Common Lisp Analytical Statistics Package: User Manual

Experimental Knowledge Systems Laboratory
Computer Science Department, LGRC
University of Massachusetts
Box 34610
Amherst, Massachusetts 01003-4610

Scott D. Anderson
David L. Westbrook
Matthew Schmill
Adam Carlson
David M. Hart
Paul R. Cohen

with contributions by

David A. Fisher

CLASP Version 1.4.2
CLIP Version 1.4

January 17, 1995

© 1994, 1990 Department of Computer Science, University of Massachusetts, Amherst, Massachusetts.
This work was supported by a University Research Initiative grant, ONR N00014-86-K-0764, and by ARPA/Rome Laboratory under contracts F30602-91-C-0076 and F30602-93-C-0100.

Abstract

This document is the user manual for the Common Lisp Analytical Statistics Package (CLASP), a tool for visualizing and statistically analyzing data, and also for the Common Lisp Instrumentation Package (CLIP), a tool for automating data collection and experimentation.

Through statistics we attempt to separate those events which are simply due to chance from those which are related to other events in our world. CLASP provides the tools necessary for these tasks. CLASP and CLIP are implemented in Common Lisp.

There are two implementations of the CLASP user interface. One in CLIM, the Common Lisp Interface Manager, and one using the Macintosh toolkit. Along with describing the operation of CLASP and CLIP, this manual details the CLASP programming interface, an aid for building extensions to CLASP.

Contents

List of Figures

Chapter 1

Introduction

When events occur in the world, people turn to statistics to analyze and describe them. This is especially true of the behavioral sciences, psychology and sociology. Through statistics we attempt to separate those events which are simply due to chance from those which are a result of, or are related to, other events in our world. Through careful analysis it is possible to understand a great deal about the state of the world. In AI, we have a new aspect of behavioral science, the study of the behavior of intelligent agents and computer programs in complex environments, and so we turn to statistical analysis to describe and analyze their behavior and the factors that predict their behavior. Two examples of such AI programs are PHOENIX, a simulator of agents that fight forest fires in Yellowstone National Park, and TRANSSIM, a simulator of transportation plans in a global logistical problem. We expect that these simulated worlds will lend themselves to the same forms of analysis we use to study the natural world. Our goal is to smoothly connect the simulators and other computer programs with statistical analysis tools, written in Common Lisp and running in the same Lisp world, so that high quality statistical analysis will be conveniently available to help us understand and test the behavior of our programs.

Why use statistics to understand our programs? Why not just look at the code? As our programs because increasingly large and complex, studying the code becomes as effective in understanding overall behavior as studying biochemistry is in understanding animal behavior. The interaction of components and the response of the program to different circumstances of its execution make global behavior hard to predict. Furthermore, programs will act differently in different executions because of differences in their environments, and statistics can help us summarize, visualize, and analyze the overall behavior, abstracting away the details of particular executions. We implemented CLASP, the Common Lisp Analytical Statistics Package, to help AI researchers analyze the behavior of their programs.

1.1 History and Acknowledgments

This document describes the second implementation of CLASP. Many people helped develop CLASP since the idea was first conceived in the Experimental Knowledge Systems Laboratory (EKSL) in 1989.

First, David Fisher designed and implemented the statistical functions and graphical interface for the first implementation of CLASP. He was assisted by David Westbrook in many ways, but primarily in the TI Explorer windowing and graphics code.

Other people implemented specific statistical functions, including Sameen Fatima's im-

plementation of Chi-square analysis of two-way contingency tables and Dorothy Mammen's implementation of log-linear analysis for higher dimensional contingency tables.

Even before the first implementation of CLASP, EKSL was collecting data from systems like PHOENIX and statistically analyzing them. Therefore, we quickly started implementing general methods for instrumenting PHOENIX, collecting data, and running experiments. This effort was begun by David Westbrook and Scott Anderson, and eventually evolved into what we call CLIP, which is basically a set of macros to make it easier to run experiments.

The second implementation of CLASP was a joint effort by Scott Anderson, Adam Carlson and David Westbrook. Its goal was to make CLASP more convenient and generally available by simplifying the data representation, eliminating its dependence on the TI Explorer by using CLIM, the Common Lisp Interface Manager, and improving the user interface based on our experience with the first implementation. The CLASP CLIM interface is described in Chapter 4.

An additional implementation of CLASP was begun in the summer of 1994. This effort involved producing a native Macintosh interface and was done by David Westbrook and Matthew Schmill with Matt Cornell providing assistance. This new interface is described in Chapter 3.

1.2 Design

This manual roughly follows the design of CLIP and CLASP, and therefore breaks down into six sections. The first is a description of CLIP, our aids for automated data collection and experimentation. The other four sections are aspects of CLASP: its two graphical user interfaces, its data manipulation functionality, and its statistical functionality. We designed CLASP so that these aspects are fairly modular: the graphical user interface makes it easy to call the data manipulation and statistical subsystems, but those functions are separate from it and from each other. The functions of those subsystems can be called directly from a user's program or from a Common Lisp interpretor. Consequently, the sixth section describes the actual Common Lisp functions. Indeed, some individual functions, CLOS classes and methods are documented by automatically typesetting of the name, argument list, type, and documentation string, pulling this information directly out of the code.

1.3 Getting CLIP/CLASP

CLIP/CLASP is available via anonymous ftp from *ftp.cs.umass.edu* in the directories `pub/eksl/clip` and `pub/eksl/clasp`.

CLIP requires Common Lisp and the Common Lisp Object System, CLOS. CLASP for Unix requires these also, and in addition requires the Common Lisp Interface Manager, or CLIM, and SCIGRAPH, a publicly available scientific graphing package written in Common Lisp and CLIM by Bolt, Beranek and Newman (BBN). SCIGRAPH is included in the distribution. Information on CLASP for the Macintosh can be obtained by sending mail to the clasp support address listed below. CLIP/CLASP runs on a number of platforms under a variety of lisp implementations. Check the release notes for a detailed list of currently supported platforms.

If you have any problems, questions, or suggestions, contact us at:

`clasp-support@cs.umass.edu`

1.4 Conventions

Wherever lisp or CLASP objects appear in a body of text, the following conventions hold:

- Names of CLASP commands are printed in SMALL CAPS and prefixed with a semi-colon.
- Names of arguments to CLASP functions and commands appear in *italics*.
- Names of keywords are printed in `teletype` and are prefixed with a semi-colon.
- Names of all other lisp objects, including functions are printed in `teletype`.

Where sections of code are included, they will be in `teletype`. In examples of interactive CLASP sessions, san serif will be used and for anything the system prints and **bold san serif** will be used for user input.

Chapter 2

CLIP

2.1 Motivation

We collect information from software systems for many reasons. Sometimes, the very purpose of the system is to produce information. Other times, we collect data for debugging, feedback to the user—in general, for understanding the system's behavior. Unfortunately, there seem to be few general tools available for this task. Instead, we often find ad hoc, domain-specific code, spread throughout a system, varying from component to component, despite strong similarity in the requirements for any data collection code.

CLIP, the Common Lisp Instrumentation Package, standardizes data collection. It is named by analogy with the "alligator clips" that connect diagnostic equipment to electrical components. By standardizing data collection, we can more easily add and delete instrumentation from our systems. We can reuse a piece of data collection code, called a "clip," in other systems we are studying. We can use general tools to analyze the collected data. Instrumentation in a system becomes more easily used and understood, because the basic functionality of the system is separated from the functionality of the instrumentation.

We designed CLIP to be used for scientific experiments, in which a system is run under a number of different conditions to see whether and how the behavior changes as a function of the condition. Consequently, we view an experiment as a series of trials, varying the settings of experiment variables and collecting information. Generally speaking, an experiment comprises the following steps:

1. Creating clips and inserting them into the system to be studied. Generally, clip measures one particular aspect of a system, but you can also define clips that combine many measurements, including other clips. Often, the clips you need will already be defined.

2. Define the experiment, in terms of what system is to be run, any necessary initialization code, and the conditions under which it is to run (different input parameters or environmental conditions).

3. Run a series of trials, saving the output into a CLASP format data file.* This format is described in the CLASP manual section Data Manipulation Functions, although it isn't necessary to understand the file format. CLASP can read what CLIP writes.

*CLIP can also write data files in standard tab or space delimited format used by most other statistical packages, databases and spreadsheets.

4. Analyze the data using CLASP.

2.2 Implementation

CLIP provides five major forms to support experiments:

- `define-simulator`,
- `defclip`,
- `define-experiment`,
- `write-current-experiment-data`, and
- `run-experiment`

Each is described in detail below.

2.2.1 Define-simulator Macro

`define-simulator` *(name &key system-name system-version reset-system start-system stop-system schedule-function deactivate-scheduled-function seconds-per-time-unit timestamp)* [*Macro*]

Define the interface to a simulation. The following options are recognized:

`:system-name`	string naming system	[2]
`:system-version`	function or form which handles the arguments of the experiment and returns a string which denotes the version of the system [a]	[1]
`:start-system`	function or form that handles the experiment variables and arguments of the experiment; this function is called during the experiment loop to begin execution of the system; when it returns the trial is considered to be completed	[2]
`:reset-system`	same as :start-system; this function will be called during the experiment loop before the system is started;	[1]
`:stop-system`	same arg handling as :start-system; this function can be used to execute code when a trial is shutdown; it is most useful when instrumenting multiprocessing systems where other processes need to be terminated	[1]
`:schedule-function`	function or form that handles the lambda list *(function time period name)* and optionally returns a data-structure which represents the event; used to provide access to the simulator's event-scheduling mechanism	[3]
`:deactivate-scheduled-function`	function or form that handles one arg, namely the data structure returned by the `schedule-function` function; used to provide access to the simulator's event-scheduling mechanism	[1]
`:timestamp`	a function or a list of (<function-name> <clip-name>) where function should return the current time in units specified by `:seconds-per-time-unit`	[3]
`:seconds-per-time-unit`	the number of seconds in 1 simulator time quantum (default is 1)	[3]

Keywords marked [1] are optional; Keywords marked [2] are required, and keywords marked [3] are required when using time-series clips

[a]Functions must accept the formal arguments specified in `define-experiment` and the actual arguments specified in a call to `run-experiment`. Forms can refer to the arguments lexically.

The `define-simulator` form is used to eliminate the need to specify redundant information when multiple experiments are to be defined for the same system. All of the options specified in this form can be overridden in a `define-experiment` form.

2.2.2 Define-experiment Macro

`define-experiment` *(name arguments &body body &aux documentation)* [*Macro*]

A `define-experiment` form sets up the environment in which the system is to be run. Options for `define-experiment` support the sequential nature of experimentation. That is, for the purposes of alligator clips, we view an experiment as a series of trials, varying the settings of experiment variables and collecting information. The `define-experiment` macro allows code to be run at the beginning and end of the experiment, usually to set up data-recording or to do data analysis. It also allows code to be run at the beginning and end of each trial; often this is used to reset counters, clear data structures, or otherwise set up the data collection apparatus. Through the `:script` keyword a user may add specific code to run at particular times during the experiment. A user can also specify experiment variables and their associated sets of values; the experiment code will ensure that all combinations of the experiment variables are run. For example, specifying:

```
:variables ((rating-scheme '(:OPPORTUNISTIC :ORDER-BY-ORDER))
      (percentage from 100 downto 50 by 10))
```

describes an experiment with twelve conditions (six values of percentage times two kinds of rating schemes), and so the number of trials will be a multiple of twelve, with an equal number of trials being executed under each condition. The options accepted by `define-experiment` are:

`:simulator`	specify the name of the simulator definition to use
`:before-trial`	called with the current values of the experiment variables and arguments to specify operations performed before each trial (such as initializations)
`:after-trial`	similar to `:before-trial`, specifies operations to be performed *after* each trial. Summary data files for each trial can be written at this time using `write-current-experiment-data`.
`:before-experiment`	called with the arguments. The value should be a form that refers to the arguments or a function that handles the correct number of arguments. Used to initialize system and define experiment environment.
`:after-experiment`	Similar to `:before-experiment`, called with just the arguments. Used to close files, write any experiment summarization output and clean up after the experiment.
`:instrumentation`	is a list of names defined with `defclip` which will be enabled during the experiment. Use `write-current-experiment-data` in the after-trial code to write the data to the output-file.
`:variables`	({(var exp) | (var {loop-for-clause})}) - define experiment variables
`:locals`	({(var exp) | var}*) - bind variables in the context of the experiment
`:script`	({(descriptive-name initial-time [next-time] form) — script-element-name}*) *initial-time* can be a string, a number or form to evaluate. *next-time* is a time interval and should be a fixnum or form.

In addition, `define-experiment` accepts all the options accepted by `define-simulator`.

The following steps are generated from the `define-experiment` specification to run a typical experiment. After initializing the system using the `:before-experiment` form, it loops over the cross-product of the specified experiment variables, running one trial for each element in the cross-product. This is repeated for as many times as is specified in the `:repetitions` option. After the last trial, the `:after-experiment` form is run.

```
Run :BEFORE-EXPERIMENT code with args
LOOP
  Update experiment-variables for trial
  Run :BEFORE-TRIAL code with experiment-variables and args
  Run :RESET-SYSTEM code with experiment-variables and args
  Instantiate Script Events
  Reset and Enable all the instrumentation
  Run :START-SYSTEM code with experiment-variables and args
  <system runs (possible periodic and event based collection done)>
  Run :AFTER-TRIAL code (possible post-hoc collection done)
END LOOP when all trials completed
Run :AFTER-EXPERIMENT code with args
```

2.2.3 Write-current-experiment-data Function

`write-current-experiment-data` *(&key separator format instrumentation stream)* [*Function*]

Causes each experiment instrumentation to write its data to *stream* or the output-files specified in the `run-experiment` call or in a `defclip`. *separator* should be a character which will be used to separate fields. It defaults to the value of `*data-separator-character*`. *format* should be one of `:CLASP` which means write a clasp native format data file, `:ASCII` which means write a standard separator delimited data file including column names or `:DATA-ONLY` which is the same as `:ASCII` except no column names are included. *format* defaults to the value of `*output-format*`. *instrumentation* can be used to specify a subset of the experiment's instrumentation to write to the data file.

2.2.4 Run-experiment Function

`run-experiment` *(experiment-name &key args repetitions number-of-trials length-of-trial output-file error-file extra-header starting-trial-number)* [*Function*]

Run-experiment starts execution of the experiment named *experiment-name*. The *args* are passed on to the experiment. *output-file* is optional, but must be specified if `write-current-experiment-data` is called from within your experiment. *number-of-trials* can be used to specify an exact number of trials to run. If *number-of-trials* is not specified it will be calculated so as to vary all the experiment variables across all their values *repetitions* (default 1) times. *starting-trial-number* can be used to change this value to something other than one (1) which is useful for continuing partially completed experiments. *length-of-trial* can be specified for time-based simulators to put a limit on the maximum trial length. *error-file* can also be used to direct the error/debug output to a file. *extra-header* is written at the end of the header of the output file.

2.2.5 Explicitly Stopping Trials

The following functions can be called from user code to explicitly terminate a CLIP trial or experiment. Usually this is done from within the code of a script or as part of an error handler. They all take no arguments and assume that an experiment is in progress.

`shutdown-and-rerun-trial` *()* [*Function*]

> This will cause the current trial to be aborted (no data written) and restarted. This is a function that users should call when they have detected an error condition of some sort that renders the trial worthless, but rerunning the trial may work.

`shutdown-and-run-next-trial` *()* [*Function*]

> This will cause the current trial to be stopped (data will be written) and the next trial started. This is a function that users should call when want to normally shutdown a trial and collect and report the data from that trial.

`shutdown-experiment` *()* [*Function*]

> This will cause the current trial to be aborted (no data will be written) and will return the system to the state it was before the experiment began (by running the after-experiment code).

