Clustsys: A system for creating software subsystem classifications

CACS-TR-95–5–1
Revised: November 10, 1995

Arun Lakhotia

The Center for Advanced Computer Studies
University of Southwestern Louisiana
Po Box 44330
Lafayette, LA 70504

Contents

Clustsys: A system for creating software subsystem classifications

Arun Lakhotia

The Center for Advanced Computer Studies
University of Southwestern Louisiana
Lafayette, LA 70504
(318) 482-6766, -5791 (Fax)
arun@cacs.usl.edu

Revision history:

1 Abstract

A subsystem classification is an organization of the components of a software system. Clustsys is a system that creates such subsystem classifications. It implements five subsystem classification methods, all of which are based on numerical cluster analysis.

This document gives an overview of the Clustsys system. It describes what this system does, how to use it, and some of its design, as may be relevant for understanding its operations. This document is not intended to be a tutorial on subsystems classification techniques, nor does it prescribe how such techniques should be implemented.

2 Request for credits

We anticipate that the system will be useful either for learning about subsystem classification techniques or for using such techniques in some experiment. If you find this system useful, we would appreciate a citation to that effect in your publications or reports.

3 References

The following papers would be helpful in understanding the details of what Clustsys does.

1. A. Lakhotia, " A unified framework for software subsystem classification recovery techniques", *Journal of Systems and Software*, to appear in 1996.
2. A. Lakhotia and J.M. Gravley, "Toward experimental evaluation of subsystem classification recovery techniques", *Proceedings of the Second Working Conference on Reverse Engineering.*
3. D.H. Hutchens and V.R. Basili , " System structure analysis: clustering with data bindings ", *IEEE Transactions on Software Engineering*, August 1985, pp. 749 – 757.

Other than common UNIX tools, Clustsys is developed using the following software systems.

4. *Software Refinery* version 4.1 and *Refine/C* version 1.1.
   Reasoning Systems, Inc., 3260 Hillview Ave., Palo Alto, CA 94304, USA. Phone: (415) 494–6201. E-mail: `help@reasoning.com.`
5. *FIELD*: a programming environment for UNIX.
   Dr. Steve P. Reiss, Computer Science Department, Brown University. E-mail: `spr@cs.brown.edu`.
6. *Splus* Version 3.0, Release 1 for DEC RISC ULTRIX
   StatSci, 1700 Westlake Ave N, Suite 500, Seattle, WA 98109. Phone: (206) 283-8691. E-mail: `mktg@statsci.com`.

This system can be used without FIELD or without Software Refinery & Refine/C, but not without both.

4 Trademarks

Software Refinery, Refine, and Refine/C are trademarks of Reasoning Systems, Inc. Splus is a trademark of MathSoft. Unix is a trademark of AT&T. Ultrix is a trademark of Digital Equipment Corporation.

5 Terminology

The reader is assumed to be familiar with the following terms: dendrogram, cluster analysis, expected dissimilarity and cluster analysis algorithms. For details see references listed above.

6 Font notation

There is a method to the madness in the usage of the font style. Discovering the method is left as a "reverse engineering" exercise for the reader.

7 Overview of the functionality

Clustsys is a system to classify the components of a software system into subsystems. The functionality it provides may be separated into three parts:

- Basic subsystem classification techniques (SCTs).
- Comparing subsystem classifications (using a congruence measure and an oracle).
- Batch processing for performing large scale experimental evaluation of SCTs.

This document currently only describes the first part, since that is assumed to be the most useful for the expected audience. I would be happy to provide input on the other parts, if there is a request.

7.1 Basic subsystem classification techniques

Clustsys implements five numerical cluster analysis based subsystem classification methods. The five methods may actually be differentiated using two parameters:

(a) the cluster analysis algorithm they use and

(b) the similarity / dissimilarity matrix.

