally (return z))))
```

Algorithm 6 Jackknifing – d many, randomly chosen samples need to be omitted in this procedure.

```
(defun jackknifing (idxs n d)
  (let* ((rowMax (- (coerce (length idxs) 'number) d))
         (dims (list rowMax n))
         (current (nshuffle (coerce idxs 'vector)))
         (z (make-array dims :initial-element 0)))
    (loop for j from 0 to (1- rowMax) do
         (loop for i from 0 to (1- n) do
              (setf (aref z j i) (aref current j)))
       finally (return z))))
```

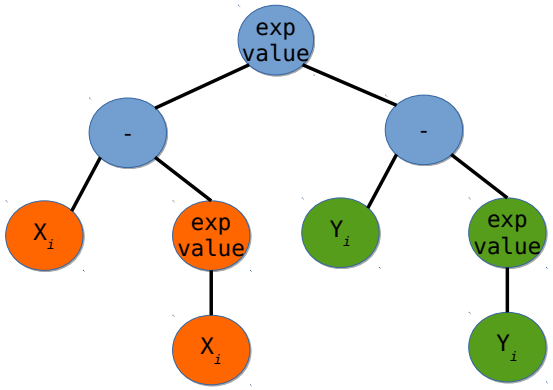

Figure 2: Tree-representation of the S-expression for the covariance in Eq. 4. The data vectors X and Y are used several times.

	a	$x^{(1)}$	$x^{(2)}$	$x^{(3)}$
a	0.0667	0.0660	0.0056	-0.0006
$x^{(1)}$	-0.0001	-0.0003	0.0009	0.0029
$x^{(2)}$	0.0056	-0.0005	0.1323	0.1323
$x^{(3)}$	0.0395	-0.0066	0.0462	0.0026

Table 1: Covariance values for the *original* data from Eq. 2. Note, that `parse-ev-calls` was expanded so that a and b in `D$`a and `D$`b took on all the combinations of columns, e.g., $\left(a = a, b = x^{(1)}\right)$; $\left(a = x^{(3)}, b = x^{(2)}\right)$; and so forth.

```
3       expectation-value
4        (*
5          (- D$a (expectation-value D$a))
6          (- D$b (expectation-value D$b))))
```

which is illustrated in Fig. 2. For brevity, we cannot show the full macro-expansion here, but publish it on the WWW[5]. The length and complexity clearly shows the benefits of a DSL. Furthermore, one can see in the macro-expanded code that no local variables other than the ones generated via `gensyms` exist, so no variable capture can occur.

When we apply our system to the four-dimensional test system from above we obtain the resulting covariance matrix between all variables in Tab. 1.

From the covariance alone we cannot judge on the influence of any variable onto the other. Eventually, we must find no connection between a and any of the xs as they are independent in Eq. 2.

Applying the permutation test with 100 repetitions we obtain the covariance values for 100 shuffled data sets. From this we can compute the (one-sided[6]) percentile as the fraction of covariance matrix entries that turned out to be smaller in the permutation test than for the original data. We obtain the output

	a	$x^{(1)}$	$x^{(2)}$	$x^{(3)}$
a	0.04	0.92	0.44	0.53
$x^{(1)}$	0.92	0.54	**1.00**	**1.00**
$x^{(2)}$	0.44	**1.00**	0.80	**1.00**
$x^{(3)}$	0.53	**1.00**	**1.00**	0.76

in which we marked the statistic insignificant results under a one-sided test in red. For significance we apply Fisher's well-known criterion of a so-called p-value smaller than 5% (or a percentile of larger than 0.95). Our permutation test based assessment shows, that we cannot find a statistically significant covariance between a and and any of the xs variables[7].

4. OUTLOOK

Above we have demonstrated the first development of a DSL for statistical resampling technqiues. We motivated our choice and described the used macros as well as their rationale. We applied it to a test problem and illustrated the necessity of such involved resampling techniques as otherwise one might attribute wrongly dependencies between variables obtained from stochastic processes.

Although our system is at present slower for simple measures like the covariance in comparison to manually tuned code like the ones we published for statistics using GPUs [16], we have with the present work laid ground for a *general* system for prototyping, interactive data science, and hypothesis generating. These steps are increasingly necessary in the realm of big data as their might be high-dimensional correlations present for which one cannot always hand-craft individual solutions. Rather, one can rely on Lisp's macro to do the job. At present, our contribution enables data scientists to implement general measures and their resampling tests easily and fast, with – at present – costs in performance.