2.3 Clip Definition

There are basically only a small number of ways to instrument a software system. These are: adding special purpose code to collect data, interrogating global (usually objects with state) data structures or using advice.

The first way is to build in special purpose code to collect data about the state of the system while the system is running. We used this technique in the Phoenix¡ testbed to keep information about the execution time of timeline entries and also message traffic patterns. The Transsim simulator also uses this technique when it keeps track of daily costs, ship locations, demon firing intervals. The key point here is that the only reason for adding the piece of code was to allow an experimenter to analyze the behavior of the system. The simulation itself does not use the information. This method increases the complexity and reduces the readability and maintainability of the software system. In code that is highly instrumented it is often difficult to determine what is intrinsic to the running of the system and what is used to instrument.

Another method involves interrogating global data structures to determine the state *post hoc*. Examples include most everything we collected about Phoenix (ie., fireline built, first plan tried, bulldozer digging times, etc.). This technique is fine if the information is hanging around after you have finished running the simulation, but this is not always the case. Also, there is some information that is time-sensitive, and therefore must be collected on a periodic or event-driven basis. Collecting this type of information often involves resorting to the first method – altering the code itself.

Alternatively, one can use the `advise` facility available in many lisp environments to non-intrusively collect information when functions are called. This is a nice modular approach, but requires a good deal of knowledge about the advise facility and the software system itself. Unfortunately, the `advise` facility is not standardized across Lisp implementations.

The `defclip` macro encapsulates the code necessary to instrument a software system and separates it from the system itself. It avoids the pitfalls of adding special purpose code directly to the software system while at the same time allowing periodic and event-driven data collection. It also allows collection to be done by perusing global data structures.

2.3.1 Simple Clips

Simple clips have no components and collect data immediately prior to reporting it to the output file at `:after-trial` time. If they are defined with a `:schedule` or `:trigger-event` `defclip` option their default behavior is store all of the data collected during a trial and report a single value which is the mean of all the values collected.†

2.3.2 Clips with Components

Clips with components, as specified by the `:components` keyword, generate multiple columns in a data file each time they are reported. Depending on other options they may produce one column per component (*composite clips*) or multiple columns per component (*mapping clips*). Mapping clips are specified using the `:map-function` option to `defclip`. Clips with components are sometimes referred to as *super clips*. For a good example of clips with components and further discussion of their use, see Appendix B.1.1.

2.3.3 Time-series Clips

Clips that have `:components` and either the `:schedule` or `:trigger-event` option are *time-series clips*. They generate multiple data columns in the manner of component clips (which they are) and also multiple data rows. Each row corresponds to a single collection and is either triggered by a particular event or activated periodically. Since time-series clips generate multiple rows, they are generally written to a data file that is separate from the main experiment (summary) data file. The name of the data file associated with a time-series clip is specified using the `:output-file` option to `defclip`.

The `:schedule-function`, `:seconds-per-time-unit`, and `:timestamp` keywords to `define-simulator` must be specified for periodic time-series clips. Event-based time-series clips require only that the Common Lisp implementation provide some mechanism similar to the `advise` function.‡

2.3.4 Defclip Macro

`defclip` *(name args (options) &rest body)* [*Macro*]

A `defclip` form defines and names a particular data-collection instrument. Each clip collects a single variable which, together with other variables, make up a dataset. The dataset can then be analyzed by CLASP. The form may contain arbitrary code to gather data for the clip: access to global variables, function calls, and so forth. Naming the clips makes it easy to turn them on and off programatically or with menu selections.

options is a list of keywords and values which modify the main form implemented by the clip. *options* can be any of the following, all of which are optional:

†The default behavior if the collected values are non-numeric is to generate an error.

‡Actually, one can explicitly call the undocumented (until now), unexported 'clip::collect' function on a particular instrumentation and achieve the same effect as an event-based clip, but requires modifying the code.

- `:schedule` a list of keyword/value pairs; currently allowed are `:period` which provides the time interval between collections for time-series data, and `:start-time` which specifies the time of the first collection; if neither `:schedule` nor `:trigger-event` is specified, collection will be done immediately prior to the report being output to the data stream
- `:trigger-event` a function, list of functions or list of trigger specifications which trigger event-driven collection ; each trigger spec should of the form (<fname> [:BEFORE | :AFTER] [:PREDICATE <fname>])
- `:components` a list of clips that are associated with this clip; ie., they are collected and reported as a group by collecting or reporting this clip
- `:map-function` provides a list of items to map the `:components` over ; this function should return the same arguments in the same order each time it is called
- `:report-key` allows overriding of the column header key which is written to the data stream; avoid using this for component clips unless you really know what you are doing as the default key will be generated to correctly differentiate between multiple invocations of the same clip with different parents; this string is used in a call to 'format'; for most component clips this string should handle the same arguments as the clip
- `:initial-status` whether the clip is enabled or disabled by default
- `:report-function` can be used to override the default report function; for expert users only
- `:enable-function` code[a] to set up data structures for the clip; runs once when the clip is turned on
- `:disable-function` code to remove data structures set up for the clip; runs once when the clip is turned off
- `:reset-function` code to reinitialize data structures at the beginning of each trial; for example, setting counters back to zero
- `:display-function` code for graphical display of the information
- `:output-file` used to specify the output file for time-series clips - merged with the pathname specified in the `run-experiment` call; should be a string or a function of one arg; the function is called with the name of the clip and should return a value suitable for use as the first argument to `merge-pathnames`

[a] 'code' at this level is specific to the user's system, and is assumed to be user-supplied Lisp code. For example, enabling a counter of events in a system may require creation and initialization of a counter variable.

Some examples of `defclip` usage:

```
A simple clip with code used to report a value:
  (defclip number-of-dead-bulldozers ()
    (length (bds-that-died)))
```

```
An example showing a clip with components:
  (defclip methods-each-bd (bulldozer)
   "Salient information for each instance of applying a recovery method:"
   (:components (trial-number
                 agent-name
                 method-type
                 failure-type
                 calculate-recovery-cost
                 method-succeeded-p
                 order-of-application)
    :map-function (gather-recovery-method-instances (name-of bulldozer)))
   ;; This code executes before the map-function is executed.
   (send (fire-system) :set-frame-system (name-of bulldozer)))

   (defclip ii-projection-attempts-made ()
     "Clip for reporting the number of attempts that were made by
ii-projection during the last projection of the trial."
     (:reset-function (setf *ii-projection-attempts-made* nil))
     (car *ii-projection-attempts-made*))
```

2.4 Examples from a Simple Agent Simulator

The following three examples illustrate experiment control and data collection using CLIP. The agents in this simulator stochastically increment their state until they reach a final state. The first example defines the experimental interface to an agent simulator and two simple clips for collecting data. The second and third examples build on the first, defining more complicated clips for mapping a simple clip onto multiple agents and for collecting time-series data.

2.4.1 Using Simple Clips to Collect Trial Summary Data

This example collects trial summary data about overall agent-cost and task completion-time. Collection occurs at the end of each trial and is written out to a summary file in CLASP format. The `define-simulator` form designates methods for starting the simulation at the beginning of the experiment, resetting and reinitializing it if necessary, and stopping it after the final trial. The `define-experiment` form designates the simulator to use, in this the case the one we have defined. It also specifies: a set of experiment variables and the experimental settings for each, a list of the clips we've defined under `:instrumentation`, initializations for global variables before each trial, and a function for writing out a row of data at the end of each trial.

For each trial, the trial number (assigned sequentially) and the value of each experiment variable are written out to the summary file along with the values of the specified clips. An example of the CLASP output file produced by this experiment is shown in Figure 2.1.

In this first example we have defined two simple clips. One collects the combined cost of all agents at the end of each trial and the other records the time at which the trial ends.

```
;;;-------------------------------------------------------
```

```
(define-simulator agent-sim-1
  :start-system (run-agent-simulation :reset nil)
  :reset-system (reset-agent-simulation)
  :stop-system stop-simulation)

;;;------------------------------------------------------
;;; Clip Definitions

(defclip agents-cost ()
  ()
  (reduce #'+ (find-agents) :key #'cost))

(defclip completion-time ()
  ()
  (current-time))

;;; *****************************************************
;;; The Experiment Definition

(define-experiment simple-agent-experiment-1 ()
  "A test experiment."
  :simulator agent-sim-1
  :variables ((transition-probability in '(.01 .1))
              (cost-factor from 1 to 3))
  :instrumentation (agents-cost completion-time)
  :before-trial (setf *transition-probability* transition-probability
                      *relative-cost* cost-factor)
  :after-trial (write-current-experiment-data))

#| Execute this to run the demo experiment.

(run-experiment 'simple-agent-experiment-1
                :output-file
                #+Explorer "ed-buffer:data.clasp")
|#
```

2.4.2 Using a Mapping Clip to Map Simple Clips Over Multiple Agents

This example defines a mapping clip that maps over all the agents at the end of each trial and returns the cost accrued by each. It produces a summary output file like that in Figure 2.2 recording after each trial an entry that includes trial number, each experiment variable, each of three agent's cost, and the completion time of the trial.

```
;;;------------------------------------------------------
;;; Mapping clip to collect cost of each agent

(defclip all-agents-costs ()
```

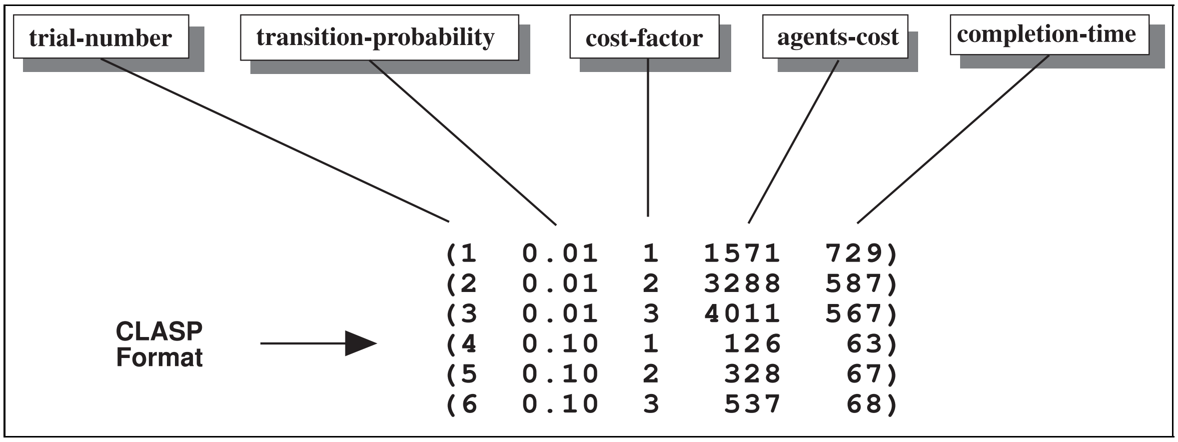

Figure 2.1: Simple Clip Example from the Agent Experiment.

```
  (:map-function (find-agents)
   :components   (each-agent-cost)))

(defclip each-agent-cost (agent)
  ()
  (cost agent))

;;; ****************************************************
;;; The Experiment Definition

(define-experiment simple-agent-experiment-2 ()
  "A test experiment."
  :simulator agent-sim-1
  :variables ((transition-probability in '(.01 .1))
              (cost-factor from 1 to 3))
  :instrumentation (agents-cost all-agents-costs completion-time)
  :before-trial (setf *transition-probability* transition-probability
                      *relative-cost* cost-factor)
  :after-trial (write-current-experiment-data))

#| Execute this to run the experiment.

(run-experiment 'simple-agent-experiment-2
                :output-file
                #+Explorer "ed-buffer:data.clasp")
|#
```

2.4.3 Full Agent Simulator Experiment with Time-Series Clips

This example uses both periodic and event-driven time-series clips to collect data about each agent's state. To use time-series clips the experimentor must supply CLIP with enough information to schedule data collection *during* trials. This is done by specify-

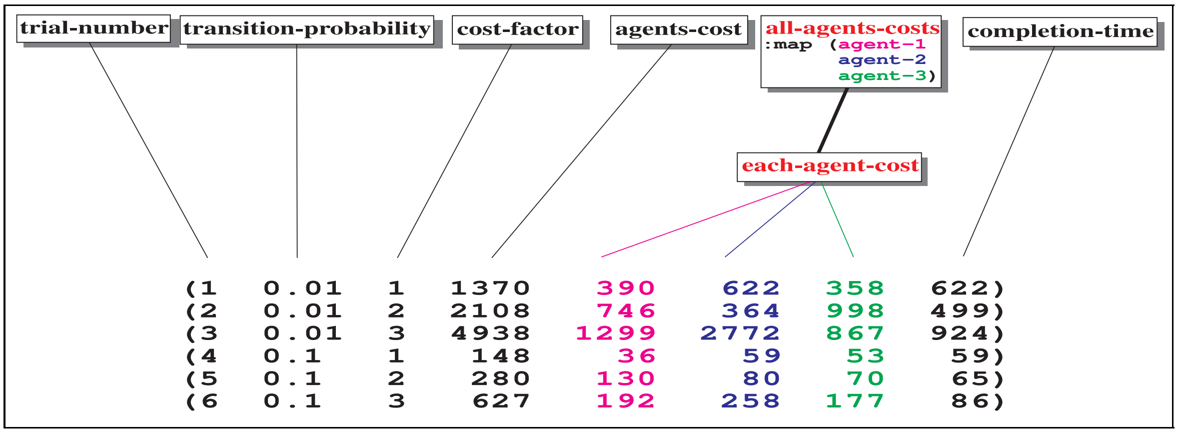

Figure 2.2: Mapping Clip Example from the Agent Experiment. Mapping clips are used to map one or more simple clips over multiple objects.

ing in **define-simulator** a **:schedule-function** that tells **defclip** how to schedule clip execution. The optional **:deactivate-scheduled-function** provides a function for unscheduling clips (if necessary). **:seconds-per-time-unit** tells CLIP how to translate the time units of the simulator into seconds. The value of `:timestamp` is used to automatically record the time of collection in each row of a time-series data file.

This example illustrates three uses of clips: collecting summary data about the highest agent state, gathering periodic snapshots of all agent states, and recording particular aspects of the agents' changes of state. The periodic clip definition specifies an output file and a scheduling interval. Two event-driven clips are triggered in this example by the same function, **change-of-state-event-function**, which looks for any change of agent state. However, since their component clips are not identical, their output is routed to separate files (CLASP file format expects all rows to have the same number of columns and column names).

Four output files are produced by this experiment. As in the previous examples, a summary file records a row of data at the end of each trial. Each change in an agent's state triggers event-driven clips that record a row in a time-series output file, as in Figure 2.3. The event-driven clip definitions **change-of-state** and **change-of-state-pred** produce one output file each. The fourth output file also records time-series data, one row for each periodic snapshot of all agents' states (cf. Figure 2.4).

