S, R, and Data Science

JOHN M. CHAMBERS, Stanford University, USA

Shepherd: Jean-Baptiste Tristan, Oracle Labs, USA

Data science is increasingly important and challenging. It requires computational tools and programming environments that handle big data and difficult computations, while supporting creative, high-quality analysis. The R language and related software play a major role in computing for data science. R is featured in most programs for training in the field. R packages provide tools for a wide range of purposes and users. The description of a new technique, particularly from research in statistics, is frequently accompanied by an R package, greatly increasing the usefulness of the description.

The history of R makes clear its connection to data science. R was consciously designed to replicate in open-source software the contents of the S software. S in turn was written by data analysis researchers at Bell Labs as part of the computing environment for research in data analysis and collaborations to apply that research, rather than as a separate project to create a programming language. The features of S and the design decisions made for it need to be understood in this broader context of supporting effective data analysis (which would now be called data science). These characteristics were all transferred to R and remain central to its effectiveness. Thus, R can be viewed as based historically on a domain-specific language for the domain of data science.

CCS Concepts: • **Software and its engineering** → **General programming languages**; • **Social and professional topics** → *History of programming languages*.

Additional Key Words and Phrases: data science, statistical computing, scientific computing

ACM Reference Format:
John M. Chambers. 2020. S, R, and Data Science. *Proc. ACM Program. Lang.* 4, HOPL, Article 84 (June 2020), 17 pages. https://doi.org/10.1145/3386334

1 INTRODUCTION

R has become a widely used medium for the practice of technically advanced data science; most importantly, a medium in which new applications and new ideas in the practice of data science are very often shared throughout the worldwide community.

The language, data structure and functional capabilities of R, as they were implemented in the late 1990s, were modelled on the S software from Bell Labs, supplemented by some new ideas, reflecting developments in programming language design during this period. To create a free, open-source language based on S, R's original authors, [Ihaka and Gentleman 1996], were joined by an international group of volunteers, subsequently known as *R Core* [Ihaka 1998].

The necessary definition of S, independent of its proprietary implementation, was taken from two books: [Becker et al. 1988] for the general features and [Chambers and Hastie 1992] for the fitting and analysis of statistical models plus some extensions to the software, notably to classes

Author's address: John M. Chambers, Statistics Department, Stanford University, 390 Serra Mall, Stanford, California, 94305, USA, jmc@stat.stanford.edu.

This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.
© 2020 Copyright held by the owner/author(s).
2475-1421/2020/6-ART84
https://doi.org/10.1145/3386334

and methods. In the R community, these books are nearly always referred to as the *blue book* and the *white book*, respectively, from their covers.

The S software was distinguished from many programming language designs in being motivated by a relatively specific scientific goal; namely, to support research in data analysis at Bell Labs and applications to challenging problems. The goal of the language was to provide interactive analysis using the best current techniques [Becker and Chambers 1984] and also a programming interface to software implementing new techniques [Becker and Chambers 1985]. These goals influenced distinctive characteristics of S: data structures designed for data analysis rather than built up from basic types; interfaces to other software as part of the language design.

Data analysis as practiced at Bell Labs is recognized as the precursor of what would now be described as "data science" [Donoho 2017]. Subsequent versions of S and of R have retained and extended a design focussed on the needs of data science, so that R can be viewed as a *domain-specific* language for the domain of data science.

This perspective helps to understand both the history and many of the design choices leading to R.

2 1965–1985: BELL LABS, DATA SCIENCE AND COMPUTING

For a long period in the 20th century, particularly for the three or four decades following the second world war, notable scientific and technological advances came from the research area of AT&T's Bell Telephone Laboratories (aka "Bell Labs").

The most famous of these, the invention of the transistor, was a critical step towards the digital and miniaturization revolution that continues to overwhelm us. Other advances were also key, notably information theory, coding and digital techniques for communication.

It was noted at the time, and even more since then, that the productivity and originality of much Bell Labs work seemed to derive from an organization and research atmosphere not easily found elsewhere. A non-technical account that nevertheless conveys much of the research management style is given in the book by Jon Gertner[Gertner 2013].

The management philosophy of Bell Labs research was to hire bright and self-motivated individuals and give them the freedom to come up with their own ideas. At the same time, there was a belief that some of these ideas would be fundamentally valuable for communication, and so for the parent company. Whoever came up with such ideas could expect to be rewarded financially (moderately) and with recognition. Along with ideas leading to the transistor and communication theory, this research environment nurtured an approach to what can now be called data science.

2.1 Data Science and Data Analysis

Techniques, applications and teaching for data science have drawn much attention recently, and for good reason. Essentially all branches of science face challenges in studying important questions due to the quantity, complexity or questionable nature of the data.

The term "data science" is relatively recent and is used somewhat loosely at times; we will assume a simple but strict definition:

> *Data science consists of techniques and their application to derive and communicate scientifically valid inferences and predictions based on relevant data.*

(In particular, just the use of "big data" does not qualify the results as data science.)

Although the popularity of the term lay decades in the future, research in data analysis at Bell Labs during the design and evolution of S is widely recognized as the precursor to data science. The fundamental inspiration for this research came originally from John Tukey. The historical summary in the paper "50 Years of Data Science" [Donoho 2017] cites his "Future of Data Analysis" paper

[Tukey 1962] as a point of origin for data science. Tukey's championing of data analysis continued through many later contributions, including the book "Exploratory Data Analysis" [Tukey 1977] and beyond. His career was divided between Bell Labs and Princeton University (along with many other activities). Tukey was an enormous influence, not to say inspiration, at Bell Labs.

Bell Labs statistics research was housed in the "Statistics and Data Analysis Research" department, surely the only group of research statisticians with "Data Analysis" in its title at that time. Interesting and potentially rewarding projects could range from the essentially theoretical (though usually with an implication of future application) through more data-analytic methods (for example, data or model visualization [Wilk and Gnanadesikan 1968]) to collaborative projects with other groups at Bell Labs and AT&T to obtain insights from particular sources of data.