There are two types of cluster analysis algorithms in Clustsys. The first which we refer to as traditional clustering algorithm is implemented using the `Splus` function `hclust`. The second is a variation of the traditional clustering as proposed by Hutchens and Basili (See Lakhotia, 96 for reference).

There are three types of matrices: a binding matrix (a similarity matrix), an expected dissimilarity matrix, and a recomputed dissimilarity matrix. The latter two are computed from the former.

The five cluster analysis methods implemented by Clustsys are referred to by the codes: `e`, `r`, `he`, `hr`, and `hb`. The matrix and cluster analysis algorithm used by each method is given in Table 1.

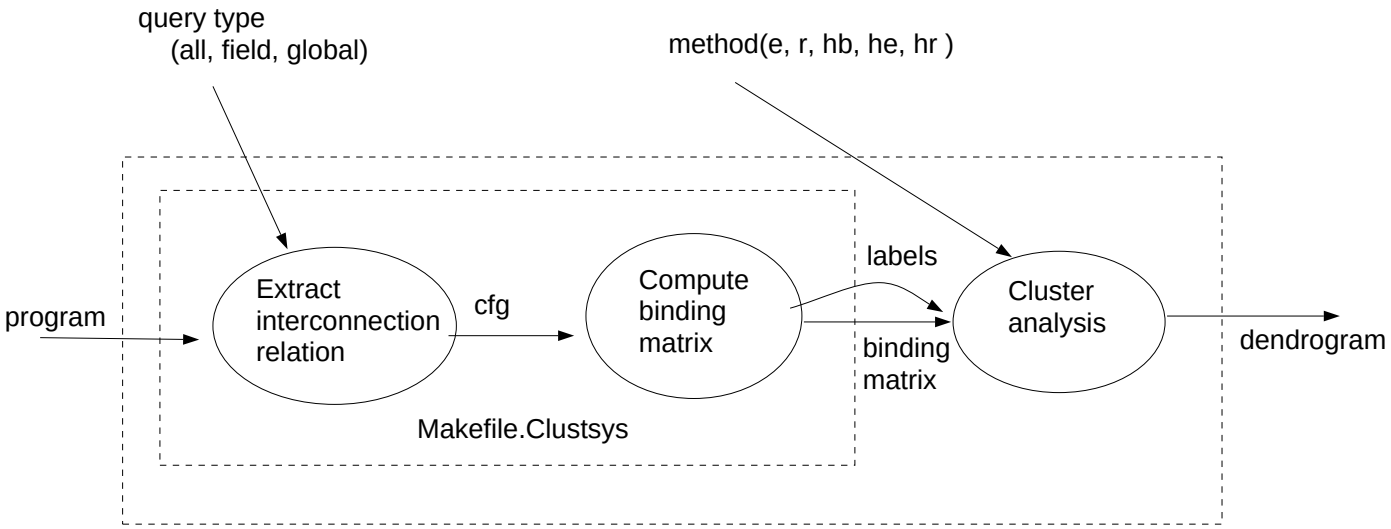
In a nutshell, the `hb`, `he`, and `hr` methods apply traditional cluster analysis on the binding, expected, and recomputed matrices, respectively.

The other two methods Hutchens & Basili's clustering algorithm. This algorithm is different from the traditional cluster analysis algorithm in the following ways. The algorithm inputs a binding matrix (which is a similarity matrix). During each iteration of the algorithm, the binding matrix is used to compute a dissimilarity matrix—an expected dissimilarity matrix for SCT `e` and a recomputed dissimilarity matrix

Table 1 A reference table for the meaning of the codes `e`, `r`, `he`, `hr`, and `hb`.

	Traditional Cluster Analysis	Hutchens & Basili's Cluster Analysis
Binding matrix	hb	
Expected dissimilarity matrix	he	e
Recomputed dissimilarity matrix	hr	r

Figure 1 Toplevel dataflow diagram of the system showing its three major components and the flow of information between them.



for SCT `r`. The dissimilarity matrix is then used to determine the pair of elements that are closest to each other. Next, a new binding matrix is created by merging these two nearest elements and the iteration continues.