Collecting time-series data is more time-consuming than collecting summary data. In the first two examples, where only summary data was collected, trials were allowed to run to completion. In the third example periodic clips run every 12 minutes and event-driven clips run as frequently as agent's states change. We can limit the duration of trials by specifying a **:length-of-trial** value to **run-experiment**. This is useful for statistical tests on time-series data, such as cross-correlation, that expect the number of rows for each trial to be the same. Thus, in this example, specifying `:length-of-trial` to be 500 minutes ensures that the periodic data collection will produce the same number of rows for each trial.

```
;;;-----------------------------------------------------


(define-simulator agent-sim
  :start-system (run-agent-simulation :reset nil)
```

```
  :reset-system (reset-agent-simulation)
  :stop-system stop-simulation
  ;; a function that places functions to run on the queue of events.
  :schedule-function (lambda (function time period name &rest options)
                       (declare (ignore name options))
                       (schedule-event function nil time period))
  ;; a function that removes functions from the queue of events.
  :deactivate-scheduled-function unschedule-event
  :seconds-per-time-unit 60
  :timestamp current-time)

;;;-----------------------------------------------------
;;; Clip Definitions

;; This post-hoc clip produces two values.
(defclip highest-agent-state ()
  (:components (highest-state highest-agent))

  (loop
    with agents = (find-agents)
    with highest-agent = (first agents)
    with highest-state = (state highest-agent)
    for agent in (rest agents)
    for agent-state = (state agent) do
    (when (state< highest-state agent-state)
      (setf highest-state agent-state
            highest-agent agent))
    finally (return (values highest-state highest-agent))))

;;;-----------------------------------------------------
;;; Periodic collection

;; This clip invokes its component every 12 minutes.
(defclip periodic-agent-state-snapshot ()
  (:output-file "snapshot.clasp"
   :schedule (:period "12 minutes")
   :map-function (clip::find-instances 'agent)
   :components   (each-agent-state-snapshot)))

;; Simple clip that returns the state of the agent.
(defclip each-agent-state-snapshot (agent)
  "Record the state at an agent."
  ()
  (state agent))

;;;-----------------------------------------------------
;;; Event-driven collection
```

```
;; This clip accepts the arguments passed to 'change-of-state-event-function'
;; and simply passes them through.
(defclip change-of-state (agent-name new-state)
  (:output-file "state-change.clasp"
   :trigger-event (change-of-state-event-function :BEFORE)
   :components (new-state agent-name))
  (values new-state agent-name))

;; This clip accepts no arguments and returns two values computed by other
;; functions.
(defclip change-of-state-pred ()
  (:output-file "state-change-pred.clasp"
   :trigger-event (change-of-state-event-function :AFTER)
   :components (fred barney))
  (values (compute-fred) (compute-barney)))

;;; ************************************************
;;; The Experiment Definition

(define-experiment agent-experiment ()
  "A test experiment."
  :simulator agent-sim
  :variables ((transition-probability in '(.01 .1))
              (cost-factor from 1 to 5 by 2))
  :instrumentation (agents-cost
                    all-agents-costs
                    completion-time
                    highest-agent-state
                    change-of-state-pred
                    change-of-state
                    periodic-agent-state-snapshot)
  :before-trial (setf *transition-probability* transition-probability
                      *relative-cost* cost-factor)
  :after-trial (write-current-experiment-data))

;; Execute this to run the demo experiment.

(defun rexp ()
  (run-experiment 'agent-experiment
                  :output-file "data.clasp"
                  :length-of-trial "500 minutes"))
```

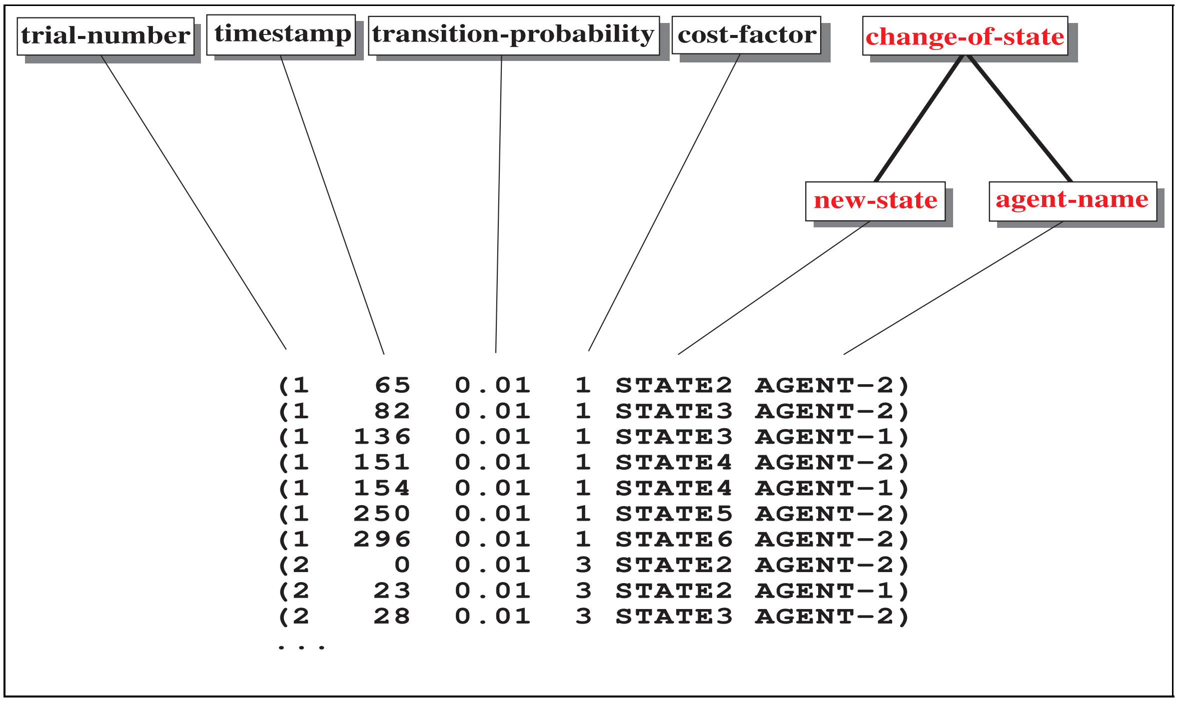

Figure 2.3: Event-Driven Time-Series Clips are used to collect data when an agent's state changes in the Full Agent Experiment.

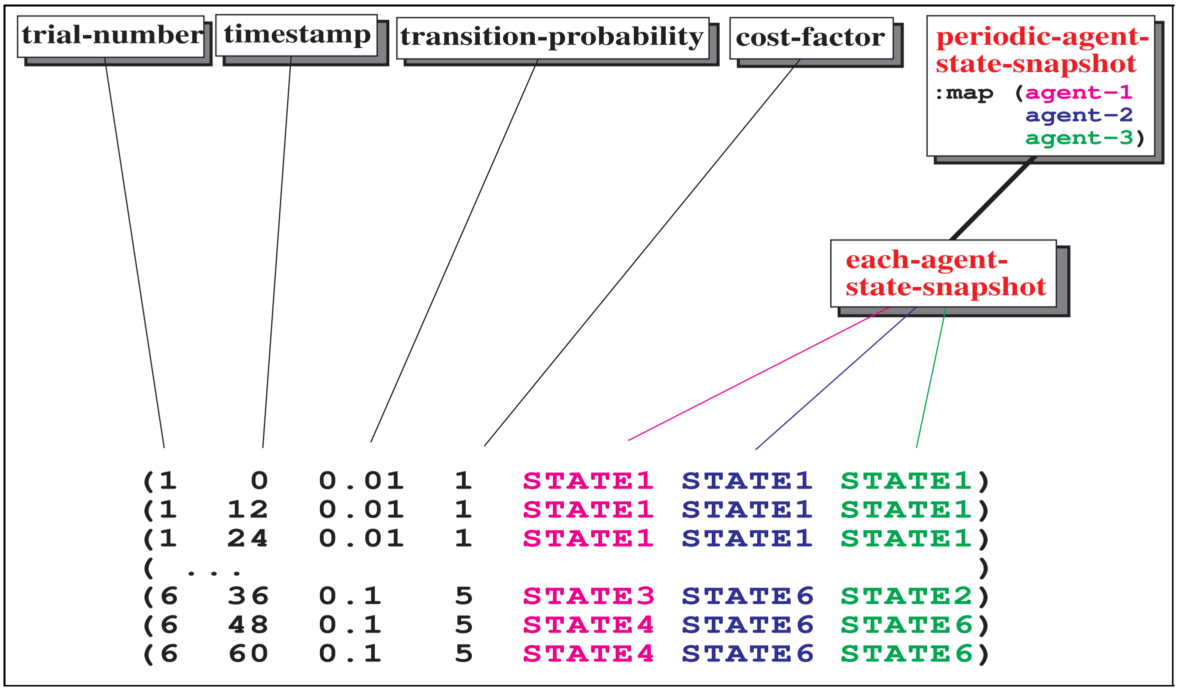

Figure 2.4: Periodic Time-Series Clips collect snapshots of each agent's state at regular intervals during the Full Agent Experiment.

Chapter 3

Macintosh User Interface

This chapter describes the Macintosh CLASP graphical user interface. The interface was built using standard Macintosh dialog resources, so the result is an interface much like that of most Macintosh applications.

3.1 CLASP Layout

The Macintosh interface to CLASP is organized as a collection of windows and a menubar. The menubar is used to initiate analyses and open windows, and each window provides some form of specific CLASP input or output. Each type of window is now considered along with the semantics of mouse clicks pertaining to each.

CLASP Window The CLASP window consists of five views. Each displays a specific category of information and provides a different meaning for mouseclicks.

Dataset View the dataset view resides in the left 1/3 of the CLASP window and lists all active datasets in bold and their variables below them, indented and in plain text.

option-click sends selection to CLASP x dock

control-click sends selection to CLASP y dock

caps-lock-click sends selection to CLASP cache dock

double-click opens selection for viewing/editing if easy-open preference is set

History View This view resides in the right upper 2/3 of the CLASP window and lists a transcript of all CLASP operations, including commands issued and their results.

option-click sends selection to the CLASP Notebook

double-click opens any openable items in the current selection for viewing

Comment View This provides one line of editable text and resides below the history view. Pressing return inserts the comment text before the first selected item in the history view.

Insert Button The exclamation mark to the right of the comment view. Clicking this button inserts the comment text before the first selected item in the history view.

Status View The progress and status of CLASP are reported in this view, which is found directly below the comment view.

Clasp Ribbon The CLASP Ribbon is a floating window which is used to display long names which may not fit in the smaller CLASP windows. Many objects, when selected, will display their verbose title in the CLASP Ribbon. Clicking on the CLASP Ribbon will toggle its location between the top and bottom of the screen. Choosing the mitem Cut from ribbon menu item in the File menu will cut the contents of the ribbon window to the clipboard.

Palettes These are windows that present a list of some type of result. Currently there is a graph palette and a table palette. Items in a palette may be deleted via the delete button. Special clicks also have the following meanings:

double-click opens the selected items for viewing

control-double opens a preference window for the selected items (graph palette only)

Notebook The CLASP Notebook is a specialized palette. It is used for storing important items reported into the history buffer. Anything in the history buffer may be sent to the Notebook. Like other palettes, items may be deleted and opened from the Notebook. The Notebook also provides a log button for logging its contents to an Apple Events aware document processing application.

double-click opens the selected items for viewing

control-double opens a preference window for selected graphs

Variable Dock The CLASP variable dock is a floating window (always in front) which consists of three "subdocks", the X Dock, the Y Dock, and the Cache. Items which are frequently used as arguments to CLASP functions are best stored in the dock, as the arguments to dialogs are initialized using the selected contents of the appropriate subdock* (or the entire contents, if there is no selection). Items may be brought to the subdocks by drag-and-drop or special clicks in the Dataset View. Each subdock also has a set of buttons below it for clearing the selection (or the entire subdock if there is no selection) and sending selections (or the entire contents if there is no selection) to the other subdocks. The CLASP variable dock supports these special clicks:

option-click replaces the y arguments of the currently selected analysis dialog with the selection in the clicked subdock

control-click replaces the x arguments of the currently selected analysis dialog with the selection in the clicked subdock

caps-lock (in conjunction with one of the above) indicates that the selected item(s) should be added to the argument slot, rather than replacing it

double-click opens the selection, if the easy-open preference setting is on

*X variables are taken from the X dock, Y from the Y dock, and should there be an extra variable slot, it will take the contents of the cache as its default value.

Analysis Dialogs MACINTOSH CLASP Analysis Dialogs are windows which allow arguments to be specified for CLASPś functions. All of the graphing, statistical, and sampling functions create a unique dialog with which to gather the appropriate arguments. Special clicks have no meaning in these windows.

Graphs CLASP Graphs are visual representations of data. When a graph is generated, it generates a `#.(GRAPH-TYPE ...)` item. This item, when double-clicked, will open a CLASP Graph window to view the results, and may be set in the CLASP Preference window to automatically open when generated. CLASP Graphs may be resized and the default size of graph windows can also be set in the CLASP Preference window.

control-double opens a preferences window for the graph

Tables Many CLASP functions produce items in the history that appear as `#.(STATISTICAL-TABLE...)`. Like graphs, these can be double-clicked to be opened into their own window for viewing, and may be set to open automatically by setting the AUTO OPEN TABLES preference. For tables containing

double-click displays observed frequencies (contingency table)

control-click displays proportions of row totals (contingency table), scheffe statistic (scheffe table)

option-click displays proportions of column-totals (contingency table), significance (scheffe table)

Lisp Listener CLASP Provides a Lisp Listener for users who wish to add their own code or use CLASP via the functional interface.

3.2 Clasp Preferences

CLASP provides a preferences system for customizing the environment and individual CLASP items such as graphs.

3.2.1 General Preferences

General preferences can be accessed through the Preferences Window by selecting the Preferences menu item from the File menu, or it's command key equivalent command-'.

- **Confirm Quit** controls whether quitting CLASP requires confirmation from the user or not.
- **Include Labels** determines whether the first line of an imported/exported data file is expected to be the column labels.
- **Bypass Dialogs** controls whether or not argument dialogs will be opened if enough arguments have already been specified (via the variable dock).
- **Auto Open Graphs** determines whether graphs will be opened automatically when generated.

- **Auto Open Tables** determines whether the primary table generated by an analysis will be automatically opened when generated.
- **Easy Open Vars** determines whether double clicking a variable or dataset will open it for editing.
- **Confirm Deletes** determines whether CLASP will ask for confirmation when deleting datasets and/or variables.
- **Use Ribbon** determines whether the CLASP Ribbon window should be present.
- **Space Delimit Tables** will use spaces instead of tabs when generating tables for the clipboard. Make sure to use a fixed width font with this option.
- **Log Output** is the logging type preferred for use with the notebook. Currently, event logging is the only supported type. See section 3.7 for more information on logging.
- **File Delimiter** is the character used to separate data items in imported and exported file formats.
- **Def. Graph Size** specifies the default size of graph windows X,Y in pixels.
- **Def. Log Size** specifies the default size to write graphs to the log in X,Y pixels.
- **Table Font** specifies the font attributes to use for table entries. Table headers and titles are derived from this font. It is suggested that this be set to :PLAIN and a non-proportional font for viewing floating point values.
- **Data Font** is the font specification for viewing variables and datasets via the Edit command. It is also suggested that this font be non-proportional for easy viewing of floating-point numbers.
- **View Font** is the font with which to derive all other CLASP text. It is used in the CLASP window, the notebook, and the palettes.

3.2.2 Graph Preferences

CLASP provides a facility for editing individual graphs. Control-clicking a graph item or choosing the Edit command from the graph menu will open a preferences window for the selected graph. Some preferences are optional and may be left out by not checking the provided check box, or specified by checking the check box and specifying the preference. The preference window contains the following features:

- **Custom Bounds** make it possible to specify the boundaries of the graph to be any real numbers. To specify custom graph boundaries, the Custom X (or Y) bounds box must be checked and the custom bounds entered in the provided boxes. If no custom bounds have been specified, default bounds are pre-calculated from the graph data.
- **Plot Groups** make up the complete graph. Graphs are made up of distinct plot groups which share attributes such as color. Graphs which have been colored by a variable are partitioned into plot groups which are then individually colored and graphed. All of the preference items below the list of plot groups pertain to individual plot groups rather than the graph as a whole. Selecting a plot group will indicate that any changes made to those preferences will apply to that plot group.