Data analysis at Bell Labs did not avoid "big data" by the standards of the time (usually meaning one or a few reels of magnetic tape); on the contrary, the challenge of doing analysis in this context was often central to a particularly interesting and important collaboration. For example, rain gauge experiments in the 1960s studying the effect of rainfall on errors in microwave transmission generated data running to several million observations, requiring some "big data" techniques for visualization and summaries ([Freeny and Gabbe 1969] and [Jaeckel and Gabbe 1974]).

Overall, the combination of opportunities and responsibilities gave data analysis at Bell Labs much of the flavor associated with contemporary discussions of data science: large datasets; iterative, probing analysis including visualization; problems of practical importance and, as a result, challenging computations. Data analysis that was useful and applicable to sizable datasets required advanced computational techniques for the time and good software to implement them.

By the time I first arrived at Bell Labs as a graduate student intern in 1964, advances in computation were already recognized as important for data analysis. S came after more than a decade of involvement in statistical computing.

2.2 Before S

The decade or so beginning in the mid-1960s was a determining period for scientific computing, and in particular for computations involving significant amounts of data. Hardware, software and algorithms all broke decisively with the first generation of computing and its emphasis on the physical elements of the computer and individual machine instructions.

By the middle of the 1960s, Bell Labs was involved in a project to create the Multics system, jointly with MIT and General Electric [Corbató and Vyssotsky 1965]. This was a pioneering and ambitious effort to implement a multi-process, multi-user operating system on a large scale.

The combination of data analysis research and large-scale applications had sensitized management at Bell Labs to the relevance of statistical computing. The first planning for a statistical system began in 1965 (with John Tukey participating in our initial meeting). The system was to be built around the PL-1 language and was predicated on Multics as the operating system environment. The proposed name was BLISS, for Bell Labs Interactive Statistical System (although the "I" was sometimes interpreted as Interim).

Bell Labs dropped out of the Multics project in 1966, for practical reasons. The Murray Hill location of Bell Labs had scheduled the replacement of its IBM 7094 system with hardware from General Electric, with the intention of running Multics. The IBM equipment was promised to a new Bell Labs location in Indian Hill, Illinois.

Not surprisingly, the implementation of Multics took considerably longer than had been predicted. It was clearly not going to be generally usable when the hardware transfer occurred. With the new hardware at Murray Hill but no Multics, most of the Research area of Bell Labs was left with a computer system from a much less experienced company than IBM, an operating system not the one desired (and neither understood by us nor bug-free) and less of a software base than the IBM, let alone what had been expected with Multics.

For data analysis, the immediate computational strategy was largely a rescue mission, to provide a capability to manage and analyze data with the scale and reliability we required, and with access to the numerical capabilities necessary for the analysis. The facility took the form of a subroutine library, callable from Fortran and largely implemented in that language.

The BLISS project was dropped, inevitably since it not only assumed the Multics operating system but was to have been an extension of the PL-1 language planned for Multics but not available otherwise. Of the small group involved in planning BLISS, only I would still be at Bell Labs and involved in statistical computing when work on S began. Only a little prototyping had been completed on BLISS, none of which was relevant to later work.

It would be a decade before the first version of S was implemented to provide an interactive environment for flexible analysis applied to a wide range of data. However, that decade was by no means static. Research in data analysis and collaborative projects continued actively. Providing state-of-the-art computing focused initially on relatively specific methods, implemented as subroutines to be called from Fortran and organized in a subroutine library.

When we came back to create an interactive environment, the computing facilities incorporated in the library had expanded enormously, both for Fortran's traditional domain of numerical computation and for the other areas that make up data science. Computations for linear algebra, optimization, function approximation, random number generation and data manipulation (e.g., sorting and searching) were among those largely revolutionized by new computer-oriented techniques. These were often implemented in publicly available sources, such as published algorithm sections. The community involved in using and testing these algorithms grew rapidly as well.

Additional areas had been advanced locally, including two of relevance for data science: visualization and data management. A flexible structure for computer graphics in support of data analysis was implemented through Fortran subroutines. This software, referred to as GR-Z [Becker and Chambers 1976], provided a structure for graphics later adopted and extended in S and therefore in R.

I wrote (but never described externally) some data management software that supported a general model of data structures defined hierarchically with named components, starting from vectors and scalars of some basic types. The structures were self-defining and extensible. Lengths of components could be queried and modified. Users could create arbitrary new types of structure. With some modification, this software provided the initial implementation for data structures in S.

2.3 First Version of S

The first meetings to plan for an interactive statistical system took place in 1976. At this time, powerful software for data analysis existed in the form of an extensive subroutine library. Interactive use of this software was becoming possible through time-shared terminals.

However, the software in the library could only be used by writing a complete control script, also a Fortran program, that managed the data, carried out the analysis and produced some informative output to be viewed later. This would be run as a "job", with control information included, in a format little changed from the days when user card decks were submitted for operators to run.

The details needed were sufficiently tricky and extensive to be outside the skills of the principal investigators, particularly for analysis involving serious data in terms of size or complexity. As a result, most data analysis involved a team including programmers. Decisions about new analysis required communicating the ideas to the programming staff, adding to the delay and discouraging repeated changes.

When Rick Becker joined us, he had experience with an interactive system at the National Bureau of Economic Research that, while much less extensive or general than our software, provided truly interactive analysis. Rick and I became convinced that we could build a system combining convenient interactive use with access to the full power of the Fortran-based library. We proposed to create an "interactive environment" (as the title of the first book on S [Becker and Chambers 1984] referred to it) that continued to support data analysis but gave the analyst a convenient, direct interface. To achieve this goal, the new software had to provide three features:

(1) *Convenience*: compact, straightforward expression for the analysis, with S handling details such as managing the data and providing graphical or formatted output.
(2) *Completeness*: the extensive range of summaries, modeling and visualization provided by the Fortran library had to be available;
(3) *Extensibility*: we were a data analysis *research* community, so new techniques would need to be available from S.

A fourth requirement was that this be implementable with a relatively modest programming effort; essentially, the two eventual authors with help from various colleagues.

The approach that succeeded in satisfying all the requirements was to build the system around an *interface* to Fortran. From the start of the project, our design was based on writing specialized code to incorporate individual Fortran subroutines into S by writing a specialized interface function for each of them.