8 Design

The system consists of three parts, See Figure 1. The first part extracts interconnection relations from a program, the second part computes a binding matrix from these relations (along with some additional information described later) and the third part performs cluster analysis and creates dendrograms. A knowledge of these three parts is significant since they are currently performed separately (in the absence of a GUI).

8.1 Extract Interconnection Relation

We have two implementations for extracting interconnection relation: one called `query-refine` is developed using Software Refinery and Refine/C and the other called `query-field` is developed using Brown University's FIELD.

For each of the implementations, there are queries for extracting three types of interconnection relations. These queries are called `all`, `field`, and `global`.

All these queries result into, what is termed in Lakhotia, 1996 as, component flow graphs (CFG). The relations represented by these graphs may be represented as a set of 3-tuples; each tuple of the form $<f_1, f_2, s>$, (broadly) stating that the component s flows from component f_1 to component f_2.

In all the three queries components f_1 and f_2 are "functions" (as in "C function"). The component s and the specific flow relation encoded by a tuple differs across the three queries, as follows:

- Query `global`: Component s is a global variable. Each tuple encodes the relation that "global variable s *appears in* the function f_1 and f_2." (It may be worth verifying the code for accuracy of this statement. The initial intent was to encode the relation: "global variable s is modified by the function f_1 and used by the function f_2." There is a possibility this is how it is currently implemented for `query-refine`.)
- Query `field`: Component s is the name of a "field" of a structure (as in C `struct`). Each tuple encodes the relation: "the field s *appears in* the functions f_1 and f_2".
- Query `all`: Component s may be any identifier symbol. Each tuple encodes the relation: "symbol s *appears in* the functions f_1 and f_2."

These days we only use the query `all` and hence the code for other queries may have gotten rotten (or fallen out of sync) due to not being used. Actually, it has been a while since we used `query-field` and are not quite sure about its consistency either. In case you have difficulty using it, please contact us.

8.2 Compute Binding Matrix

The computation of binding matrix is done using a collection of `awk`[1] and shell scripts. The whole process goes through several intermediate steps and generates an intermediate file.

8.3 Makefile.Clustsys

`Makefile.Clustsys` glues together the scripts for extracting interconnection relations and for computing a binding matrix.

Assuming that the environment variable `PATH` is correctly set, the command:

`% make –f Makefile.Clustsys S=`*`pgmname`* `Q=`*`all`*

will generate several files with the primary name *`pgmname-all`*. Of these, two files *`pgmname-all`*`.matrix` and *`pgmname-all`*`.labels` are output to the next phase. Others are intermediate files.

In the above command, the symbol *`pgmname`* refers to the program being analyzed. What it refers to depends upon whether you are using `query-field` or `query-refine`, a choice made by assigning an appropriate value to the variable `QUERY-SYS` in `Makefile.Clustsys`.

When using `query-field` the symbol *`pgmname`* should be the name of the executable (such as `a.out`[2]) of the program being analyzed, and when using `query-refine` the symbol *`pgmname`* corresponds to the prefix part of the file *`pgmname`*`.prog`, a file input to Refine/C to describe the source files that belong to the program being analyzed.

NOTE: Notice that the suffix `.prog` is omitted from the command.

It is worth emphasizing that if *`pgmname`* contains a directory path then the intermediate files and output files created by executing the above command will be created in the corresponding directory. If that is not desired, then one can use the variable `DIR` to give the directory path where information corresponding to *`pgmname`* (i.e., executable *`pgmname`* or *`pgmname`*`.prog` file, as the case may be) may be found.

---

[1] There are two versions of each awk file, one to use with `nawk` and the other for use with `awk`. The `nawk` version is developed recently. It is cleaner and is recommended for use. The `awk` version was developed first. It has not been removed for no good reason. Hence, at the moment `Makefile.Clustsys` uses the old versions.