- **Color** may be specified via the pull down menu by selecting the color desired. Selecting the CUSTOM item activates the RGB items, which can be used to indicate an RGB specification of a 48 bit color[†], and the pick button, which opens the Macintosh color picker for selecting custom colors. RGB intensities are legal in the ranges 0-65535.
- **X Shift** specifies the amount to shift a line or row-line plot by when it is displayed.
- **Custom Symbols** allow a custom symbol to be used by a scatterplot. Any plot group which is a scatterplot may take a symbol from the custom symbol menu as its icon.[‡]
- **Histogram Bins** specifies how many bins the histogram should have for a continuous variable. This item is not activated unless the graph being edited is a histogram.

3.3 Clasp Menus

CLASP Menus provide access to statistical commands by opening appropriate dialogs for gathering the arguments of each command. There are nine menus in the Mac CLASP interface, each offering a distinct category of functionality to the user. A description of each menu item along with an explanation of the arguments required by the corresponding CLASP command follows. Arguments listed as X or Y are CLASP variables unless otherwise noted, and the presence of *(s)* denotes that multiple arguments are accepted. In such cases, CLASP performs a mapping of these arguments to produce a series of operations. The standard mappings are as follows:

Univariate operations use straight mapping, performing the operation on each X argument in turn, using all numeric input across all X(s).

Bivariate operations, multiple X(s), one Y; compute the operation for each X on the single Y argument. The single exception is with multiple linear regression, which performs the analysis with all X variables on the single Y.

Bivariate operations, equal number of X and Y arguments, use lexical mapping. That is, the first X variable is paired with the first Y, and so on.

Bivariate operations, $X(s) > 1$, $\#(X(s)) \neq \#(Y(s)) > 1$, perform lexical mapping on all X(s) which have a corresponding Y, ignoring extra Y(s) or X(s) with no match.

Multivariate operations consider variates in addition to X(s) and Y(s), such as partial correlation. Such operations use lexical mapping across the variates.

3.3.1 Apple Menu

CLASP installs two menu items under the Apple menu, one for information and one for help. About opens a window containing information about CLASP its authors, and the UMass EKSL. Help opens a window containing documentation pertaining to the currently selected window.

[†]To my knowledge, there are no monitors which provide support for more than 24 bit color available for the Macintosh. The system automatically quantizes RGB values as appropriate.

[‡]The character symbols tend to render slower and as such should be shunned in favor of the symbols (diamond, box, circle) when possible.

3.3.2 File Menu

The File menu offers access to files on disk, preferences, and control over most of the CLASP windows. All filenames are specified by way of a standard Macintosh file requester.

LOAD DATASET `command-o`

Loads a dataset from a file in CLASP format. The CLASP format presents data as a list of rows, and is described in more detail in section 7.3.2

IMPORT DATASET `command-i`

Loads a dataset from a file in columnar format. The FILE DELIMITER preference of the CLASP preferences window specifies the column separator, and the INCLUDE LABELS preference specifies whether or not the first line of the file is a row of column names.

SAVE DATASET `command-s`

Saves a dataset to a file in CLASP format.

EXPORT DATASET `command-f`

Writes a dataset to a file in columnar format. As with Import Dataset, the FILE DELIMITER preference is the character used to separate columns and INCLUDE LABELS specifies whether or not to write the first line of the file as column names.

CLOSE `command-w`

Closes the frontmost selected window. This function will not close a floating window.

CLEAR HISTORY `command-backspace`

Clears the history view of its contents. The contents of the history buffer cannot be restored, so use this command with caution.

UPDATE DATASET VIEW

Refreshes and updates the dataset view of the CLASP window. This command can be used to view any new datasets or variables resulting from CLASP commands entered from the Lisp listener, since they do not automatically update the dataset view.

LOG SELECTION `command-t`

Copies the current selection in the CLASP history view to the CLASP Notebook.

COPY TO CLIPBOARD `command-c`

Expands and copies the selected items in the currently selected window to the clipboard for pasting into other applications.

COPY RIBBON TO CLIPBOARD `command-b`

Copies the contents of the ribbon window to the clipboard. This command is very useful for copying the lisp code which is generated in the history view of the CLASP window[§]

[§]The commands printed in the history view have been beautified for succinctness. The actual lisp code, which is what is displayed in the ribbon window when the command is clicked on, is often far more complex and unclear. This is why one must cut from the ribbon to get at the actual lisp expressions. See section 3.5 for more info.

PASTE FROM CLIPBOARD `command-v`

Pastes the contents of the system clipboard into the currently selected window, when possible.

PREFERENCES `command-'`

Opens the CLASP Preferences window for editing the preference settings.

QUIT `command-q`

Exits CLASP. Results of a session are not automatically saved by CLASP. It is up to the user to write valuable data before ending a CLASP session.

3.3.3 The Graph Menu

The graph menu offers access to the EKSL plotter routines for generating graphical views of data.

HISTOGRAM *X(s)* `command-h`

Creates a histogram from the variable X.

SCATTER PLOT *X(s) Y(s)* `command-a`

Creates a scatter plot of X against Y.

LINE PLOT *X(s) Y(s)* `command-l`

Creates a line plot of Y on X.

ROW LINE PLOT *Y(s)* `command-n`

Creates a line plot of Y on 1,2,3...N, where N is the data length of variable Y.

REGRESSION PLOT *X(s) Y(s)* `command-g`

Creates a scatter plot of X against Y and overlays the regression line from the linear regression of Y on X.

PLOT TABLE *Table* `command-p`

Creates a graph from the currently selected table window (the table must be viewed to be plotted). Each row of the table is represented as a line plot, with X being the column number and Y being the value at $(row, column)$. Note that this command does not make sense for certain tables, such as the Anova table, which will not produce plots.

OVERLAY GRAPHS *Graph(s)* `command-=`

Produces a single graph from the composition of the graphs selected in either the graph palette or the history view of the CLASP window. The Overlaid Graph dialog box offers a radio button to specify whether to use the graph palette or the history view to generate the overlaid graph.

EDIT `command-e`

Opens a graph editor for the currently selected graph window.

GRAPH KEY `command-k`

Displays the graph key (legend) for the currently selected graph window.

3.3.4 The Describe Menu

The describe menu offers access to the descriptive statistical functions in CLASP. Since each of these functions are discussed in the 5.1 section, only the arguments are considered here.

DATA LENGTH *X(s)*

MEAN *X(s)*

VARIANCE *X(s)*

STANDARD DEVIATION *X(s)*

MINIMUM *X(s)*

MAXIMUM *X(s)*

RANGE *X(s)*

QUANTILE *X(s) (Percentile)*

Percentile is entered into the provided number box and is a real number in the range 0 to 1. This value is mapped over all X arguments.

MEDIAN *X(s)*

TRIMMED MEAN *X(s) (Trimming Factor)*

Trimming Factor is entered into the provided number box and is a real number of the range 0 to 0.5. This value is mapped over all X(s).

MODE *X(s)*

Determines the most commonly occurring value of a discrete distribution. If the distribution is continuous, CLASP estimates the mode by considering all groups of consecutive data with size n, $n = max(3, datalength/10)$, and returning the midpoint of the group with the smallest range. CLASP can be asked to provide its best guess as to whether to use the mode for continuous or discrete data.

INTERQUARTILE RANGE *X(s)*

SKEW *X(s)*

Calculates the skew (third moment) of the distribution.

STATISTICAL SUMMARY *X(s)* `command-m`

Creates a table of descriptive functions for each X in turn. The functions applied are data length, mean, variance, standard deviation, minimum, maximum, range, median, interquartile range, and skew.

COVARIANCE *X(s) Y(s)*

CORRELATION *X(s) Y(s)*

PARTIAL CORRELATION *X(s) Y(s) IV(s)*

Computes the partial correlation for each mapped X,Y pair with the variables specified in IV(s) held constant.

CROSS CORRELATION *X(s) Y(s) (Min lag) (Max lag)*

Min lag and Max lag are entered into the provided number boxes and are integers in the range 0..N, where N is the minimum data length of the mapped pairs X and Y.

AUTOCORRELATION *X(s) (Min lag) (Max lag)*

Min and Max lag are as for cross correlation.

CORRELATION TABLE *X(s)*

Produces a table of all pairwise correlations of X(s).

CONTINGENCY TABLE *X(s) Y(s)*

Produces a contingency (observed frequency) table of X against Y.

3.3.5 The Manipulate Menu

The Manipulate menu has commands for manipulating datasets, tables, and variables and their properties.

RENAME *(Dataset/Variable) (Name)* `command-r`

Changes the name of Dataset to Name. Dataset is selected in the dataset view and Name is entered into a text entry dialog generated by the command.

DELETE RESULTS *(Results)* `command-d`

Removes the Results (variables and/or datasets) from the dataset view. Results are selected in the dataset view before the command is issued.

PARTITION DATASET *(Dataset) (Partition Clause) (Include Variables)*

Partitions Dataset using Partition Clause. The Include Variables may be specified explicitly, or if none are provided, all of the dataset's variables are included. The partition clause should be a regular lisp expression. Use variable names to refer to a dataset's variables and quoted symbols to refer to categorical values. Also note that an appropriate predicate must be used for partitioning categorical variables, such as `'eql`. For example, to create a partition of all rows where a variable named "outcome" is `success`, the partition clause might be `(eql outcome 'success)`.

PARTITION ON *(Dataset) (Partition Variable) (Include Variables)*

Partitions Dataset on each unique value of Partition Variable. If more than 50 partitions will result, you will be queried before CLASP proceeds with the partitioning.¶ Include Variables is as for Partition Dataset.

MERGE DATASETS *(Datasets)*

Creates a new dataset by combining the rows of Datasets.

Tables

The tables menu item contains sub-items for viewing different properties of some tables. There are three sub-items, which correspond to control, option, and double

¶The value 50 is determined by the CLASP variable `*more-partitions-than-you-really-want*`.

clicking the currently selected table respectively. For many tables, these clicks have no effect, and the menu items are disabled. If the menu item does apply, the menu item's text will reflect the type of view it will provide for the currently selected table. For example, if a contingency table is selected, the first menu item will be "Row Proportions" and will convert the table to row proportions.

Variables

The variable menu item contains sub-items for moving, viewing, and renaming variables.

X `command-x`

If the current window is the CLASP window, this menu item copies the variables selected in the CLASP data view to the X dock of the CLASP dock if there is one open, otherwise it sends the selected items to the X slot of the frontmost statistical dialog. If the current window is a CLASP statistical dialog, the X Dock command grabs the items in the X Dock (if there is one) and replaces the X arguments of the dialog with them, or if there is no dock, the X arguments are replaced by the selected items in the dataset view of the CLASP window (if there is one).

Y `command-y`

If the current window is the CLASP window, this menu item copies the variables selected in the CLASP data view to the Y dock of the CLASP dock if there is one open, otherwise it sends the selected items to the Y slot of the frontmost statistical dialog. If the current window is a CLASP statistical dialog, the Y Dock command grabs the items in the Y Dock (if there is one) and replaces the Y arguments of the dialog with them, or if there is no dock, the Y arguments are replaced by the selected items in the dataset view of the CLASP window (if there is one).

X ← <VAR NAME> `command-[`

This function recalls the most recently generated variable result and inserts it into the currently selected statistical dialog (if there is one). If caps lock is active, the most recent variable result will be added to the end of the list of X arguments. If caps lock is not active, the most recently generated variable will replace the existing X arguments. The most recent variable result is stored in the variable `*most-recent-vresult*`.

Y ← <VAR NAME> `command-]`

This function recalls the most recently generated variable result and inserts it into the currently selected statistical dialog (if there is one). If caps lock is active, the most recent variable result will be added to the end of the list of Y arguments. If caps lock is not active, the most recently generated variable will replace the existing Y arguments. The most recent variable result is stored in the variable `*most-recent-vresult*`.

Z ← <VAR NAME> `command-}`

This function recalls the most recently generated variable result and inserts it into the currently selected statistical dialog's third variate slot (if there is one), such as the "fixed" variable in a partial correlation. If caps lock is active, the most recent variable result will be added to the end of the list of Z arguments. If caps lock is not active, the most recently generated variable will replace the existing Z arguments. The most recent variable result is stored in the variable `*most-recent-vresult*`.

CHANGE TYPE **command-j**

Changes the selected variables in the data view of the CLASP window to either categorical or interval designation.

EDIT **command-e**

Opens a window for viewing and editing the selected variables or datasets[‖] in the CLASP data view. Double-clicking an item while viewing a variable or dataset will request a new value for that item.

3.3.6 The Transform Menu

The Transform menu contains commands that transform data to produce new variables, which are named according to the transformation applied.

USER DEFINED *(Dataset) (Expression)*

Evaluates Expression on each row of Dataset to produce a new variable. Expression should refer to variables in dataset by name and should be a regular lisp expression. For example, to calculate processing time per node of a search, Expression might be `(/ nodesvisited totaltime)`.

SORT *X(s)*

Sorts each X in ascending order.

INDEX *X(s) Index(s)*

Sorts each X by its mapped pair Index. Both must be CLASP variables from the same dataset.

ADD A CONSTANT *X(s) (Constant)*

Adds Constant to each data item of X(s). Constant may be any real number, specified in the provided number entry box.

MULTIPLY BY CONSTANT *X(s) (Constant)*

Multiplies each data item of X(s) by Constant. Constant may be any real number and is specified in the provided number entry box.

NATURAL LOG *X(s)*

Takes the natural log of each data item in X(s).

LOG10 *X(s)*

Takes the common log (base 10) of each data item in X(s).

RECODE CATEGORICAL *X (Old Values) (New Values)*

Substitutes each member of New Values for the corresponding value in Old Values. This can be used to create an integer coding for a categorical variable (or vice versa). New Values are specified by selecting the old values in the list provided, entering the new value in the text box, and clicking SET or pressing return. The result type may

[‖] Viewing datasets is generally slower, but preferred for viewing data across rows.

be specified as Number or Symbol via the radio buttons, and the final transformation applied by clicking RECODE. Any selection in the list of unique values may be changed via the arrow keys.

REASSIGN BINS *X (Bin Limits) (Bin Names)*

Converts a continuous variable into a discrete variable. Bin Limits and Bin Names are computed from supplied data in the Recode dialog. The Start and End bins are supplied via provided numerical boxes. The third required argument is either the number of bins in the new range or the bin size.

SMOOTH *X(s) (Type) (Window)*

Applies a single smoothing operator to each X. The smoothing operation is specified as either a mean smooth or a median smooth with a window size as an integer $2 \geq window \leq 5$.

SMOOTH 4253H *X(s)*

Applies a series of median smoothing operators to X(s). The series is *median-4, median-2, median-5, median-3, hanning.*

DISCRETE DERIVATIVE *X(s)*

Takes the pairwise difference of successive values of X.

STANDARDIZE *X(s)*

Standardizes each variable X according to the transformation $x_i' = \frac{(x_i - mean)}{std.dev}$

3.3.7 The Test Menu

The commands in the Test menu are described in detail in the 5.1 chapter, and so only the arguments are discussed here.

CONFIDENCE INTERVAL USING Z STATISTIC *X(s) (Confidence level)*

Confidence level is a real number between 0 and 1, is specified in the provided number entry box, and is not a mapping argument.**

CONFIDENCE INTERVAL USING T STATISTIC *X(s) (Confidence level)*

Confidence level is a real number between 0 and 1, is specified in the provided number entry box, and is not a mapping argument.

CONFIDENCE INTERVAL OF A PROPORTION *(Successes) (Trials) (Confidence level)*

Success is an integer ≥ 0, Trials is an integer ≥ 1, Success $<$ Trials, and Confidence level is a real number in the range 0 to 1.

T-TEST ONE SAMPLE *X(s) (Mean) (Tails)*

Mean is the mean to test against, and is not a mapping argument. Tails designates whether the rejection region is in the poitive or negative tail of the t distribution; BOTH indicates a two-tailed test.