Typically for Bell Labs research, we felt free to start the project without any formal approval. The first meeting, on May 5 1976, involved about five people as I recall, none of them management. We presented some ideas and preliminary software. Figure 1 is the first "visual" of the first talk. The upper half of the figure illustrates the concept of the interface implemented in Fortran: a Fortran subroutine (`XABC`, the rectangle) interpreting the user's interactive expression, passed in as an S object by the argument `INSTR`. Eventually the interface calls the Fortran algorithm (`ABC`, the circle). The result is returned to the interactive user as an S object (through the argument `OUTSTR`, since `XABC` is a subroutine rather than a function). The lower half of the figure sketches the implementation of the objects for the user's call and the value returned.

The details of how all this would work were somewhat unclear initially, of course, but the design suggested by the Figure is broadly consistent with the implementation over the next couple of years. By the time S was distributed generally, the interface mechanism, as well as being the implementation for most of the system, was provided to users as an interface *language*, [Becker and Chambers 1985], the chief mechanism for extensibility. The interface language would be pre-compiled into Fortran.

The essential programming unit was a function definition. Each function would be available in S with the name and arguments defined in the interface code. The arguments in a call to the function would be objects in S. The function would return an S object as its value. The body of the function could contain any Fortran code but typically it would call a subroutine, not dependent on S, to do the actual analysis, visualization or other computation. In particular, this made essentially all the code in our library available for incorporating into S.

The interface language also had some facilities for creating and manipulating certain S objects, mainly vectors, matrices and lists with named components that were themselves S objects. The interface language mapped objects or their components into Fortran arrays, usually numeric. The language also had built-in accessor functions to provide necessary scalar information for Fortran, such as the length of a vector or the dimensions of a matrix.

In addition to the interface routines, top-level code parsed the user language, loaded the code for individual functions and evaluated the call to the function.

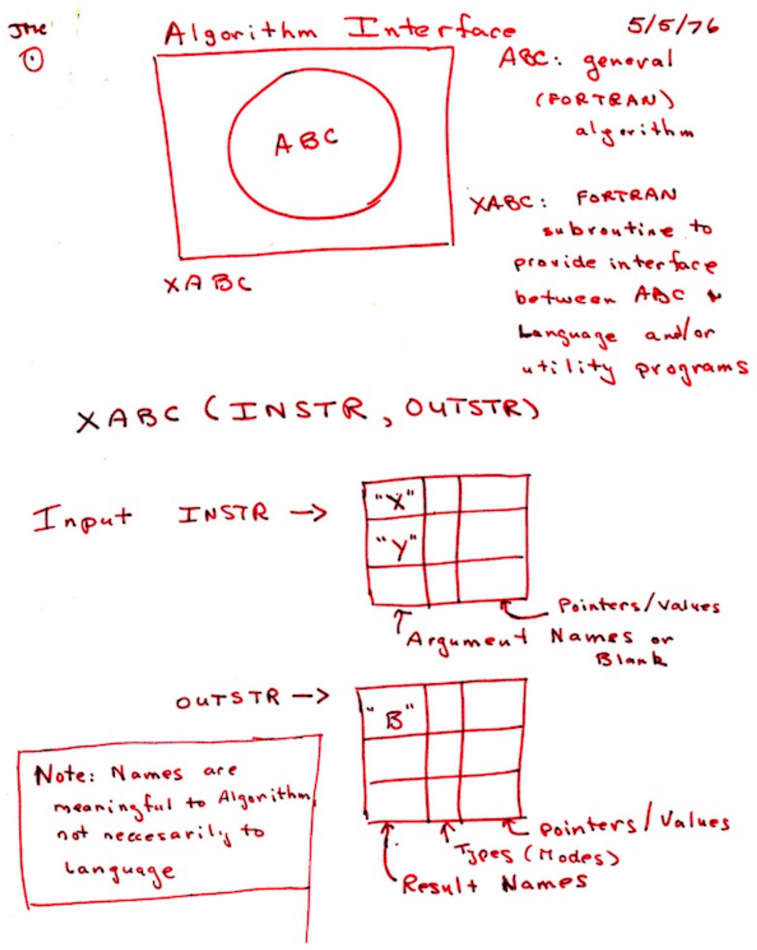
JMC
①
Algorithm Interface
5/5/76
ABC
ABC: general (FORTRAN) algorithm
XABC: FORTRAN subroutine to provide interface between ABC & Language and/or utility programs
XABC
XABC (INSTR, OUTSTR)
Input INSTR →
"X"
"Y"
Pointers/Values
Argument Names or Blank
OUTSTR →
"B"
Note: Names are meaningful to Algorithm, not necesarily to Language
Pointers/Values
Types (Modes)
Result Names

Fig. 1. Design for an Interactive System. May 5, 1976

The user language consisted of expressions, generally C-style, plus a few extra operators and minus declarations. The essence of the system was in the functions, several hundred of which were included by the time S was distributed outside AT&T. All function calls took some S objects as arguments and returned an S object as the value.

For example, the function

```
reg(x, y)
```

computed the linear regression of a vector `y` against the columns of a matrix `x`, returning an S object representing the regression, in this case a list with named components.

If the user had, say, read in a matrix and assigned it in S as `myData`, a regression of the first column against the next 3 columns would be:

```
r <- reg(myData[,1], myData[,2:4])
```

Information from the regression object such as `r$coef` and `r$resid` for coefficients and residuals could be used for further analysis of the results.

An interface function would have been written for `reg()` specifying the arguments, with `x` to be interpreted as a numeric matrix and `y` as a numeric vector. The interface function would call a Fortran subroutine expecting such arguments, typically also supplying expressions for additional required information. For example, the length of `y` would be available as `LENGTH(y)` in the interface language.

Such Fortran subroutines would usually return their result by filling in output arguments. The interface language included expressions for allocating corresponding objects in S and for returning an object containing the computed results. (See [Chambers 2016, pp 26–29] for the actual interface function for `reg()`.)

Interface functions were written for many of the analytical techniques and for graphics, including interaction. Basic summaries, data manipulation and simple facilities for data input and report generation were also included. All of these made use of existing code in the Fortran library.