As this approach is work in progress, several improvements will be implemented and researched in the future. In particular, we hope to encourage participation of the larger Lisp community on these issues:

Parallelization the resampling procedures laid out above are all data-parallel in the columns of a a dataframe. The iterator over those columns in Algo. 1 is thus "embarrassingly parallel" [8], begging for parallelization via, e.g., the `lparallel` library.

Sampling variants The implemented algorithms 4, 5, and 6 have a common pattern, but also some subtle differences. It seems to be promising find at an abstract level a macro to implement any conceivable resampling/reshuffling technique. Furthermore, in the literature all three resampling techniques can be found in variants; thus a DSL is no overkill, but rather a first step to offer a general framework.

User-provided aggregation mechanisms The expectation value as a sum over samples as in Eq. 1 is the

[5] http://www.kay-hamacher.de/macro-expanded.lisp

[6] a one-sided test tests for original data to be larger or smaller than the resampling ensemble; a two-sided test would test for the *absolute value* to be smaller.

[7] For the diagonal entries: as this a degenerated case, we compute the variance of a variable that does not change under the permutation test, thus we can – by construction of the test – not obtain any significant values.

predominant procedure in statistics. It aggregates the function values of a measure over a randomized sample into the arithmetic mean. This is, at present, hard-coded into the macro `parse-ev-calls`. Still, other aggregation procedures are also conceivable. We will extend the package to provide for any user-provided mechanism.

Auto-generation of measures On can use Lisp S-expressions and Lisp's ability to modify these to "evolve" ASTs implementing an appropriate f via, e.g., genetic programming.

Caching / Memoization The AST representing the measure f to be evaluated might be a rather complex, time-consuming function. Here, memoization techniques are one way to cope with this; this route will be taken in the future development of our system.

A first version of the package is made available on the web under http://www.kay-hamacher.de/Software/Resampling_Lisp.tar.gz.

5. ACKNOWLEDGMENTS

The author gratefully acknowledges financial support by the LOEWE projects iNAPO & compuGene of the Hessen State Ministry of Higher Education, Research and the Arts. This work was also supported by the German Federal Ministry of Education and Research (BMBF) within CRISP.

6. REFERENCES

[1] IEEE Transactions on Software Engineering (TSE), special issue, volume 25, number 3, may/june 1999.

[2] S. D. Anderson, A. Carlson, D. L. Westbrook, D. M. Hart, and P. R. cohen. Common lisp analytical statistics package: User manual. Technical report, Amherst, MA, USA, 1993.

[3] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah. Julia: A fresh approach to numerical computing. November 2014.

[4] J. de Leeuw. On abandoning XLISP-STAT. *Journal of Statistical Software*, 13(1):1–5, 2005.

[5] M. Dwass. Modified randomization tests for nonparametric hypotheses. *Ann. Math. Statist.*, 28(1):181–187, 03 1957.

[6] B. Efron. Bootstrap methods: Another look at the jackknife. *Ann. Statist.*, 7(1):1–26, 01 1979.

[7] P. Graham. *ANSI Common LISP*. Prentice Hall, Nov. 1995.

[8] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 1 edition, Mar. 2008.

[9] D. Hoyte. *Let Over Lambda*. 2008.

[10] F. Pérez and B. E. Granger. IPython: a system for interactive scientific computing. *Computing in Science and Engineering*, 9(3):21–29, May 2007.

[11] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2008. ISBN 3-900051-07-0.

[12] T. Rossini. *Common Lisp Statistics*. https://github.com/blindglobe/common-lisp-stat, retrieved 03/15/2016.

[13] T. Rossini. *R-Common Lisp gateway*. https://github.com/blindglobe/rclg, retrieved 03/15/2016.

[14] P. Seibel. *Practical Common Lisp*. Apress, Sept. 2004.

[15] J. W. Tukey. Bias and confidence in not-quite large samples. *Ann. Math. Statist.*, 29(2):614–623, 06 1958. just the abstract to a talk.

[16] M. Waechter, K. Jaeger, D. Thuerck, S. Weissgraeber, S. Widmer, M. Goesele, and K. Hamacher. Using graphics processing units to investigate molecular coevolution. *Concurrency and Computation: Practice and Experience*, 26(6):1278–1296, 2014.