**That is, its value is set at compute time and is constanct across all pairings when the "compute" button is pushed.

T-TEST TWO SAMPLES *X(s) Y(s) (Tails)*

X and Y are mapped via the standard bivariate CLASP mapping. Tails designates whether the rejection region is in the poitive or negative tail of the t distribution; BOTH indicates a two-tailed test.

T-TEST MATCHED PAIRS *X(s) Y(s) (Tails)*

X and Y are mapped via the standard bivariate CLASP mapping. Tails designates whether the rejection region is in the poitive or negative tail of the t distribution; BOTH indicates a two-tailed test.

D TEST *X(s) Y(s) (Resamples) (Tails)*

X and Y are mapped via the standard bivariate CLASP mapping. Resamples is the number of pseudosamples on which to run the test, and is an integer > 0. Tails designates whether the rejection region is in the poitive or negative tail of the t distribution; BOTH indicates a two-tailed test.

ANOVA - ONE WAY *X(s) Y(s)*

X and Y are mapped via the standard bivariate CLASP mapping. Displays an anova table, a list of the group means, and a Scheffe table, if specified in the Scheffe check box. If a confidence is specified and confidence intervals are enabled, the group means table and any graphs based on it will display confidence intervals.

ANOVA - TWO WAY *Y(s) X1(s) X2(s)*

Y, X1, and X2 are mapped via lexical mapping. Displays an anova table and a table of the group means.

CHI SQUARE COUNTS *A B C D*

Performs a chi square test on the contingency table $\begin{matrix} A & B \\ C & D \end{matrix}$.

CHI SQUARE 2X2 *X(s) Y(s)*

X and Y are two-valued (binary) variables and are mapped via the standard bivariate CLASP mapping. Displays the chi square statistic, G test results, and the resulting contigency table.

CHI SQUARE RXC *X(s) Y(s)*

X and Y are mapped via the standard bivariate CLASP mapping. Displays the chi square statistic, G test results, and the resulting contigency table.

LINEAR REGRESSION BRIEF *X(s) Y(s)*

X and Y are mapped via the standard bivariate CLASP mapping. Displays the slope, intercept, r squared statistic, standard error of the slope, and the significance.

LINEAR REGRESSION VERBOSE *X(s) Y(s)*

X and Y are mapped via the standard CLASP mapping. Displays the information Linear Regression Brief does, as well as an anova table and a correlation table.

MULTIPLE LINEAR REGRESSION *X(s) (Y)*

Performs a multiple linear regression of Y on X's, displaying the correlation and anova tables, the F statistic, coefficients, betas, and t statistic for each X.

3.3.8 The Sample Menu

The Sample menu contains commands which produce samples from various theoretical distributions. The variables in the dataset will be sampled from the specified distribution based on supplied parameters. All sampling commands sample from a theoretical distribution require the arguments Number of Samples, an integer indicating how many variables (samples) to create, and Sample Size, an integer indicating how many elements each sample should contain.

UNIFORM DISTRIBUTION *(Minimum) (Maximum)*

Produces samples which are uniformly distributed between Minimum and Maximum, which are both real numbers.

NORMAL DISTRIBUTION *(Mean) (Std Deviation)*

Produces samples with mean of Mean and standard deviation of Std Deviation, which are both real numbers.

BINOMIAL DISTRIBUTION *(P) (N)*

P is a real number from 0 to 1 indicating the probability of success. N is an integer ≥ 1, which is the number of Bernoulli trials per sample.

POISSON DISTRIBUTION *(Mean)*

Mean is a real number.

GAMMA DISTRIBUTION *(Integer Order)*

Integer Order is an integer ≥ 1.

MONTE CARLO *(Function-1) [Function-2] (Statistic) (Resamples)*

The Monte Carlo dialog provides an interface for testing hypotheses on known distributions. The mechanism calculates a statistic on a sample of a known distribution *resamples* times to produce a new dataset representing the results. The statistic may be univariate or bivariate, and is named in the provided box labelled STATISTIC. A menu provides shortcuts to some common statistics. Likewise, a lisp form must be typed in for the X (and Y for bivariate) argument to the statistic. The result of the lisp form must be a sequence which can be passed to the *statistic* function. A pull-down menu also provides templates for this function which may be filled in with actual numeric aruments. The name of the resulting dataset may be indicated in a provided text box.

BOOTSTRAP/RANDOMIZE *(Variable-1) [Variable-2] (Sampling-type) (Statistic) (Resamples)*

Randomization and bootstrap methods extend the Monte Carlo procedure by allowing the samples to come from a known variable instead of a functionally defined distribution. The variable(s) are supplied in drop slots and the sampling type is specified by selecting the appropriate radio button. The radio buttons in the left column draw samples exclusively from one variable, and are suitable for univariate or bivariate statistics. Options in the right column draw samples considering X and Y jointly, and are only suitable for bivariate operations.

MIXED SAMPLING *(Sample1) [Sample2] (Bootstrap Function) (Resamples)*

The Mixed Sampling dialog mixes the Monte Carlo dialog with the Randomize/Bootstrap dialog to provide a hybrid approach to hypothesis testing should it become desirable to mix the two approaches in a bivariate test.

3.3.9 The Window Menu

The Window menu contains items for opening special CLASP windows and a list of open windows. To activate an open window, choose its name from the Windows menu.

CLASP WINDOW `command-1`

Opens the main CLASP window.

CLASP DOCK `command-2`

Opens a CLASP variable dock.

CLASP NOTEBOOK `command-3`

Opens the CLASP Notebook.

GRAPH PALETTE `command-4`

Opens a CLASP graph palette.

TABLE PALETTE `command-5`

Opens a CLASP table palette.

3.4 Supplying Arguments

Supplying arguments to a CLASP command is the last step before viewing the results. CLASP'S Mac interface generates dialog boxes with appropriate mechanisms for collecting the required arguments to run a command. There are several common graphical items for gathering arguments.

Pop Up Menus are used for specifying a single argument from a list of possibilities. The TAILS argument to a t-test is specified with a pop up menu. For fixed pop up menus, the currently selected pop-up menu item at the time the compute button is clicked will be used in the command.

Variable Pop Up Menus are used when multiple arguments apply. Commands which offer mapping, such as the t-tests, make use of variable pop up menus for gathering X and Y variables. Arguments may be added to a variable pop up menu by drag and drop, automatically from the dock, by the X and Y Dock menu items, or from the dock by special clicks (see the section on CLASP Layout). Items may be removed from a variable pop up menu by selecting the items one at a time in the menu and clicking the DELETE button directly to the pop up menu's right.

Drop Slots appear as printed text in the statistical dialogs. An empty drop slot appears as the text <NONE>. These items are used as in the case of pop up menus, but in situations where only one arguments is needed. These items are used for procuring graph arguments. Items may be supplied to a drop slot as with pop up menus, except that only the last item supplied is considered, and items may not be deleted from drop slots.

Text and Number Boxes are used for gathering arguments which occur as numbers and text. CLASP uses these items to gather information that cannot be specified any other way. These items operate as standard Macintosh text entry items except that the number items will not accept non-numeric input.

Check Boxes are used to specify boolean arguments. If there is an 'X' in the box, it is indicated that that option should be used. The SCHEFFE TESTS item in the Anova One Way box is an example of this type.

3.5 Understanding CLASP Output

During a CLASP session, the history view will be continuously updated with output from the various operations as they occur. Each action can add one or more results to the history view. Understanding the types and meanings of the output provided is crucial to the value of CLASP. Each of the history view's content types, with a sample of each, are discussed below.

command `(CORRELATION (X-ARGS (VARIABLE 'X)) (Y-ARGS (VARIABLE 'Y)))`

Commands are informative representations of the command issued. Note that these are not evaluatable instances of lisp code. Many CLASP commands require preprocessing[††] and are uninformative as is in the history view. Selecting a command in the history buffer will display the actual lisp expression in the ribbon window, if that window is active. Once in the ribbon window, the command may be cut to the clipboard and then pasted into the listener for editing and/or evaluation. For more information on generating lisp code, see the items listed below and consult the documentation on CLASP's functional interface. Double-clicking ona command item re-executes that command.

simple result `Mean of X is 0.475`

Simple results are displayed as one line of text.

dataset `#.(DATASET 'UNIFORM-DISTRIBUTION)`

This is a regular dataset result. It will appear in the dataset view and may be manipulated as a regular dataset. These objects, when appearing in lisp code cut from the ribbon, will be roughly equivalent to the form `(CLASP::GET-DATASET <name>)`, and thus can be used directly in lisp code. For example, `(CLASP::STANDARDIZE #.(VARIABLE X))`.

variable `#.(VARIABLE 'STANDARDIZED-X)`

[††]such as with clasp::simplify

This is a regular variable result. It will appear in the dataset view and may be manipulated as a regular variable. These objects, when appearing in lisp code cut from the ribbon, will be roughly equivalent to the form `(CLASP::GET-variable <name>)`, and thus can be used directly in lisp code.

statistical table `#.(TABLE 'STATISTICAL-TABLE 'T Test of X and Y)`

Statistical tables are columnar formatted bodies of information produced as results of analyses. Some commands produce multiple tables. All tables produced are echoed into the table palette, and may be opened by double-clicking them. These objects may not be manipulated manually, and attempts to execute lisp code refering to tables by their printed name will fail.

composite table `#.(TABLE 'COMPOSITE-TABLE 'Results for anova of X against Y)`

Composite tables compose multiple tables produced by a function into one unified table. Like statistical tables, they may be viewed by double clicking and are echoed into the table palette. These objects may not be manipulated manually, and attempts to execute lisp code refering to composite tables by their printed name will fail.

graph `#.(GRAPH 'ROW-LINE-PLOT 'Row-Line-Plot of X[Dataset-A])`

Graphs are visual representations of data. The type of the graph is the first part of the line of text in the history buffer. These items are mirrored to the graph palette. Control-double-clicking these items opens a preference dialog for the graph, and double-clicking them opens them for viewing. These objects may not be manipulated manually, and attempts to execute lisp code refering to graphs by their printed name will fail.

3.6 Example Usage

The following command sequence illustrates typical usage of CLASP, starting with the creation of a dataset and touching on usage of CLASP analyses.

- In the sample menu, choose uniform distribution. Click on compute, using the default values except changing number of samples to 2. This creates a new dataset with two variables named uniform-distribution and uniform-distribution-1 containing uniform distributions of 100 elements each.
- Press command-1 and then command-2 to open the main CLASP window and a variable dock.
- In the CLASP window, control-click uniform-distribution. It will appear in the X Dock. Click uniform-distribution-1 and press command-y. It will appear in the Y Dock. You are now ready to perform analyses on these two variables.
- Choose T-Test Two Samples from the test menu. Note that X and Y arguments will automatically be in place. Click compute. If a table does not automatically pop open (because the auto open tables preference is not set), double click the resulting `(STATISTICAL-TABLE ... )` item in the history view.
- Now choose the Preferences item from the file menu and make sure that BYPASS DIALOGS is not checked. Click USE.

- Choose Smooth 4253h from the transform menu. Note that uniform-distribution is automatically assumed as the X variable. Select the Smooth 4253h dialog and then control-click uniform-distribution-1 from the Y Dock. It will replace the existing X argument in the Smooth 4253h dialog. Click COMPUTE.

- Control-click the new variable smooth-of-uniform-distribution-1 in the history view. It will appear in the X Dock. Now select it in the X dock (do not multi-select uniform-distribution). Choose Row-Line Plot from the graph menu. Note that only the selected X Dock variable is assumed to be an argument to the graph. Click GRAPH. If a graph does not open automatically (because the auto open graphs preference is not set), double-click the resulting `(ROW-LINE-PLOT ... )` item in the history view.

3.7 Preparing CLASP Reports

Macintosh CLASP provides two methods of output for preparing session results for presentation. These "logging" methods are activated by choosing the appropriate logging type in the preference window, then sending the desired results to the CLASP notebook by option-clicking them in the CLASP history view, and then clicking the LOG button in the notebook window. In addition, CLASP also supports the use of the Mac clipboard for cutting results from CLASP and pasting into any other system compliant document preparation package.

Cut and Paste Using the Cut Selection menu item from the file menu, the selection range in the current window can be copied to the clipboard and pasted into another application. For instance, a range of CLASP commands and tables could be selected and then copied via the CUT SELECTION item (or by command-t, its keyboard shortcut). A document preparation application could then be selected and those results pasted into a document. Due to limitations of the Macintosh clipboard, text and graphs may not be mixed, nor can multiple graphs reside in the clipboard simultaneously.[‡‡]

Event Logging Event logging takes advantage of AppleEvents, a interprocess scripting system. Through event logging, the cut and paste process can be automated and directed to any application which also supports the AppleEvents system and is accessible to the machine running CLASP. After preparing the notebook with the desired results and choosing EVENT LOGGING in the preferences window, clicking the LOG button will produce a dialog asking you to choose a destination application. This window is much like the chooser window. Once you have selected an application, CLASP will attempt to generate chunks of results, wake up the selected application, and paste the results into it.

It is important to take care that several conditions are met before trying to log results to an application. First, the destination application must be AppleEvents aware. There is no way to determine this from CLASP, and so you should consult the documentation of any packages in question. Second, the application must be prepared to accept a paste command. For most applications, this requires that there is an open document window and an active paste menu item. Failure to meet these conditions will result in a lisp condition, sending the Listener to the front of the screen.

[‡‡] This is not actually a limitation of the Mac clipboard, but rather a limitation of most programs which paste from it.

3.8 Clasp Errors

With Macintosh CLASP, there are two levels on which errors can be generated. One is at the interface level, and the other is at the lisp environment level.

Interface errors are generated when the interface detects improper use of the CLASP system, or an error is signalled during the process of executing an interface command. Errors of this type include not supplying enough arguments to an analysis, trying to plot a table which does not have a clear graphical representation (an anova table), and attempting to find the mean of a categorical variable. This type of error is handled by presenting a dialog box explaining the error and stopping further execution.

Lisp environment errors are errors which are not trappable by the interface. Such errors are signalled when using the functional interface to CLASP. Since this type of error occurs below the GUI level, it is not handled with a dialog box. Internal errors report a message to the Lisp listener, and increment the listener error level. Pressing command-. will abort the error and decrement the listener error level.

Chapter 4

CLIM User Interface

This chapter describes the layout and use of the CLIM CLASP graphical user interface. The interface is built using the Common Lisp Interface Manager (CLIM), so many CLIM features are available in CLASP.

The first section describes the layout of CLASP. The second section discusses CLASP command syntax. The next section explains how to interact with CLASP using the mouse and keyboard. Some examples of CLASP usage follow. Finally, each of the command menus are described.

4.1 CLASP Layout

The main window of CLASP is called the CLASP *frame*. This frame contains a number of sections called *panes*. Information in CLASP is displayed in one of four areas.

Menu Bar The CLASP menus will appear across the top of the CLASP frame. The menus are: File, Graph, Describe, Manipulate, Transform, Test and Sample. Clicking on a menu name will pop up the menu.

Datasets Display This contains the names of all Datasets and Variables currently loaded into CLASP. Dataset names are aligned to the left margin; variable names for each dataset are indented below the dataset name.

Results Display A mouse-sensitive icon for each CLASP result appears in this window.

Notebook CLASP commands and lisp expressions are entered here.

While the `Menu Bar` and `Notebook` are always in the CLASP frame, the `Datasets Display` and `Results Display` are under user control. They can either be panes at the left edge of the CLASP frame, or in separate windows. This is controlled through the :PREFERENCES command.

4.2 CLASP Command Syntax

4.2.1 Basic syntax

Commands have the form :VERB *arg1 arg2 ... argn*. :VERB is the name of the command and *arg1 arg2 ... argn* are the arguments it accepts. Note that when being entered, commands are prefixed by a colon character (:).