The choice of a C-style user language seems obvious now, but was not standard for a statistical system at the time. For example, some of our colleagues who were Unix authors or users suggested that a shell-style language built around pipes would be just as capable and easier for users to learn. Our feeling was that nearly all the likely early users of S were comfortable with a scientific expression language. Also, general computations in data analysis fairly quickly become more tree-like. Both arguments in the regression example above are themselves function calls (in S, operators are functions), making a pure pipe syntax less convenient.

The user language retained this form throughout the evolution of S and R, although additional structure was added.

In a 2016 book on R, I asserted that its design can be summarized by three principles, [Chambers 2016, pp 4–11]:

objects: Everything that exists in R is an object.
functions: Everything that happens in R is a function call.
interfaces: Interfaces to other languages are a part of R.

These principles did not suddenly appear at a late stage in the evolution of the software; rather, they are broadly visible from the first version of S and explain a number of detailed decisions.

For example, the centrality of function calls is clear from Figure 1 although functions as objects came later, as will be discussed below. The fundamental role of interfaces is also clear from the Figure; initially, only to Fortran but later to C also and to executable code generally.

The uniform approach to dynamically allocated S objects was partly a result of depending on the existing data management routines in the subroutine library. A resulting idiosyncrasy was that S had no scalar types as distinct from general objects. The intention was that low-level computations depending on these would be done through the interface to Fortran. It was also a practical decision: the available programming resources would have been insufficient to implement a language with similar capabilities from the ground up.

By 1978, a version of the system existed and was in use on the Murray Hill computer system (now Honeywell, which had purchased the General Electric computer division). The system survived without a name for its first few months but was eventually called "S" (initially with quotation marks).

2.4 S Outside Bell Labs

By the end of the 1970s, S was proving popular with users in statistics research and also in some other research and development organizations at Murray Hill. Only at Murray Hill, however, because it was implemented in the operating system for our unintended Honeywell computer and built upon the local subroutine library written for that machine.

A desire to enlarge the user community motivated an effort to port S to other hardware and operating systems. The prospect of rewriting the system directly for a variety of targets was daunting. The mechanism in the S main program to swap in and communicate with code for individual functions was a custom-built overloading computation specialized to this operating system. Porting this and other parts of the system management to each target machine seemed likely to be tricky and tedious.

The solution came via another, separate fallout from the Multics debacle. Some of the Bell Labs computer science researchers involved with the Multics effort decided to pursue related operating system ideas, on a deliberately smaller scale. As the ideas crystallized into an actual system, it was called Unix [Ritchie 1984].

At first, Unix seemed an unlikely target for S and not particularly helpful for portability. Like S, Unix was initially implemented on the available machine; in its case, the 16-bit PDP-11. Hardly a promising target for substantial data science and the initial uses of Unix did not include sizable numerical computations (no Fortran compiler, for example).

Fortunately, the growing popularity of Unix motivated them, like us, to strive for portability. A new version of Unix was designed for portability with operating system functionality callable from C [Johnson and Ritchie 1978]. Unix was ported to some 32-bit machines and a Fortran compiler included, through translation into C.

We took advantage of the popularity of Unix and its adoption on many platforms to define portable S to be a Unix implementation. Non-portable operations such as loading the code for functions and transferring control to them were re-implemented in C for Unix. This was technically a second version of S but retaining the existing user and interface languages, with only a few changes in individual functions.

Unix did us a second important service. Through negotiations with the appropriate legal organizations at Bell Labs, Unix was licensed for distribution outside AT&T. We were able to arrange for similar licensing of S.

By 1985, an S community was starting to grow, including particularly welcome interactions with university users, for whom an S license was relatively inexpensive. Statisticians in a number of prominent university departments included some enthusiastic users (e.g., at Wisconsin, Carnegie Mellon, Berkeley and Toronto). Two books described the software: a user's manual [Becker and Chambers 1984] and a book describing the extension of S by writing functions in the interface language [Becker and Chambers 1985].

3 1985–2000: S, LEADING TO R

At the same time that the S community was growing using the current version, the original authors were at work on a new version. The user language retained its grammar and the system supported nearly all of the functional capabilities, but the implementation was based on a new computational model.

The S software in this version was the basis for R. The plan for R was to reproduce the form and analytic capabilities of S, with additional features. This was in fact what happened, complicated by further evolution of S during the implementation of R, from which some new features were also incorporated.

To sort this out, it will be helpful to document the evolution of S during the period (Section 3.1), describe the creation of R (Section 3.2) and summarize the structure of R as it derives from S but also its relevant new features (Section 3.3). Lastly, we will relate the shared computational model of S and R to data science (Section 3.4).

3.1 S, Versions 3 and 4

The "new S", as the title of its 1988 user manual [Becker et al. 1988] described it, was not back-compatible. However, it aimed to give the user the same experience of high-level interactive data analysis combined with the ability to incorporate new research.

The subtitle of the original user manual [Becker and Chambers 1984] was *An Interactive Environment for Data Analysis and Graphics.* The subtitle of the new book was almost identical except that *An Interactive ...* was replaced by *A Programming ....* The design aimed to provide a unified and convenient organization for programming in the language and for the organization of data.

The new S retained the three fundamental principles mentioned on page 7, but in a revised form.

objects: The uniformity of objects became explicit for the user, with properties such as the object's class available for programming. New classes of objects extended the analysis.

functions: The key change for programming was to add function definitions to the user language. Programming would now be centered on the creation of new S-language functions. Function definitions were now S objects that could be passed around and computed on.

interfaces: The interface language disappeared but inter-language interfaces remained central, especially for C but still for `Fortran` as well, along with an interface to the `Unix` shell.

S continued to evolve during this period, but largely by adding new capabilities or programming features without breaking back-compatibility, documented for the user community mainly by the 1988 book and two subsequent books.

A 1992 book, [Chambers and Hastie 1992], introduced an approach to fitting and analyzing statistical models, using S's object-based computations and a version of functional object-oriented programming. The statistical modeling software made use of the flexibility in objects and functions to create a unified and statistically informative approach to fitting models.

Statistical models were described and implemented in several classes (e.g., linear models, smooth curve fitting, . . .). Within each class a particular model was defined by a structural *formula*, essentially an expression in S that was a symbolic representation of the particular model, and by the *data* from which the model should be estimated. The data would typically be gathered together in a new class of objects, the *data frame*, representing a sequence of n observations on the same p variables.