[2] The name `a.out` is only given as an example. Its use does not imply that I endorse its use for naming executable files.

The two files, *pgmname*`-all.matrix` and *pgmname*`-all.labels`, created due to the above command contain the following information:

- *pgmname*`-all.matrix` contains a `N`×`N` binding matrix, where `N` is the number of functions participating in some relevant relationship according to the `all` query.
- *pgmname*`-all.labels` contains the name of the functions corresponding to each row (or column) of the binding matrix.

The "*pgmname*" part of the file name comes from the parameter `S=`*pgmname* in the `make` command. The "*all*" part comes from the parameter `Q=`*all*. This parameter selects the type of interconnection relation to be used[3].

There are several other intermediate files that are worth knowing about:

- *pgmname*`-<query>.cfg` — contains 3–tuples returned as a result of the <query>. The tuple consists of `<f1, f2, s>` where `f1` and `f2` are functions and `s` is a symbol. The exact relation represented depends on the query and whether `query-refine` or `query-field` is used.
- *pgmname*`-<query>.sorted-cfg` — similar to the corresponding `.cfg` file, except that this one is sorted.
- *pgmname*`.allfunc` — contains the set of all functions in the program. The contents do not depend on any query.
- *pgmname*`-<query>.used-funcs` — contains the functions appearing in the corresponding `.cfg` file.
- *pgmname*`-<query>.fmatch` — contains the file match matrix. This file is used for computing the congruence of a recovered classification with an expected classification. This file gives the oracle against which the recovered classification is compared. Details will be provided later (maybe in another version of the document).
- *pgmname*`-<query>.factor` — contains factor information, also used for computing the congruence with expected classification.

8.4 Cluster analysis

The cluster analysis component is written in two programming languages: Splus and C. Splus was chosen early on in the project for the simple reason that it provides a primitive function, `hclust`, that performs traditional cluster analysis. Furthermore, it also has some primitive functions to graphically display dendrograms. It therefore provided a quick way to develop a prototype system.

C has been used to implement Hutchens & Basili's cluster analysis algorithm[4] and for the computation of the two types of dissimilarity matrices for the `he` and `hr` analyses. The C part is compiled and linked into a library called `cluster-analysis.o` that is dynamically loaded in the Splus code. The file `src/C/main.c` provides a driver for the C part to perform Hutchins & Basili's analysis directly (i.e., without going through Splus)[5]. The driver will not be documented any further.

---

[3] The symbol "Q" implies "Query." It comes from the use of FIELD in the development of the initial version, where a query is written to extract the appropriate interconnection relation from the `xrefdb` program database.

[4] The corresponding functions could have been written in Splus itself. Actually, they were first written in Splus. But our implementation was pathetically slow, primarily because we had essentially coded a C program in Splus, i.e., we were not taking advantage of the vector operations of Splus. Rather than reformulate the analysis in vector form, we it more convenient to develop it in C and integrate it with the Splus code.

[5] This driver is very likely obsolete, because it has not been kept current with changes to the interface of important functions. On the other hand I do not remember any changes made to the interfaces since we first wrote the C part. Nonetheless, the file `src/C/main.c` provides a good starting place for developing such a driver.

To use the cluster analysis part one typically performs the following steps:

1. Load Splus (most likely using the command `Splus -e`).
2. Dynamically load the C library file.
   `> cload()`
   See *Installation instructions* to load the Splus files.
3. Start up X graphic window
   `> motif()` (or `X11()`), etc.
4. Perform the analysis, using one of the following ways:

   - `> demo.clustsys("pgmname-all.matrix" )`
   - `> demo.clustsys("pgmname-all.matrix", labelsfile="pgmname-all.labels")`
   - `> demo.clustsys("pgmname-all.matrix", labelsfile="pgmname-all.labels", method="e")`

   Remember that the symbol *pgmname* corresponds to the name of the program being analyzed and the symbol *all* corresponds to the type of query used to generate the interconnection relations.
   The difference in the behavior of the three commands can be inferred from the description of the function `demo.clustsys` below.
5. Quit Splus
   `> q()`

   Using Splus conventions the interface of the function `demo.clustsys` may be stated as:
   `demo.clustsys(matrixfile, labelsfile="", method="he")`.