Some examples of command usage:

:Load Dataset (Pathname) **my-data.clasp**

:Rename Variable (Variable) **Var-1** (New name) **Numnodes**

:Merge Datasets (Datasets) **Dataset-1, Dataset-2**

4.2.2 Mapped arguments

Many CLASP commands give the user the option of entering multiple values for their arguments. When this happens, the command will be executed for each value of the argument it is given. This is called argument mapping. If a command has more than one argument that allows mapping, each argument must be given the same number of values and multiple values will be mapped over in parallel. That is, the command will be executed on all the first values of its arguments, then all the second values, etc.

Some examples of mapped arguments are:

:Mean (X(s)) **Var-1, Var-2, Var-3**
Will return the mean of `Var-1`, `Var-2` and `Var-3`.

:Anova One Way Variables (Y(s)) **Y1, Y2** (X(s)) **X1, X2**
This will do a one-way anova of `Y1` on `X1` and a one-way anova of `Y2` on `X2`.

:Correlation (Y(s)) **Y1, Y2** (X(s)) **X1**
This is illegal, `X` and `Y` are both mapping arguments, but they were given different numbers of inputs.

The following example shows an instance where not all the arguments of a command are mapped arguments. In :T TEST TWO SAMPLES, although the X and Y arguments map, the *Tails* argument doesn't. Whatever single value is entered for *Tails* will be used for every iteration of the other arguments. The prompt for an argument that may be mapped over will end in *(s)*.

:T Test Two Samples (Y(s)) **Y1, Y2** (X(s)) **X1, X2** (Tails: [Both, Positive or Negative]) **Positive**
This will perform two 2 sample t-tests, one on `X1` and `Y1` and another on `X2` and `Y2`. Both tests will be `Positive` tail tests.

4.3 Interacting with CLASP

At the command prompt ($\Rightarrow$), CLASP will accept either a CLASP command or a lisp expression. If a command is entered, its arguments will be prompted for and then the command will be run. If a lisp expression is typed, it will be evaluated and the value printed as in a normal lisp listener. When CLASP is looking for a particular kind of input, a variable, a string, etc. it is said to be *accepting* that kind of input. When accepting a particular type of input, CLASP makes it easy to enter that type of data by making objects of that type mouse selectable and also by doing completion on the names of those objects. For instance, when CLASP is accepting a variable as an argument to a statistical command, all CLASP variables will be selectable. In addition any sequence of numbers being displayed in the CLASP frame will be selectable, as will any lisp symbol which evaluates to a sequence of numbers.

4.3.1 Using the mouse

To enter a command, click on the menu in which the command resides and then click on the command. When a command has been selected and is waiting for arguments to be entered, any object which is a valid argument will become mouse selectable.

To select multiple objects, either for a mapped argument or a command which accepts multiple objects (such as :MERGE DATASETS), click on each object.

At the command prompt, clicking on a previously entered command or lisp expression will re-execute it. Clicking on a CLASP object (a Dataset, Variable or Result) will cause it to :OPEN, clicking on an open object will cause it to :CLOSE.

While entering a lisp expression, clicking on any other lisp expression will insert it at the cursor. Clicking on a CLASP variable will insert its value into the current expression as a quoted list.

4.3.2 Using the keyboard

As with the mouse, whenever CLASP is accepting a variable, the name of any CLASP variable may be typed. In addition, the name of any lisp symbol which has a sequence of numbers as its value may be typed. To enter multiple objects, separate them by commas.

CLASP also performs completion on typed input. When entering a command or the name of a variable, the Tab key will do partial completion, filling in as much information as possible until there are two possibilities which will be distinguished by the next character. For instance, since there are a number of sampling commands, all of which start with :SAMPLE, typing :SAM followed by a `Tab` will complete the word :Sample. The Space key completes the current word or object, and then prompts for the next piece of input (either the next word of a command, or the next argument.) The Return key indicates to CLASP that the input is complete and ready for processing. Returning to the :SAMPLE commands, if the user types :SAMPLE U (which uniquely determines the command :SAMPLE UNIFORM), then the three completion characters will behave differently. Space will complete the command and prompt the user for the first argument. Tab will complete the command, but it will not prompt for the first argument. Return will complete not only the command, but the first few arguments, which have default values, and then it will try and process the input line. In general, a good rule of thumb is: "When in doubt, hit the Space key, if that doesn't do what you want, hit the Backspace key and go from there."

Since CLASP uses the CLIM input editing facility, most of the normal CLIM editing keys will work. The following is a table of editing keys available at the CLASP command line.

When typing a lisp expression in CLASP, the bang key (`!`) can be used to reference CLASP variables. The expression `!variable-name` will substitute the value of the variable into the expression. If more than one variable has the same name,

```
!variable-name!dataset-name
```

will uniquely determine a variable, and its value will be used. If there are multiple variables with the same name, and the dataset is not specified, the matching variable belonging to the *most recently created dataset* will be used. When a final close parenthesis is entered in a lisp expression, the expression will immediately be evaluated – there is no need to hit Return.

A useful trick for editing a lisp expression from a previous line is:

Forward character	control-F
Backward character	control-B
Beginning of line	control-A
End of line	control-E
Delete next character	control-D
Delete previous character	Rubout
Kill to end of line	control-K
Transpose adjacent characters	control-T
Yank from kill ring	control-Y

Table 4.1: Keys used for editing input

1. Type an open parenthesis - This will tell CLASP that you are entering an expression.
2. Click on the line to edit - This will insert it into the current line.
3. Edit the line- Make whatever changes are desired using the keyboard editing characters described below.
4. Remove initial parenthesis - Go to the beginning of the line (you can use `Control-A`) and delete the extra open parenthesis (`Control-D`.)
5. Move to the end of the line (`Control-E`) - This will cause the new line to be entered.

4.4 Examples of CLASP command usage

Some examples of other CLASP commands follow. These commands make a new dataset, define a new variable for the dataset and rename the new variable.

> :Make Dataset from Rows (Name) **Exper-1** (Data) **A, 13, 431; A, 10, 389; B, 11, 214; B, 10, 203** (Variable names) **Method, Time, Score**
> Create a new dataset called `EXPER-1` with three variables, `METHOD`, `TIME`, and `SCORE`, and 4 rows of data. Note that in entering data, single items are separated by a comma, rows are separated by a semicolon.
>
> :User Defined (Dataset) **exper-1** (Expression) **(/ score time)**
> :USER DEFINED is a command from the `Transform` menu which allows arbitrary transformations of variables. In this case, the command will add a new variable to `EXPER-1` which divides `SCORE` by `TIME`.
>
> :Rename Variable (Variable) **/-score-time** (New name) **points-per-second**
> Commands from the transform menu compute a name for the new variable by using the expression that created it. Often the user will want to give the new variable a more semantically appropriate name. This command rename the newly created variable, `/-SCORE-TIME` to `POINTS-PER-SECOND`.

4.5 Clasp Command Menus

The Clasp command menu pane has seven menus, each of which is described in this section. Clicking on a menu name will pop up the menu, left clicking on a menu item will select that item, right clicking on it will bring up help for that item.

4.5.1 The File Menu

File
Load Dataset
Import Dataset
Save Dataset
Export Dataset
Clear All
Clear Notebook
Save Notebook
Print
Preferences
Quit

File menu commands are used to manage datasets and the Clasp notebook, and to print results.

:Load Dataset *(Filename)* [*Command*]

Loads a dataset from a disk file in Clasp format.*

:Import Dataset *(Filename) (Separator) (Include variable names)* [*Command*]

Loads a dataset from a disk file in columnar format. *Separator* is the character used to separate columns in the file. If *Include variable names* is `Yes`, then the first line of the file will be interpreted as variable names.†

:Save Dataset *(Filename)* [*Command*]

Saves a dataset to a disk file in Clasp format.

:Export Dataset *(Dataset) (Filename) (Separator) (Include variable names)* [*Command*]

Saves a dataset to a disk file in columnar format. As with :Import Dataset, *Separator* is the character used to separate columns. If *Include variable names* is `Yes`, then the variable names will be written to the first line of the file.

:Clear All [*Command*]

Clears the Clasp notebook and all datasets and results.

:Clear Notebook [*Command*]

Clears the Clasp notebook, but doesn't delete datasets and results.

:Save Notebook [*Command*]

Saves the current notebook to a postscript file.

:Print *(Object) (Filename)* [*Command*]

*Clasp file format is described in 7.3.2.

†For more information on importing and exporting files, see 7.3.2.

Object may be a CLASP dataset or the result of a CLASPcommand. The output is written in postscript format to *Filename*. Because of limitations in Clim 1.1's postscript support, printing graphs is difficult. The following procedure will produce reasonable postscript output for graphs.

1) Unless the graph will be printed with a color postscript printer, set the color of each dataset in the graph to black. This can be accomplished by clicking on the datasets in the graphs legends and using the window that pops up to set the dataset color. (Note, since SCIGRAPHdisplays graphs with a black background, the datasets will not be visible on screen.)

2) Print the graph to a file using the :PRINT command.

3) Run the postscript file through the sed script `ps-fix.sed`, to do this type
`sed -f ps-fix.sed file.ps > new-file.ps`
at the shell prompt. In the above example, `file.ps` is the name of the file that was created by the :PRINT command and `new-file.ps` is the name of the file created by sed. The file `ps-fix.sed` is available via anonymous ftp from the same location as CLASP.

:PREFERENCES [*Command*]

Brings up a window in which the user can modify the behavior of CLASP. The window has three sections, the results display preferences, the error handling preferences and the datasets and results display preferences.

In the first section of the preferences window, the user can specify how a number of result types are displayed in CLASP. Each type will be followed by four options, `Display-In-Interactor`, `Display-In-Window`, `Iconify` and `NIL`. Clicking on an option will select it. If `Display-In-Interactor` is selected, then when results of that type are created, they will be displayed directly in the notebook. If `Display-In-Window` is selected, then results of that type will initially be displayed in a separate window and an icon will be placed in the notebook. If `Iconify` is selected, then an icon will be placed in the notebook and the result will not be displayed until the icon is opened. If `NIL` is selected, then nothing will be shown in the notebook. In all cases, the results window will always show all existing results. The default setting for all result types is `Display-In-Interactor`.

The available error handling preferences are `Use-Native-Debugger` and `Trap-Underflow-Errors`. When `Use-Native-Debugger` is not selected, CLASP captures lisp errors and prints a message in the notebook. When it is selected CLASP passes lisp errors down to the underlying lisp process. This can be useful when debugging code. When `Trap-Underflow-Errors` is not selected, CLASP will always convert floating point underflow errors to 0.0. Setting this parameter will cause CLASP to signal an error when a floating point underflow occurs. The default value for both error handling preferences is unselected.

The `Display datasets and results` preferences controls where the datasets and results lists are displayed. The `Pane` setting will display these lists directly in the CLASP frame, to the left of the notebook. The `Window` setting will cause them to be displayed in separate windows. This could be useful if when there are many datasets or variables with long names. Most window managers allow resizing of windows, so with this option, the user may set the datasets and results listings to a convenient size. The final setting is `None` which will suppress displaying these lists altogether.

:QUIT [*Command*]

Exits from CLASP. It is up to the user to save whatever datasets were created or changed during the session. CLASP will not save these automatically, nor does it prompt the to save datasets before quitting.

4.5.2 The Graph Menu

Graph
Histogram
Scatter Plot
Line Plot
Row Line Plot
Regression Plot
Overlay Graphs

The Graph menu has commands which allow the creation of graphical views of data.

Graphs in CLASP are created by the SCIGRAPH system from BBN. The elements of these graphs are mouse sensitive. To change the style or color of a line or point, select the entry for that data from the legend. This will pop up a window which allows for editing of data attributes. For a more detailed description of SCIGRAPH and its functionality, see the SCIGRAPH documentation, available via anonymous ftp from cambridge.apple.com in the directory /pub/clim/clim-1-and-2/scigraph.

:HISTOGRAM *(X(s)) (Color by)* [*Command*]

Creates a histogram from the variable X. Unless *Color by* is `None`, the points in X are grouped by the *Color by* variable and each group is shown in a different color. To use the *Color by* option, both X and *Color by* must be CLASP variables and they must come from the same dataset. *Color by* is not a mapped variable, if multiple variables are selected for X, the same *Color by* option will be used for all of them.

:SCATTER PLOT *(Y(s)) (X(s)) (Color by)* [*Command*]

Creates a scatter plot of Y on X. Coloring is the same as for :HISTOGRAM, except that since there are two variable arguments, they must both be from the same dataset as *Color by* for coloring to be used.

:LINE PLOT *(Y(s)) (X(s)) (Color by)* [*Command*]

Creates a line plot of Y on X. Coloring is the same as for :SCATTER PLOT.

:ROW LINE PLOT *(Y(s)) (Color by)* [*Command*]

Creates a line of Y against `1, 2, 3 ... N`, where N is the length of Y. Coloring is the same as for :HISTOGRAM.

:REGRESSION PLOT *(Y(s)) (X(s))* [*Command*]

Creates a scatter plot of Y on X and overlays the regression line from the linear regression of Y on X.

:OVERLAY GRAPHS *(Graphs)* [*Command*]

Graphs is a sequence of graphs. :OVERLAY GRAPHS produces a single graph with each member of *Graphs* overlayed. :OVERLAY GRAPHS will attempt to choose high contrast colors so that points from different source graphs may be distinguished.

To produce colored graphs when the original data are not variables which are in the same dataset as the coloring variable, partition the data by hand, produce a single graph for each partition, and overlay the graphs.

4.5.3 The Describe Menu

Describe
Data Length
Mean
Variance
Standard Deviation
Minimum
Maximum
Range
Quantile
Median
Trimmed Mean
Mode
Interquartile Range
Statistical Summary
Covariance
Correlation
Cross Correlation
Autocorrelation

The Describe menu contains commands which produce descriptive statistics on data. Since most of these commands are described in section 5.1, only their arguments are given here.

:DATA LENGTH *(X(s))* [*Command*]

:MEAN *(X(s))* [*Command*]

:VARIANCE *(X(s))* [*Command*]

:STANDARD DEVIATION *(X(s))* [*Command*]

:MINIMUM *(X(s))* [*Command*]

:MAXIMUM *(X(s))* [*Command*]

:RANGE *(X(s))* [*Command*]

:QUANTILE *(X(s)) (Percentile)* [*Command*]

Note that *Percentile* isn't a mapped argument. It must be a real number from 0 to 1 inclusive, and the same value will be mapped over all *X's*.

:MEDIAN *(X(s))* [*Command*]

:TRIMMED MEAN *(X(s)) (Trimming factor)* [*Command*]

Note that *Trimming factor* isn't a mapped argument. It must be a real number from 0 to .5, and the same value will be mapped over all *X's*.

:MODE *(X(s))* [*Command*]

:INTERQUARTILE RANGE *(X(s))* [*Command*]

:STATISTICAL SUMMARY *(X(s))* [*Command*]

Displays a variety of commonly used descriptive statistics.

:COVARIANCE *(Y(s) (X(s))* [*Command*]

:CORRELATION *(Y(s) (X(s))* [*Command*]

:CROSS CORRELATION *(Y(s) (X(s)) (Min lag) (Max lag)* [*Command*]

Note that *Min lag* and *Max lag* are not mapped arguments.

:AUTOCORRELATION *(Y(s) (X(s)) (Min lag) (Max lag)* [*Command*]

Note that *Min lag* and *Max lag* are not mapped arguments.

4.5.4 The Manipulate Menu

Manipulate
Rename Dataset Rename Variable Delete Result Make Dataset From Rows Make Dataset From Columns Add Variable To Dataset Partition Dataset Partition On Merge Datasets Open Close Describe

The Manipulate menu has commands for manipulating data within a dataset and deleting, renaming, opening, closing and describing data objects.