The conceptual framework of this software was carried over into R, along with most of the specific functionality described in [Chambers and Hastie 1992], and became influential for future work in data analysis, two aspects in particular.: models as objects and the data frame as a representation of data for scientific studies. Viewing fitted models as objects from corresponding classes emphasized the data analysis philosophy encouraging visualization, examination and further modification. In S and R, generic functions for plotting, updating and extracting information will have methods for the various classes of models. The structure has lent itself to research and implementation of new classes of models, with corresponding R packages.

The data frame concept is central. It can be viewed both as a table of named entries (the variables) and as a rectangular array of the individual observations. However, it differs from a general dictionary or table in that each entry must have n elements corresponding to the observations and it differs from a matrix in that the entries may be of different types. In S and R, the data frame is implemented as a class built on a vector of type `"list"` with attributes defining the variable names and other properties. The rectangular structure is implemented by functional methods for the class; for example, to extract or replace data by matrix-like expressions. The data frame concept has been replicated in other software and languages, such as the `DataFrame` class in the `pandas` software in `Python` (https://pandas.pydata.org). Section 3.3 will discuss further details of the implementation.

The example of linear regression illustrates the concepts. The original `reg()` function regressed a numeric vector on a numeric matrix, but the new structure allowed more flexible possibilities. So, a fit of runTime to runners' age from a dataset of racing results [Kaplan and Nolan 2015] might have the form

```
        r1 <- lm( runTime  ~ age, data = cbMen)
```

The first argument is now an expression that effectively produces a symbolic form for the model, typically containing the names of variables in the data frame supplied as the second argument — potentially in general S expressions, e.g. log(age).

The object returned is from a class, "lm" in this case, for which functional methods simplify further study, such as specialized graphical displays:

```
        plot(r1)
```

creates a specialized visualization useful for studying linear regression fits. Other statistical modelling techniques would replace lm() with aov(), for example for an analysis-of-variance model, but giving the user similar features for specifying the model and studying the result. Section 3.3 will relate these techniques to the design of the language.

At the same time that R was being implemented as a "free" S, additional research in statistical computing at Bell Labs produced a new version of S, extending it in a generally back-compatible way. From 1997, this was the version of S as licensed by Bell Labs (by then part of Lucent Technologies after the further split of AT&T) and generally referred to as Version 4 of S, described in a 1998 book [Chambers 1998].

Some of its features were incorporated in the initial version of R or added later. A more general, more formal version of object-oriented programming was provided, although it did not replace the earlier version in the white book. An additional interface to C was provided that passed references to S objects, on which users could program via C macros and utility functions. Classes of connection objects were defined to deal more generally with input and output (e.g., fifo and pipe connections). (Section 3.3 will examine these topics in more detail).

The licensing of S had always provided a re-sellers option for those wanting to produce a commercial product extending S. Of several such efforts, the dominant one became software known as S-Plus, managed by a company called MathSoft in 1998. The company changed names a few times until becoming a subsidiary of TibCo (the Wikipedia article, https://en.wikipedia.org/wiki/S-PLUS outlines the history). Eventually, the owners of S-Plus obtained an exclusive resale license and in 2004 bought the rights to the S software.

The S-Plus user community existed, and continues to exist to some extent, along with the growing R community. Some desire for compatibility of R with S-Plus was relevant in the development of R, as will be noted in Section 3.3.

3.2 The Birth of R

Ross Ihaka and Robert Gentleman published a paper in 1996 [Ihaka and Gentleman 1996] describing a "language for data analysis and graphics". The project had been previously announced through an S community letter and a small but growing group of contributors existed. Interest in R grew following the publication and by 1998 the contributors had expanded to a group of 11 "volunteers", with write permission on the R source (an informal group that came to be known as *R core* and that has continued, with gradually varying membership, to be responsible for the official versions of R until the present day).

None of us at Bell Labs was consulted, ensuring a valid implementation freely available for distribution. Some notes by Ross Ihaka appear to be the main documentation of the early R core work [Ihaka 1998]. The goal of the project took its key form at this time: R would be "a free implementation of something 'close to' version 3 of the S language".

Setting the goal of a "GNU S" (as R is still described on the project web site) was a watershed decision. The original form of the language and the basic approach to objects resembled S, with some internal distinctions partly reflecting Lisp-style languages and the writings of Abelson and Sussman [Abelson and Sussman 1983]. In particular, the data type known in R as a `pairlist` was the traditional Lisp list and was used to manipulate objects and for other basic computations. S function objects and the evaluation of calls used a form of closures.

Implementing a close approximation to S required a different model for objects and for evaluation; respectively, vectors with attributes and so-called *lazy* evaluation (Section 3.3). Pair-list objects and closure semantics are still used internally, but are now largely isolated from the R user or programmer.

As noted, the definition of S used as a model for implementation was taken from the "blue" book and "white" book, since there was no formal definition. The blue book contained a semi-formal model of the language, including the evaluation semantics [Becker et al. 1988, Ch. 11]. An appendix, as before, provided the detailed documentation of the functions supplied with S. These two inclusions supplied at least an approximation to a definition of S, available without obtaining the licensed software itself.

The key content of the white book in terms of data frames and model objects was reproduced, along with the more standard of the specific types of models; e.g., linear regression and analysis of variance. Data frames in particular, [Chambers and Hastie 1992, Ch. 3], became central to many extensions in R, reflecting their basic relevance to data science. A data frame represents a set of n values for p observable variables, the classical form of scientific data. Computationally, it combines the properties of a list (of variables, possibly of different types) with those of an n by p array.

R thus inherited the computational techniques, interactive user interface and programming structure of S, shaped by the data analysis philosophy of Bell Labs. This in turn featured an emphasis on deep engagement with data, exploration and collaboration that constituted a pre-adaptation to the needs of modern data science.

The work of the R core group resulted in the release of version 1.0.0 of R on February 29, 2000.

3.3 R and S

This section examines the main technical characteristics of R, noting the features of S it implemented and those where it diverged or added features original with R. In addition to this historical, "vertical" view, we note a few of the main "horizontal" distinctions that differentiate both R and S from some other languages, since these often relate to the data science domain-language aspect of R.