This function actually takes five parameters, the last two are for debugging purposes and have been omitted here. The first three parameters are:

1. The name of a file containing the binding matrix (a `.matrix` file), a necessary parameter provided as the first argument.
2. The name of a file containing the function labels (a `.labels` file) , an optional parameter provided by assigning to the actual parameter variable `labelsfile`. The row numbers in the input matrix are used to label the dendrogram as a default.
3. The cluster analysis method to be used, an optional parameter provided by assigning to the actual parameter variable `method`. The default is assumed to be `method="he"`.

This function applies the chosen cluster analysis method on the binding matrix provided in the file `matrixfile` and pops up a graphic display of the resulting dendrogram.

8.5 Data structure for representing dendrograms

It may be of interest to someto know how dendrograms are maintained internally (or when printed during debugging). This may be useful if Clustsys is being interfaced with other tools, such as with a GUI script, or a better graphic displayer of dendrograms, or a software understanding or reengineering tool.

The design of the data structure for dendrograms has been constrained by the design used by Splus, which it turns out is not documented. We did a bit of reverse engineering to get to the structure, and present a sketch here to save time for others. This information pertains to version of Splus stated in the *Reference* section. We hope that newer versions of Splus have maintained the same data structure. Otherwise this documentation exercise will be moot.

Splus maintains a dendrogram using a vector containing three fields with the names: `merge`, `height`, and `order`, as described below.

1. `merge` is a $2 \times$ `N-1` matrix. It is used to represent the binary tree aspect of the dendrogram, in which each node is created due to merging of two elements during an iteration of the cluster analysis algorithm. A leaf element is represented as the negated value of its row number in the original binding matrix. An intermediate node is represented by its index in the `merge` matrix.
2. `height` is a vector of length `N-1`. It maintains the height at which a node in the `merge` matrix is merged.
3. `order` is an vector of length `N`. We are not very sure what this contains. The only thing we know is that: (a) Each element in this vector is the number of a row in the original matrix. (b) The order of elements in this vector influences how Splus displays a dendrogram.

8.6 Data structure for representing matrices

A data structure for representing matrices is not a big deal in this day and age. However, the cluster analysis algorithm has some special properties that require us to use some tricks in the C code (in the name of efficiency). Although the design is most likely very clean—lots and lots of ADTs—it may still be worth giving the top level picture and the rationale.

First, notice that a cluster analysis algorithm starts with `N` $\times$ `N` matrix. At each iteration it creates a new matrix with one less dimension, thereby going from `N-1`, to `N-2`, ..., to `2` dimension (square) matrices.

A simplistic strategy would be to create a new matrix in each iteration (using `malloc`), copy the relevant contents into this matrix, and then `free` the old matrix. This, we felt, would lead to fragmentation of memory besides requiring copying of the whole matrix at each iteration.

We designed an approach that reuses the same matrix area and also does not copy elements. This is done by maintaining what is termed as a `SuppressVector` along with the matrix. This vector maintains which rows (and columns) of the matrix are considered to be deleted.

Maintaining the suppress vector requires some bit of algorithmic jugglery that may be easier to understand based on the above rationale[6]. In addition, that a (dis)similarity matrix is symmetric and has 0s in the diagonal position offers some more opportunity for saving space by not keeping redundant/duplicate information. This part does not require much jugglery, except for mapping a `(i, j)` index to an offset.

9 Acquiring Clustsys

Clustsys is distributed electronically via `ftp` or the web. To access it through the web, follow the threads from the URL `http://www.cacs.usl.edu/~arun`. From here a link to an index of ftp-archive of software systems will lead you to Clustsys.