:RENAME DATASET *(Dataset) (New name)* [*Command*]

Changes the name of *Dataset* to *New name*.

:RENAME VARIABLE *(Variable) (New name)* [*Command*]

Changes the name of *Variable* to *New name*.

:DELETE RESULT *(Object)* [*Command*]

Deletes *Object*. *Object* can be a variable, a dataset or a result from a statistical command.

:MAKE DATASET FROM ROWS *(Name) (Data) (Variable names)* [*Command*]

Creates a new dataset called *Name*, using *Data* as the row-major data and *Variable names* as the names of the columns. To enter data, each row is a comma separated sequence, and semicolons separate the rows. For instance, to create a dataset called DS1, with variables V1, V2, V3 and V4, and containing the data:

2 4 2 3
5 7 4 3

Enter `DS1` for *Name*, `2, 4, 2, 3; 5, 7, 4, 3` for *Data*, and `V1, V2, V3, V4` for *Variable names*.

:MAKE DATASET FROM COLUMNS *(Name) (Data) (Variable names)* [*Command*]

Creates a new dataset called *Name*, using *Data* as the column-major data and *Variable names* as the names of the columns. To enter data, each row is a comma separated sequence, and semicolons separate the rows. For instance, to create a dataset called DS1, with variables V1, V2, V3 and V4, and containing the data:

2 4 2 3
5 7 4 3

Enter `DS1` for *Name*, `2, 5; 4, 7; 2, 4; 3, 3` for *Data*, and `V1, V2, V3, V4` for *Variable names*.

:ADD VARIABLE TO DATASET *(Dataset) (Data) (Name)* [*Command*]

Adds a new variable, called *Name* to *Dataset* using *Data*, which should be a comma separated sequence.

:PARTITION DATASET *(Dataset) (Partition clause) (Include variables)* [*Command*]

Partitions *Dataset* using *Partition clause*. *Include variables* should be a comma separated list of variables from the original dataset to include in the new dataset(s). The default is to include all variables. Partitioning is described in greater detail in section 6.

:PARTITION ON *(Dataset) (Partition variable) (Include variables)* [*Command*]

Partitions *Dataset* on *Partition variable*. *Include variables* is the same as for :PARTITION DATASET.

:MERGE DATASETS *(Datasets)* [*Command*]

Creates a new dataset by combining the rows of *Datasets*. Merging is described in greater detail in section 6.

:OPEN *(Object)* [*Command*]

Displays *Object* in a separate window. *Object* can be a dataset, a variable or a result.

:CLOSE *(Object)* [*Command*]

Closes the window that *Object* is being displayed in. *Object* can be a dataset, a variable or a result.

:DESCRIBE *(Object)* [*Command*]

Prints a history of *Object*'s creation. *Object* can be a dataset, a variable or a result.

4.5.5 The Transform Menu

Transform
User Defined
Sort
Index
Add a Constant
Natural Log
Log10
Recode Categorical
Reassign Bins
Smooth 4235h
Discrete Derivative

The Transform menu contains commands that allow for various transformations of data. Unlike most other commands, the Transform commands require that their arguments be actual CLASP variables. All the menu options create a new variable in the dataset to which the variable being transformed belongs. The name of the new variable will be derived from the transformation being performed, for instance, smoothing a variable called `A` will result in a variable called `SMOOTH-OF-A`.

:USER DEFINED *(Dataset) (Expression)* [*Command*]

Adds a new variable to *Dataset* by evaluating *Expression* for each row of *Dataset*. *Expression* may refer to variables in *Dataset* by name. For example, to calculate the processing time per node of a search, *Expression* might be `(/ nodesvisited totaltime)`.

:SORT *(Variable)* [*Command*]

Sorts *Variable* in ascending order.

:INDEX *(Variable) (Index)* [*Command*]

Sorts *Variable* by *Index*. Both must be CLASP variables from the same dataset.

:ADD A CONSTANT *(Variable) (Number)* [*Command*]

Adds *Number* to *Variable*.

:NATURAL LOG *(Variable)* [*Command*]

Takes the natural log of *Variable*.

:LOG10 *(Variable)* [*Command*]

Takes the common log (base 10) of *Variable*.

:RECODE CATEGORICAL *(Variable) (Old values) (New values)* [*Command*]

Substitutes each member of *New values* for the corresponding member of *Old values* in *Variable*. This can be used to create an integer coding of a categorical variable.

:REASSIGN BINS *(Variable) (Bin limits) (Bin names)* [*Command*]

Allows the conversion of a continuous variable into a categorical variable. *Bin limits* are the upper limits on the bins, and *Bin names* are the names of the corresponding bins. The list of *Bin names* must be one element longer than the list of *Bin limits*. The last element of *Bin names* will name the bin which holds any values of *Variable* which are larger than the largest element of *Bin limits*.

:SMOOTH *4253h (Variable)* [*Command*]

Apply a 4253h smoothing operator to *Variable*. This is described in greater detail in section 6.

:DISCRETE DERIVATIVE *(Variable)* [*Command*]

Take the pairwise difference of each value of *Variable* with the next value.

4.5.6 The Test Menu

Test
Confidence Interval Using z Statistic
Confidence Interval Using t Statistic
Confidence Interval of a Proportion
t-test One Sample
t-test Two Samples
t-test Matched Pairs
D test
Anova - One Way
Anova - Two Way
Chi-Square Counts
Chi-Square 2x2
Chi-Square RxC
Linear Regression - brief
Linear Regression - verbose
Multiple Linear Regression - verbose

The commands in the Test menu are described in section 5.2, and so only their arguments are given here.

:CONFIDENCE INTERVAL USING Z STATISTIC *(X(s)) (Confidence level)* [*Command*]

Confidence level is a real number from 0 to 1 inclusive. Note that it is not a mapping argument.

:CONFIDENCE INTERVAL USING T STATISTIC *(X(s)) (Confidence level)* [*Command*]

Confidence level is a real number from 0 to 1 inclusive. Note that it is not a mapping argument.

:CONFIDENCE INTERVAL OF A PROPORTION *(Successes) (Trials) (Confidence level)* [*Command*]

Successes is an integer ≥ 0, *Trials* is an integer ≥ 1, and *Confidence level* is a real number from 0 to 1 inclusive.

:T-TEST ONE SAMPLE *(X(s)) (Mean(s)) (Tails)* [*Command*]

Tails must be one of `Both`, `Positive` or `Negative`.

:T-TEST TWO SAMPLES *(Y(s)) (X(s)) (Tails)* [*Command*]

Tails must be one of `Both`, `Positive` or `Negative`.

:T-TEST MATCHED PAIRS *(Y(s)) (X(s)) (Tails)* [*Command*]

Tails must be one of `Both`, `Positive` or `Negative`.

:D TEST *(Y(s)) (X(s)) (Tails)* [*Command*]

Tails must be one of `Both`, `Positive` or `Negative`.

:ANOVA - ONE WAY *(Y(s)) (X(s))* [*Command*]

Displays an anova table, a list of the group means, a plot of the group means, a scheffe table and an alternate calculation of the sum of squares total.

:ANOVA - TWO WAY *(Y(s)) (X 1(s)) (X 2(s))* [*Command*]

Displays an anova table, a table of the cell means, and row and column based means plots.

:CHI SQUARE COUNTS *(A) (B) (C) (D)* [*Command*]

Performs a chi square analysis on the contingency table
A B
C D.

:CHI SQUARE 2X2 *(Y(s)) (X(s))* [*Command*]

Displays the Chi Square statistic, the G statistic and contingency tables expressed in terms of frequency, percent of row totals, percent of column totals and expected values. Y and X must each be two-valued variables.

:CHI SQUARE RXC *(Y(s)) (X(s))* [*Command*]

Displays the Chi Square statistic, the G statistic and contingency tables expressed in terms of frequency, percent of row totals, percent of column totals and expected values.

:LINEAR REGRESSION - BRIEF *(Y(s)) (X(s))* [*Command*]

Displays the slope, intercept, r^2, standard error of the slope and significance.

:LINEAR REGRESSION - VERBOSE *(Y(s)) (X(s))* [*Command*]

Displays all the information the :LINEAR REGRESSION - BRIEF does, and in addition, it displays the correlation, an anova table and a regression plot.

:MULTIPLE LINEAR REGRESSION - VERBOSE *(Y) (X's)* [*Command*]

Performs a multiple linear regression of Y on X*'s*, displaying the correlation, an anova table, the F statistic, the coefficients, betas and t statistic for each X, and a correlation matrix.

4.5.7 The Sample Menu

Sample
Sample from Uniform Distribution Sample from Normal Distribution Sample from Binomial Distribution Sample from Poisson Distribution Sample from Gamma Distribution

The Sample menu contains commands which take samples from various theoretical distributions. The variables in the dataset will be sampled from the specified distribution. In addition to the distribution parameters, all the sampling commands take a `Size of`

`Sample` argument, indicating how many values each variable should contain, and a `Number of Samples` argument indicating how many variables to create. The following descriptions only list the parameters of the distributions.

:SAMPLE FROM UNIFORM DISTRIBUTION *(Minimum) (Maximum)* [*Command*]

Will produce samples that are uniformly distributed between *Minimum* and *Maximum*, which must both be integers.

:SAMPLE FROM NORMAL DISTRIBUTION *(Mean) (Standard Deviation)* [*Command*]

Mean and *Standard Deviation* are both real numbers.

:SAMPLE FROM BINOMIAL DISTRIBUTION *(P) (N)* [*Command*]

P is a real number from 0 to 1, it is the probability of success. N is an integer ≥ 1, it is the number of Bernoulli trials per sample.

:SAMPLE FROM POISSON DISTRIBUTION *(Mean)* [*Command*]

Mean is a real number.

:SAMPLE FROM GAMMA DISTRIBUTION *(Integer Order)* [*Command*]

Integer Order is an integer ≥ 1.

Chapter 5

Statistics

Some computations on our data are intended merely to summarize, to distill the general trend or tendency, to help us see the signal by eliminating the noise. Other computations are meant to answer a question, to say whether this data is like that data or whether this model fits the data. The former computations are called descriptive statistics and the latter are called inferential or test statistics. In practice, the line between them is rather blurry. For example, fitting a model to the data describes its basic shape, but asking *whether* the model fits the data is a test, yet many statistical computations do both simultaneously. Nevertheless, CLASP's functions are categorized this way with respect to its menus and so is this manual, to help in organizing them.

5.1 Descriptive Statistics

Descriptive statistics are usually a function from the data to a single number. One example is the arithmetic mean, which we are all familiar with and which describes the data by a particular kind of central tendency or location. Some of the following statistics may be unfamiliar to you, but the basic goals are these: what is the data's location, how spread out is it, and how strong is its association with other data?

5.1.1 Location

The following statistics describe the "location" of the sample, usually the "center" of the sample, for some reasonable definition of the center.

Mean The mean is the average of a group of numbers, which describes the center of mass of the sample. This value is a good measure of the central tendency of a population; however, it is affected by a skewed distribution or when there are extreme outliers.

Median Like the mean, the median is another measure of central tendency; it indicates the value at the center of the distribution. Half of the scores will be above and half below. The median is less affected by outliers than the mean, since it is insensitive to how far the numbers are from the median. However, like the mean, the median is also affected by skewed distributions. The larger the difference between the median and the mean, the more likely that the distribution is not normal.

Quantile A generalization of the median is a quantile: given an fraction q, the qth quantile is a number x such that q of the data is greater than x and $1-q$ is less than x. Clearly, the median is simply the 1/2 quantile.

Trimmed Mean Discarding the highest and lowest values in a distribution and averaging the remaining values yields the trimmed mean. It is common to discard the highest and the lowest 25 percent of the values, but other trimming percentages may be used. Here is a sorted distribution:

(1 1 2 15 16 18 19 22 23 28 31 100)

The 25 percent trimmed mean is the mean of the middle six numbers. The trimmed mean is a robust alternative to the mean and is in some ways preferable to the median.

Mode The mode is the most common value in a distribution; for example, the mode of the distribution (1, 2, 2, 3, 4, 4, 4) is 4. If the data are continuous, real numbers, then the mode is apt to be uninformative because of the very low probability that two or more numbers will have exactly the same value. One solution is to map real-valued data into discrete numbers by rounding or sorting into bins for frequency histograms and so on, in which case the mode of a distribution depends on bin size. Although it is a contradiction in terms, we often speak of a distribution having two or more modes. This means the distribution has two or more values or ranges of values that are common. The decision to characterize a distribution as bimodal or multimodal is subjective.

Minimum and Maximum The minimum and maximum of a sample give an idea of its location, though certainly not of its central tendency. They are often ignored in statistics books, which concentrate on statistics that are used in hypothesis testing, such as the mean. Nevertheless, they are important because they help to visualize the data, outlining its location and helping to identify outliers, which are sometimes the most important elements of the data because they are most in need of explanation.

5.1.2 Spread

The following measures describe how much the data spreads out around its central value.

Range The range is simply the difference between the maximum and the minimum, and consequently indicates how spread out the data are.

Variance Variance is an important measure of how spread out a sample is around its center of mass—the mean. The higher the variance, the more spread out the sample. Mathematically, the variance is sum of the squared distances from the mean of each number in the sample. For example, the mean of (1 2 3 4 5) is 3 and the variance is 10, while the mean of (0 1 3 5 6) is also 3 but the variance is 26, reflecting the greater distance of the data from the mean.

Standard Deviation This statistic is simply the square root of the variance. The standard deviation is useful because its units are the same as the data's. For example, if the data are distances in meters, the mean will also be in meters, while the variance will be in meters-squared. The standard deviation will be in meters, and can be usefully visualized as the "average" deviation of the data from the mean. If the data are normally distributed, 96% of the values will fall within a range of plus or minus 2 standard deviations from the mean.

Interquartile Range The interquartile range is a robust alternative to the variance and standard deviation; that is, it measures how spread out the data are, but it is less sensitive to one or two values that are wildly far away. The interquartile range is found by dividing a sorted distribution into four contiguous parts, each containing the same number of individuals. Each part is called a quartile. (The quartiles are the 1/4, 1/2 and 3/4 quantiles.) The difference between the highest value in the third quartile and the lowest value in the second quartile is the interquartile range. For example, if the sorted distribution is (1 1 2 3 3 5 5 5 5 6 6 100), then the quartiles are (1 1 2), (3 3 5), (5 5 5), (6 6 100), and the interquartile range is $5 - 3 = 2$.

5.1.3 Covariance

Pearson's Correlation This statistic describes the degree of association between two samples. Alternatively, it describes how well the value of the dependent, Y, variable can be predicted from the score of the independent, X, variable. This relationship does not imply causality; it simply indicates that a high value of Y is likely to co-occur with a high value of X. The idea of co-occurrence means that the data must have some kind of pairing: (x_i, y_i). This statistic is considered by some to be overused and abused, but generally it is a good indicator of an effect or relationship between two variables.

Cross-Correlation Given two samples, X and Y, we can generalize the notion of correlation to ask what is the correlation between X and Y when Y is shifted or translated by some amount. This idea often arises when analyzing data collected over time, where you think that there will be a strong correlation between, say, daily barometric pressures in Buffalo and daily barometric pressure in Boston two days later. The time-shift is often called a "lag," and is an additional argument to the cross-correlation function.

Autocorrelation Autocorrelation is simply the cross-correlation of a variable with itself. Obviously, this is uninteresting unless the lag is non-zero. This is a way of measuring cyclic patterns in the sample.

5.2 Test Statistics

Many different kinds of questions can be asked of our data, and so there are many kinds of test statistics. Some questions ask simply whether two groups of numbers are different, specifically whether their central tendencies (such as their means) are different. Others ask whether there are differences among many groups of data. Still others fit models, such as a straight line or a curve, to the data and ask whether it fits well enough or just fits by chance. This chapter cannot explain all of these statistical tests in detail, but there are many excellent statistics texts available. Two good general texts are *Statistical Reasoning* *in Psychology and Education* by Edward W. Minium [9] and *Probability and Statistics for Engineering and the Sciences*, by Jay L. Devore [4].