Throughout this section, any descriptions unqualified by mentioning either S or R will indicate characteristics common to both, implemented in R to replicate known behavior in S. When S and R use different terms to refer to the same concept, the R term will be used.

The discussion is organized by the three principles into *Objects*, *Function calls* and *Interfaces*, plus a fourth topic, *Packages*. The impact of R as well as its relation to data science has been strongly influenced by the growth of the R community of users and contributors. For this growth, the R package structure has been the key addition to the software. The packages available from several sites, especially the central CRAN archive, now form an extensive, usable and relatively coherent code base for applications.

Objects. Objects in the original version of S were an extension of Fortran one-way arrays, called *vectors* in S. Unlike Fortran, S allocated vectors dynamically as required in function calls. These vectors contained elements of a single specified type, initially corresponding to the types found in Fortran. Two characteristics distinguished these S vectors from the basic types in other languages: for all types, elements may be "missing", denoted by `NA`; and there are no scalar types.

Version 3 of S, and therefore R, retained vectors as the core data structure. An extensible facility for defining general object structure was built on this through two features. Vectors could be of type `"list"`, with elements being arbitrary objects; and any vector could have a named list of *attributes* to specify additional information.

Vectors with attributes supported an extensible mechanism for adding specialized structure to simple objects, at first implicitly and later explicitly. Objects essential to data science, such as matrices and multi-way arrays, could be considered built-in without requiring a primitive implementation. A general array is a vector that has an attribute named `"dim"` containing an integer vector of the dimensions. Thus arrays automatically can have any type of data and can allow for missing values. Separating the data from the attributes is helpful for data analysis, separating the logic that depends on the structure from the computations on the specific type of data.

Advances in data analysis led to the need for more specialized data structures and for specialized computations to generate and operate on them. The natural, and perhaps inevitable, language extension to implement this coherently was *functional object-oriented programming*. Functions may be *generic*, with the computational method for particular arguments selected corresponding to the class of the argument(s). Statistical models for data are a natural application and were in fact the motivation for the first implementation [Chambers and Hastie 1992, Appendix A]. A more general and more formal version followed in [Chambers 1998], but the simpler one continues to be popular.

The uniformity of objects extends to functions. In particular, functions are simply objects, with a syntactic definition in the language. Note that a name is not part of the function definition, in contrast for example to Python or Julia. Assigning a function is not different from assigning any other object. As noted below, the semantics of function call evaluation are defined directly from the object, regardless of how that object was obtained.

R supports all the object structure of S but with an implementation at the primitive level reflecting influence from Lisp. Language objects such as function definitions and unevaluated function calls are implemented via Lisp-style lists, but these are largely hidden from users who are encouraged to manipulate such objects by conversion to and from vectors of type `"list"`.

A more important influence, specifically from the Scheme form of Lisp, is that R is lexically scoped. In particular, any assignment of a function object incorporates a reference to the *environment*—the other assigned objects existing where the assignment took place. Since everything is an object, this environment is itself an object, of type `"environment"`. As an ordinary object, an environment is effectively a dictionary of objects indexed by character strings. But in an environment, objects are accessed by reference, which deliberately contradicts the usual non-reference semantics of S. Changing an environment, by changing the object associated with a particular string, changes that environment wherever it is currently referenced.

Environments are key to evaluating function calls and to the installation and use of R packages, as discussed under these topics below. Environments can also be used directly in R and have been, for example to implement the usual form of object-oriented programming [Chambers 2016, Ch. 11]. Their reference-style semantics does pose some dangers: the same computation on an object in R might have different results if that object was an environment on one hand or a list with the same elements corresponding to the same names on the other.

Function calls. The "everything that happens is a function call" principle reflects a design goal of S to encourage and support *functional programming* in the language. Combined with the object principle, functional programming implies an explicit conceptual model for a function call. Based on the definition of the function and the objects supplied as arguments, an object is computed and returned as the result of the call. No modifications to the arguments or other side effects should result and the function definition should determine the computations. R is not a pure functional language: various tricks and special computations exist to violate the principal. However, the essential language structure promotes functional programming; in particular, through the implementation of the function call itself which differs from languages that regard the call as simply taking a vector of references to the arguments as objects.

A function call in R, when it is about to be evaluated, essentially consists of two objects: the function definition and an environment containing objects corresponding to each of the formal arguments in that definition. These usually resulted from a function call in the language with the function identified by name and with some number of actual arguments supplied as expressions, but nothing in the evaluation depends on assuming this. The objects corresponding to formal arguments are special objects of type `"promise"`. A promise object has the expression for the corresponding actual argument (or a special marker for missing arguments), the value of that argument (*if* it has been evaluated), and a flag set when evaluation takes place.

The call is evaluated by evaluating the body of the function "in" the environment of the call; i.e., a name encountered in the evaluation will be searched for there. Ordinary assignments will create objects there. When the name corresponds to a promise object, the promise will be evaluated if it has not been already, in the environment from which the function was called, and the result will be stored in the promise. If the argument was missing and the function definition included a default expression, that expression is evaluated, this time in the environment of the call. So, for example:

```
function(x, scale = sd(x, na.rm = TRUE))
  x/scale
```

is a function that scales an object by the value given and uses the standard deviation of the object (missing values removed) by default.

This evaluation mechanism is often called "lazy" evaluation, but a better term would be evaluate-when-needed. It usually would give the same result as a model where all arguments were evaluated at the start of the call, but some functions depend on the distinction, and might fail without the extra flexibility. For example, default values can use intermediate results computed in the call before the missing argument was needed.

When a name occurs in the evaluation, it is first matched to the environment of the call, then to the parent of that environment, and so on. The parent of the call is the environment of the function definition, which is the environment in which the definition was evaluated. This is used in some computations to create functions inside a call, with the effect that these can then share variables in the original function. More importantly, all the functions in an R package have the same environment, providing a mechanism for sharing specialized tools and data within and between packages.