Alternatively, one may `ftp` Clustsys from `basin.cacs.usl.edu` (IP number `130.70.32.51`), directory: `pub/uslstuff/srl`, file: `clustsys-tar.Z`. Make sure you are using the *binary* mode when ftp-ing this file.

[6] Only yesterday (10/12/1995), in explaining this to Pablo Mejia I happened to do a detailed time and space analysis of this choice of data structure. It turns out that the rather complex algorithm I have implemented is a very poor choice. What it saves in space, it makes up for in time. A better strategy would be to simply reuse the same matrix space for creating the smaller matrices and free the whole matrix only in the end.

Pablo came up with a still better algorithm that uses a heap. It turns out it has already been published (Kurita, *Pattern Recognition*, **24**(3):205-209, 1991).

10 Installation instructions

10.1 Unfold the system

To install Clustsys, first decide where you wish to place it. We recommend creating a separate directory for it, which we will refer to as <*ClustsysHome*>.

`% cd` <*ClustsysHome*> # change to the ClustsysHome directory

`% uncompress –c clustsys-tar.Z | tar xf –` # uncompress and untar the distribution file.

10.2 Compile the .c files

Now compile the `.c` files and create the library `cluster-analysis.o`

```
% cd src/C
% make
```

10.3 Set up environment variables

Several scripts in the Clustsys system need to know where the Clustsys source is installed. This information is provided to them by the environment variable "`CLUSTSYSPATH`" by placing the following command in your `.cshrc` file.

`setenv CLUSTSYSPATH` <*ClustsysHome*>

You also need to modify the file `Makefile.Clustsys` and change the definition of this variable in that file[7].

If you use a shell other than `csh`, you need an appropriate, equivalent command.

NOTE: The default value for `CLUSTSYSPATH` is assumed to be `~/Clustsys`, or `$HOME/Clustsys`. If that is where you have installed Clustsys then you do not need to define the above environment variable.

Additionally, you should also include <*ClustsysHome*>`/src/Shell` in the shell variable `path`.

`set path = ( $path` <*ClustsysHome*>`/src/Shell)`

10.4 Initialize Splus repository

Execute the command:

```
% init-splus
```

This command loads the Splus files into the repository of Splus, which by default is `~/.Data`. It needs to be done only once, unless you change the Splus files provided.

[7] I do not yet know how to conditionally assign to a `make` variable, i.e. assign a value if and only if there is not definition for an equivalent shell variable. If you know how to do this, please let me know and I will incorporate it in `Makefile.Clustsys`.

10.5 Fix Makefile.Clustsys

The `Makefile.Clustsys` uses the syntax `$(SYMBOL)` to expand the macro `SYMBOL`. This works fine on SunOS but breaks on DEC Ultrix. The latter prefers `$SYMBOL`, which SunOS does not like. The problem is that DEC Ultrix `make` introduces a blank after expanding `$(SYMBOL)` (but not when expanding `$SYMBOL`). This causes problems since `Makefile.Clustsys` creates filenames and command arguments by contactenating values of multiple symbols.

This problem is fixed by creating another file `Makefile.Clustsys.alternate` (which is identical to `Makefile.Clustsys`, except for using `$SYMBOL` instead of $(SYMBOL). This is done by the following command:

```
% make –f Makefile.Clustsys Makefile.Clustsys.alternate
```

All examples in this document use `Makefile.Clustsys`. To use the alternate file one may simply replace the name `Makefile.Clustsys` by `Makefile.Clustsys.alternate`.

11 Directory structure

The directory structure of the system is still evolving. The main issue is how to organize the various source files. Currently, these files are placed in different subdirectories, one for each type of language involved in the system, as follows:

```
src/Awk
src/C
src/Shell
src/Splus
src/field
src/refine
```

The contents of these directories are implicit from their name. The other directories are:

`doc` — contains documentation
`test.data` — contains some test data, input and output

12 Copyright

How to contact the author: e-mail `arun@cacs.usl.edu`. Snail-mail: The Center for Advanced Computer Studies, University of Southwestern Louisiana, P.O. Box 44330, Lafayette, LA 70506. Phone: (318) 482–6766. Fax: (318) 482-5791.