Interfaces. Effective data analysis today needs to use a variety of powerful tools for modelling and visualization. Well-developed implementations may exist in any of a variety of languages, including C++, Python, Java or Julia as well as R itself and (still) C or Fortran. It is neither practical nor sensible to reprogram the software in a single language; therefore, convenient interfaces from the user's preferred programming language are essential. R now has interfaces on the CRAN repository to all of the above, with several of them being widely used.

The three interfaces of S were retained, to Fortran and C routines for simple arguments and to C with general pointers to R objects. The latter is generally more in use, lending itself to extensions and general computations. For C-level access to objects, R initially replicated the C macro calls used with S, but this has been extended and replaced with its own version. There is also a third C interface, using the Lisp-style representation of the argument list.

The Rcpp interface to C++ is used extensively in packages based on specialized C++ code. The original Rcpp is described in [Eddelbuettel and François 2011], but the interface has been much extended in the version now on CRAN. Approximately 10% of the packages on CRAN use Rcpp. Rcpp includes extensions to C++ to support a high-level programming style with R objects that in many ways resurrects the features of the original interface language of Section 2.3, but now for C++.

Packages. For any open-source software, an important advantage is that experienced users are encouraged to share their extensions and applications, and that the license for the software may enforce these to also be freely available, if distributed. The shared software may just be a folder of source code files, but will be more useful to the community if accompanied by documentation and made straightforward to include and to access from the user's software.

R has an extended definition for shared software, the R package. This has a prescribed hierarchical structure. The structure provides for documentation, R source, data files and some top-level description of the package. There is also a standard structure that simplifies inclusion of code for compilation in C, C++ and Fortran. Further optional structure allows inclusion of essentially anything, notably code from any other language. A collection of tools is used to install, load and invoke software from the package.

Compared to libraries or modules in other languages, the package in R imposes considerable demands on the programmer; for example, where a Python module is essentially just a folder containing code, an R package organizes code in a specific structure of subdirectories that then enables a very general but standardized format. In the basic package-sharing mechanism, a source copy of the package is processed by an `INSTALL` utility into a folder whose contents can then be loaded into an R session, making the code, documentation and any other content available to the user of the session.

The package structure and repositories of contributed packages have played a major role in the usefulness and popularity of R. They put some extra burden on providers of the extended software, particularly if the package is to be accepted by one of the central repositories, notably CRAN, which is by far the largest and most used site and is associated with the R project itself. CRAN enforces standards for the documentation, portability and usability of contributed packages. This is more than compensated by benefits to users in terms of software documentation and testing, and usually benefits the authors also in the long-term evolution of their software.

3.4 Data Science

Some of the central and influential features of S as described in the two books of 1988 and 1992 illustrate its nature as a domain-specific language and system for data science. R took over these features, adding some important extensions and improvements but with the focus still on data science.

In 1988, the preface to the blue book [Becker et al. 1988], stated:

> The primary goal of the S environment is to enable and encourage good data analysis.

This explicitly states the goal as supporting the domain of data analysis, Bell Labs style, very much in the spirit of modern data science.

The domain of data science was also implied (though still not named as such) in the citation when S received the 1998 ACM Software System Award, [ACM 1998]: S had "forever altered how people analyze, visualize, and manipulate data".

Some of the capabilities of S important to users trying to do data science are implied in the citation:

- *visualization*, usually referred to as "graphics" in the books. The blue book preface, in listing key features, said "Especially, S is about *graphics*: ... flexible ways of looking at data".

- *analysis.* S introduced an object-based view of analysis. A linear regression fit, for example, returned an object from a corresponding class. Simple S expressions then produced visual and numerical information, encouraging interactive exploration. This style has become the norm for modern data analysis.
- *data.* Some classes of data introduced have become central to data science in R and beyond.

From a data science perspective, the most important class of objects is the *data frame*, which models the structure in which scientific data has always been recorded: a table indexed by observations and variables in which each observation records corresponding values of the variables. As noted previously, this class was introduced to S in the context of statistical models, but is widely used and remains an active area for new developments, such as the "tidy" version in *R for Data Science* [Wickham and Grolemund 2016]. Analogous types have been added to software in other languages, such as the `DataFrame` structure in the Pandas software in Python.

From a programming perspective, however, a data frame does not correspond to a standard type of data. It cannot be a two-way array in the sense of Fortran (or R or Julia), because while all the values of a single variable will be constrained to have the same type, different variables can correspond to different types (numeric versus categorical, for example). Neither can it be a simple table or dictionary indexed by the names of the variables. The number and order of values from the variables is linked by their correspondence to particular observations; operations on the object cannot be allowed to revise some elements inconsistently with the rest.

4 FROM 2000: R

S did not quite disappear with the initial arrival of R. During the years 1996 to 2000 when the R Core team was preparing the official version of R, another version of S appeared at Bell Labs, as noted, while independently the commercial S-Plus system based on S maintained a significant user base.

After the official launch of R in 2000, open-source R gradually became the dominant source of new software for statistics and data science. S-Plus continued as a commercial product. However, by late 2000 the Bell Labs researchers still involved with S, Duncan Temple Lang [Temple Lang 1997] and the present author, had both accepted invitations to join R Core. By the start of the 21st century, the evolution of Bell Labs' S had ended.

Since its official first version, R has expanded in all measures: users, contributors, citations and public awareness. Some measures have shown literally exponential growth, such as the number of packages in the main repository, CRAN. Although S had accumulated a significant user base by the time of the Software System Award in 1998, the impact of R is on an entirely different scale.

R's popularity is no doubt due to a number of factors but a principal one is its evident link with data science, which has shown a similar explosion of public interest and involvement. R features in the teaching and practical projects for nearly all programs in data science. Data science is inevitably mentioned in popular articles on R, e.g. [Thieme 2018].

R inherited the data science orientation of S by replicating the structure and contents of Version 3 of S. To this R added some key contributions of its own in the internal computational model; for example, the role of closures and of R environments generally. For the future of its contributions to data science, perhaps the most important feature of R was the package structure.

In the balance between effort required from the software developer and the usefulness of the result, R tips the scales toward the user. This is reinforced by the central CRAN repository. A package on CRAN is more visible, easier for users to install and has some extra prestige, motivating developers to spend some effort to comply with the requirements.