13 Acknowledgments

Clustsys has evolved over a period of time, starting from Summer 1991. Its development was initiated primarily to experiment with cluster analysis based subsystem classification methods. The system was designed for batch processing to enable experiments involving 60+ programs.

The system was not designed with the intent to distribute it for public use, nor was it expected to live this long. It is a classic case of a prototype turning into a product.

The design and implementation effort of Clustsys has been led by the author in collaboration with three other people. In the Summer of 1990, Paul Poolkasem and Sanjay Mohan participated in developing the initial version of this system. This version used Brown University's FIELD to extract cross-reference relations. In Summer of 1994, John Gravley developed the code to extract these relations using Software Refinery. In Summer 1995 he added several congruence measures (see Lakhotia & Gravley, 1995), reimplemented the `awk` scripts, and performed a "walk-through" of the entire system. Pablo Mejia ran thru a quick test of the system and the documentation, starting from installation and its use.

Paul Poolkasem is now at Softool Corporation, Sanjay Mohan at Altera, John Gravley is getting ever closer to completing his dissertation, and Pablo is slowly getting lured into graduate school.

I thank Paul, Sanjay, John, and Pablo for their participation in this work.

This work was supported have been supported by grants from Louiana Board of Regents (LEQSF 1991–92 ENH-98 and 193–95 RD-A-38) and US Army Research Office (ARO DAAH04–94–G-0334). However, the contents of this document or the implementation of this system do not reflect the position of the policy of the University, or the State or Federal government, and no official endorsement should be inferred.

14 Clustsys: Quick reference guide

14.1 Acquiring Clustsys

- URL `http://www.cacs.usl.edu/~arun`. Follow the link to an index of ftp-archive of software systems.
- `ftp basin.cacs.usl.edu` (IP number `130.70.32.51`)
  ```
  bin
  get pub/uslstuff/srl/clustsys-tar.Z
  ```

14.2 One-time initializations for the installation

- `% cd` <*ClustsysHome*>
- `% uncompress –c clustsys-tar.Z | tar xf –`
- `% cd src/C`
- `% make`

14.3 One-time initializations per user for using Clustsys

- Include the following, or equivalent, in the shell start up file(s).
  - `setenv CLUSTSYSPATH` <*ClustsysHome*>
  - `set path = ( $path` <*ClustsysHome*>`/src/Shell)`
- Logout/Login (or else source the shell files)
- `% init-splus`

14.4 Platform related dependencies

- On DEC Ultrix use `Makefile.Clustsys.alternate` instead of `Makefile.Clustsys`. This file may be created using:
  `% make –f Makefile.Clustsys Makefile.Clustsys.alternate`

14.5 Generate binding matrix

- `% make –f Makefile.Clustsys S=`*`pgmname`* `Q=`*`all`*
- `Makefile.Clustsys` variables
  - S: name of executable or `.prog` file
  - `Q`: `all`, `field`, or `globals`
  - `QUERY-SYS`: `query-refine` or `query-field`
  - `DIR`: directory containing the system (if other than the current directory).
- Files created: *`pgmname-all`*`.matrix` and *`pgmname-all`*`.labels`

14.6 Perform cluster analysis

- `% Splus –e`
- `> cload()`
- `> motif()` (or `X11()`), etc.
- ☐ `> demo.clustsys("pgmname-all.matrix")`
  ☐ `> demo.clustsys("pgmname-all.matrix", labelsfile="pgmname-all.labels")`
  ☐ `> demo.clustsys("pgmname-all.matrix", labelsfile="pgmname-all.labels", method = "e")`
- `> q()` (to quit Splus)