That effort has been reduced by tools to assist the creation, modification and distribution of packages, particularly before they are ready to be part of CRAN or similar repositories. Some of the tools are R-independent; for example, github repositories have become virtually standard for circulating a package in a more flexible evolving format.

Integrated development environments (IDEs) specifically for R have also accelerated the creation and revision of packages and other R software. A notable example is the RStudio IDE. This is a desktop integrating the use of R with editing, graphics, documentation and a variety of utilities that typically replace specialized R- or shell-level commands with button clicks or other interactions. RStudio is a commercial enterprise but the IDE and many associated R packages are freely available and open-source (www.rstudio.com). The RStudio IDE has become popular for teaching and specifically for courses associated with data science. It also greatly simplifies editing and installing an R package.

Repositories of contributed packages, and in particular CRAN, have become a key driver in the growth and extension of R as specialized for scientific disciplines or other areas of application. This has relevance for data science: progress will require increasing collaboration between researchers in the scientific disciplines, on one hand, and professionals developing the statistical and computer-science techniques for data science, on the other. R and its package structure are popular ways to bring data science techniques to a specialized audience; for example, a plethora of "Using R for xxx Data" books have appeared, with corresponding packages.

Data science will increasingly require a widening range of high-quality software for diverse purposes. No single language or environment will be universally suitable. Interfaces between languages have always been part of the design of S and now R. The R package structure facilitates including code from other languages: compiled code from C, C++ and Fortran but also code in any other language to which R has an interface [Chambers 2016, Part IV]. Multi-language IDEs such as Jupyter are complemented by such interfaces, allowing the R programmer to request specific computations from the other software in a natural R style. The end user does not need to do any programming in the language on the other side of the interface, in contrast to the IDE approach.

ACKNOWLEDGMENTS

Thanks to Jean-Baptiste Tristan, the conference historian, and the referees for suggestions on early versions of the paper and to members of R Core for correcting some historical details. Most especially, much gratitude to all the members of the S and R teams over these many years. Both languages have always been the product of close collaboration, including all the authors of the related books in the reference list but also those extending S and R, now a community of thousands over different disciplines and applications.

REFERENCES

H. Abelson and G. J. Sussman. 1983. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, MA.

ACM. 1998. ACM Software System Award. https://awards.acm.org/award_winners/chambers_6640862.

Richard A. Becker and John M. Chambers. 1976. GR-Z: A System of Graphical Subroutines for Data Analysis. In *Proc. 9th Interface Symp. Computer Science and Statistics.*

Richard A. Becker and John M. Chambers. 1984. *S: An Interactive Environment for Data Analysis and Graphics.* Wadsworth, Belmont CA.

Richard A. Becker and John M. Chambers. 1985. *Extending the S System.* Wadsworth, Belmont CA.

Richard A. Becker, John M. Chambers, and Allan R. Wilks. 1988. *The New S Language.* Chapman & Hall, Boca Raton, FL.

John M. Chambers. 1998. *Programming with Data: A Guide to the S Language.* Springer, New York.

John M. Chambers. 2016. *Extending R.* Chapman & Hall/CRC.

John M. Chambers and Trevor Hastie (Eds.). 1992. *Statistical Models in S.* Chapman & Hall, Boca Raton, FL.

F. J. Corbató and V. A. Vyssotsky. 1965. Introduction and overview of the Multics system. In *Proceedings of the November 30–December 1, 1965, Fall Joint Computer Conference, Part I (AFIPS '65 (Fall, part I)).* ACM, New York, NY, USA, 185–196.

https://doi.org/10.1145/1463891.1463912

David Donoho. 2017. 50 Years of Data Science. *Journal of Computational and Graphical Statistics* 26, 4 (2017), 745–766. https://doi.org/10.1080/10618600.2017.1384734 arXiv:https://doi.org/10.1080/10618600.2017.1384734

Dirk Eddelbuettel and Romain François. 2011. Rcpp: seamless R and C++ integration. *Journal of Statistical Software* 40, 8 (2011), 1–18. https://doi.org/10.18637/jss.v040.i08

A. E. Freeny and J. D. Gabbe. 1969. A statistical description of intense rainfall. *Bell System Technical Journal* 48 (1969), 1789–1851.

Jon Gertner. 2013. *The Idea Factory: Bell Labs and the Great Age of American Innovation.* Penguin.

Ross Ihaka. 1998. *R : Past and Future History.* (draft for Interface Symp. Computer Science and Statistics): https://cran.r-project.org/doc/html/interface98-paper/paper.html.

Ross Ihaka and Robert Gentleman. 1996. R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics* 5 (1996), 299–314.

Louis Jaeckel and John Gabbe. 1974. Crawford Hill rainfall data. In *Exploring Data Analysis: The Computer Revolution in Statistics*. University of California Press, Chapter 3.

S.C. Johnson and D. M. Ritchie. 1978. UNIX time-sharing system: portability of C programs and the UNIX system. *Bell System Technical Journal* 57, 6 (1978), 2021–2048.

Daniel Kaplan and Deborah Nolan. 2015. Modeling Runners' Times in the Cherry Blossom Race. In *Data Science in R*, Deborah Nolan and Duncan Temple Lang (Eds.). Chapman and Hall/CRC, Chapter 2, 45–103.

D. M. Ritchie. 1984. The evolution of the UNIX time-sharing system. *AT&T Bell Laboratories Technical Journal* 63, 8 (1984), 1577–1593.

Duncan Temple Lang. 1997. *A Multi Threaded Extension to a High Level Interactive Statistical Computing Environment.* Ph.D. Dissertation. University of California, Berkeley.

Nick Thieme. 2018. R Generation. *Significance* 15, 4 (August 2018), 14–19.

John W. Tukey. 1962. The future of data analysis. *The Annals of Mathematical Statistics* 33, 1 (1962), 1–67.

John W. Tukey. 1977. *Exploratory Data Analysis.* Addison-Wesley, Reading, Massachusetts.

Hadley Wickham and Garrett Grolemund. 2016. *R for Data Science: Import, Tidy, Transform, Visualize, and Model Data.* O'Reilly.

Martin B. Wilk and Ram Gnanadesikan. 1968. Probability plotting methods for the analysis of data. *Biometrika* 55, 1 (1968), 1–17.