Test statistics fall into several categories: parametric, non-parametric, and bootstrap, each of which is most applicable for certain specific types of problems. Parametric statistics make assumptions about the shape of the population from which the data was drawn; typically, it assumes the population is normal, but with unknown mean and variance. (These parameters are estimated from the sample, hence the term "parametric" statistics.) Parametric statistics are the best available, but only when their assumptions are true. If the assumptions are false, it is hard to determine the degree of error introduced into the analysis. Non-parametric statistics avoid making distributional assumptions, and so are more widely applicable. However, they are less powerful than their parametric competitors, which means that on the home turf of the parametric statistics (normal distributions), the parametric statistics are better. Non-parametric statistics typically work by ignoring the absolute magnitudes of the data and instead using only their relative magnitudes; essentially, they sort the data and map the original numbers onto 1...N, because the behavior of the numbers 1...N can be analyzed and tables precomputed. James Bradley has an excellent book on non-parametric statistics [2]. Bootstrap statistics also avoid making distributional assumptions, but they retain the absolute magnitudes of the data. They substitute computation for analysis and tables, essentially by computing tables as necessary. Bootstrap statistics are even more widely applicable than non-parametric statistics because there is no reliance on precomputed tables. They are, however, very computation-intensive. A much more detailed discussion may be found in Cohen's primer [3].

This section will briefly review the test statistics that are implemented in CLASP and their uses.

5.2.1 Confidence Intervals

Descriptive statistics of the sample are often used to estimate the equivalent statistic on the population. For example, one can estimate the mean of the population by the mean of the sample. Of course, the two will never be exactly equal, which gives rise to the question: how close is the estimate? One way of answering this is by constructing "confidence intervals." A confidence interval is a range, a to b, centered around the sample statistic. It also has an associated probability, say 90 percent. What does this probability mean? Its meaning may be best understood with an analogy, where the size of the confidence interval $b - a$ corresponds, say, to the size of a horseshoe in a game of horseshoes. A bigger horseshoe has a higher probability of landing around the stake, which corresponds to the parameter we are estimating. Equivalently, if we were to construct 90 percent confidence intervals for many samples, 90 percent of the confidence intervals would contain the desired parameter.

CLASP has two ways of computing confidence intervals, both for means, where the difference depends on the population the mean is from. "Confidence Interval Using Z Statistic" is for large samples, where the Central Limit Theorem assures us that the sampling distribution of the mean is normal. "Confidence Interval Using T Statistic" is for small samples, where the sampling distribution of the mean is a t-distribution.

Bootstrapping is a general technique for deriving confidence intervals for any statistic on any sample, and so confidence intervals for other statistics can easily be obtained. These will be implemented in future releases of CLASP.

5.2.2 t Tests

To test whether two means are significantly different, use a t test. The t test is reasonably robust, even for small samples, provided the underlying assumption of normal populations is not badly violated.

CLASP supports three versions of t tests. The *one-sample* t test is used to test the hypothesis that a sample mean has a specific value. For example, imagine you are a burglar, driving through an unfamiliar part of town, trying to judge whether the houses are worth robbing. You decide to base your decision on the price of the cars parked on the street. One short cul de sac appears promising: it has just five parked cars, but they all cost a lot. In fact the mean price of the cars is \$20,270 and the standard deviation of this sample is \$5811. You know that the mean cost of cars in town is \$12,000, and you want to know whether the cars in the cul de sac are significantly dearer than the "average" car in town. The test statistic is:

$$t = \frac{20270 - 12000}{5811/\sqrt{5}} = 3.18$$

The probability of attaining this result by chance, under the null hypothesis that all cars cost on average \$12,000, is less than .01.

To compare two means, use a two-sample t test. For example, to test the hypothesis that one search algorithm expands significantly more nodes than another, on average, simply enter the sample for each algorithm in a separate column, and CLASP will calculate the t statistic:

$$t = \frac{\bar{x}_1 - \bar{x}_2}{\hat{\sigma}_{\bar{x}_1 - \bar{x}_2}}$$

CLASP will also return a p-value for the t statistic. If the p-value is small (conventionally, .05 is considered small and .01 very small), the test indicates that the two samples are significantly different.

A special case of a two-sample test is a *paired-sample t test.* To continue the previous example, imagine that the two search algorithms were tested on the same problems, so for each problem you have a pair of scores, one for each algorithm. Under the null hypothesis that the algorithms are equal, the expected value of the difference of scores is zero. The paired-sample t test computes the average difference (summing the differences and dividing by the number of problems) and then runs a one-sample test of the hypothesis that the average difference is zero.

Randomization versions of the one-sample, two-sample and paired-sample t tests are available in CLASP. These tests substitute massive amounts of computation for parametric assumptions about the sampling distribution of the mean. Thus, they are the preferred alternative when one suspects that one's data are not drawn from a normal populations. Returning to the burglary example, above, we know that the distribution of automobile prices is not normal; rather, it is skewed to the right (high costs). For such data, or when you simply do not know the form of the underlying population distribution, use the randomization versions of t tests.

5.2.3 Analysis of Variance

Analysis of Variance (ANOVA) may initially be thought of as a generalization of the t-test to multiple groups. Rather than test whether the means of two groups are equal, it tests whether the means of k groups are equal. Analysis of variance is so named because it identifies the amount of variation in a sample due to the effect of some treatment, where each different treatment results in a different group.

There are two forms of ANOVA implemented in CLASP, "One Way" and "Two Way." One-way ANOVA is used to examine the effect of a single treatment on several samples, where the treatment is administered at several discrete levels. For example, in the Phoenix firefighting system, we once did an experiment with three levels of wind speed—low, medium and high—and performed a one-way ANOVA to test whether wind speed had an effect on performance. Two-way ANOVA is used when considering the effects of multiple treatments and their interactions. For example, in the Phoenix firefighting system, we might have run an experiment with three levels of wind speed and two kinds of firefighting plans—called multiple-shell and model-based. We could use a two-way ANOVA to test whether the performance of the plans depends on wind speed and which plan is better.

Each analysis of variance is calculated in a similar fashion and the results reported in an ANOVA table. Values used in calculating the ANOVA table are based on the sum of squares of the deviation scores, SS, and the degrees of freedom, df, associated with that sum of squares. The general relationship is in the form

$$s^2 = \frac{SS}{df}$$

The types of variation involved for one-way ANOVA are those of:

1. Variability of the values about the grand mean (the mean of all scores). Given by the deviation: $(X - \overline{\overline{X}})$, where $\overline{\overline{X}}$ is the grand mean. This is the known as the *total* variance, symbolized as s_T^2. This value is useful only in the calculating of the following items.

2. Variability of scores about their subgroup sample means. Given by the deviation: $(X - \overline{X})$,where $\overline{X}$ is the mean of the subgroup which contains X. This is known as the *within groups* variance, denoted by s_W^2.

3. Variability of subgroup sample means about the grand mean of all scores. Given by the deviation: $(\overline{X} - \overline{\overline{X}})$. This is the *among groups* variance, denoted by, s_A^2.

The *within groups* variance indicates the amount of variation inherent in the subgroup, all of the members of the group receive the same treatment so none of the variation can be due to the treatment. The *among groups* variance contains two components, the variance inherent in the population and the variance due to the treatment. By examining these components, the effect of a treatment on a population can be determined.

Calculations for these values are described by the formulas in Figure 5.1, where X_i are the scores in subgroup i, $\overline{X}_i$ is the mean of subgroup i, n_i is the number of observations in the ith subgroup and k is the number of subgroups.
Using these values, the F ratio is calculated. The F ratio indicates the probability that the variation in the samples could be accounted for by chance. Calculation of the F ratio uses the following formula:

$$F = \frac{s_A^2}{s_W^2}$$

For two-way ANOVA, there is an additional interaction between different treatments, sometimes called factors. These are separated into *Row* and *Column* treatments. In our second example, above, we had two treatments, wind speed and plan, and so one becomes

$$
\begin{aligned}
SS_W &= \sum(X_i - \overline{X}_i)^2 + \sum(X_j - \overline{X}_j)^2 + \cdots \\
SS_A &= \sum_i^k(\overline{X}_i - \overline{\overline{X}})^2 \\
SS_T &= \sum(X - \overline{\overline{X}})^2 \\
df_W &= \sum(n_i - 1) \\
df_A &= k - 1 \\
df_T &= \sum n_i - 1 \\
s^2_W &= \frac{SS_W}{df_W} \\
s^2_A &= \frac{SS_A}{df_A}
\end{aligned}
$$

Figure 5.1: One Way ANOVA Calculations

$$
\begin{aligned}
SS_C &= \sum(\overline{X}_{C_i} - \overline{\overline{X}})^2 \\
SS_R &= \sum(\overline{X}_{R_i} - \overline{\overline{X}})^2 \\
SS_{WC} &= \sum(X - \overline{X}_{cell})^2 \\
SS_{R\times C} &= SS_T - SS_C - SS_R - SS_{WC} \\
df_{WC} &= \sum^{all\ cells}(n_{WC} - 1) \\
df_C &= (C - 1) \\
df_R &= (R - 1) \\
df_{R\times C} &= (C - 1)(R - 1) \\
df_T &= (R)(C)(n_{WC}) - 1 \\
s^2_C &= \frac{SS_C}{df_C} \\
s^2_R &= \frac{SS_R}{df_R} \\
s^2_{WC} &= \frac{SS_{WC}}{df_{WC}} \\
s^2_{R\times C} &= \frac{SS_{R\times C}}{df_{R\times C}}
\end{aligned}
$$

Figure 5.2: Two Way ANOVA Calculations

the row (say, wind speed, giving three rows: low, medium, and high) and the other becomes the column (plan, giving two columns: multiple-shell and model-based), and each cell of the table is a particular combination of the two. Having two treatments means there may be an additional effect called an interaction effect, where the condition of, say, low wind can combine with condition of, say, the multiple-shell plan, to produce a different performance. This interaction effect, also called a *Row* by *Column* effect, is calculated with the formulas in Figure 5.2, where the four variance estimates correspond to:

s^2_{WC} *within cells estimate*, derived from the individual cell variation. This measures the inherent variation in an subgroup free from the effect of any treatment.

s^2_C *column estimate*, derived from the differences from the column means. If there is an effect from the column treatments this value will tend to be larger than s^2_{WC}.

s^2_R *row estimate*, as above but for row treatments.

$s^2_{R\times C}$ *interaction estimate*, derived from the discrepancy between the means of several cells. This value will tend to be larger than s^2_{WC} if there is an interaction effect between the *Row* and *Column* treatments.

There are three F ratios calculated—column, row, and row by column—in the form:

$$\begin{aligned} F &= \frac{s^2_C}{s^2_{WC}} \\ F &= \frac{s^2_R}{s^2_{WC}} \\ F &= \frac{s^2_{R\times C}}{s^2_{WC}} \end{aligned}$$

A significant value of F indicates that the hypothesis that the means of the subgroups are equal, that is that there is no variance due to the treatment, should be rejected. Two way ANOVA requires that each of the cells have the same number of observations in order to calculate a meaningful statistic, because unequal group sizes make the statistic unstable.

5.2.4 Linear Regression

Many statistical tests can be thought of as fitting a particular kind of model to the data then measuring how plausible it is to have drawn the data from that model. For t-tests and ANOVA, those models are simple Gaussian distributions. A slightly more complex model, but still very simple, is a linear model. Essentially, we think the data points are drawn from a line with "fuzz" around it—Gaussian fuzz with constant variance:

$$y_i = mx_i + b + \epsilon$$

In this equation, m is the slope of the line, b is its y-intercept, and ϵ represents the Gaussian fuzz around the line. Linear regression is a way to find m and b and also to estimate how big the ϵ's are and how plausible it is that the data is generated by this linear model.

A plot of the regression line with a scatter plot of the individual data points provides a picture of the relationship between the variables and also gives a good indication of the correlation between the variables. It is important to remember that there may be relationships between the variables which are non-linear, such as a logarithmic or quadratic relationship. Don't use linear regression if the relationship between the variables is clearly non-linear.

Multi-variate linear regression performs the same calculations with multiple independent variables, using the model

$$Y = a + b_1X_1 + b_2X_2 + \cdots + b_nX_n + \epsilon$$

where the b_i are the *coefficients of regression* for the corresponding X, a is the intercept of the regression line, and ϵ is the error, or residual. Each b has a corresponding t statistic indicating the probability of whether or not the relationship could occur by chance. A significant b indicates that a portion of the variance of the dependent variable is due to the influence of the associated independent variable. The amount of the influence is measured as a percentage of the total variance of the dependent variable, which is calculated as the square of the correlation between Y and X_i. The percentage of the variance due to all of the independent variables is given by R^2, with the F statistic for the regression calculated from this value. The report for this procedure includes the correlation coefficients between the independent variables.

5.2.5 Chi Square

Chi square is a useful statistic when we are counting discrete things, such as the number of birds of each of several species, and we are interested in whether the observed proportions (such as ten percent big black birds and ninety percent little brown birds) is likely to have been drawn from some theoretical population. This is done by subtracting the expected number of events (for example, the expected number of little brown birds calculated from our theoretical population and the number of birds we saw) from the actual number of events, squaring the difference, and summing over each category of event. (There is also a normalizing factor in each term.) By assuming that deviations are normally distributed, the probability of drawing the sample from the theoretical population can be calculated. Hence, chi square is a parametric statistic. With a slightly different calculation, we can compare two samples to test whether they are the same, in the sense of having roughly the same distribution.

An important special case of the chi square is to test for independence. Again, we are counting events, and the events fall into categories, but the categories fall along two different dimensions. The question we are asking is whether the two dimensions are independent: the distribution of events along one dimension is the same for each category of events along the other, and vice versa. For example, suppose we are tossing a nickel and a penny, and we are counting the outcomes. There are four possible events, falling into categories along two dimensions, namely the identity of the coin. The following figure is a 2x2 "contingency table" to represent the data, where each cell is the observed frequency for the combination of the two categories.

		penny	
		H	T
nickel	H	a	b
	T	c	d

We can use "Chi square" to analyze the contingency table and test whether the two coins are independent. (We would be surprised if they weren't.) Larger contingency tables can be handled by CLASP.

When the sample is small, the deviations can't be normally distributed, since the data are discrete integers. In that case, the user can choose *Yate's correction*, which adjusts for the discontinuity.

Chapter 6

Data Manipulation

CLASP data is organized into variables and datasets. A variable is an ordered column of numbers which have some semantic interpretation, such as "Number of Ships Allocated" in a transportation simulator. The values might represent different measurements over time or measurements during different trials in an experiment. A dataset is a collection of variables, all of which have the same number of values. Usually a dataset will represent the variables collected during a single experiment or set of experiments. In CLASP, all variables belong to a dataset. Datasets are analogous to tables of data, where variables are columns, and a row is a set of the ith value in each column. For example, in an experiment where thirty trials are run and, during each trial, "Runtime," "Num-Nodes" and "Goal-Found?" were measured, the resulting dataset would have thirty rows, each corresponding to one of the trials, and three Variables, "Runtime," "Num-Nodes" and "Goal-Found?" Most statistical operations involve performing some operation on the values of one or more variables. For example, the mean operates on one variable and the t-test compares two variables to test if they have statistically different means. CLASP offers a variety of ways of accessing these values for statistical manipulation.

6.1 Function Application

Most data manipulation consists of transforming one or more variables by mapping a function across them. You may think of this as equivalent to the Common Lisp 'map' function, in which each variable is treated as a sequence. By typing in an expression, you can transform your data in arbitrary ways. Certain standard operations, such as log transforms, are built in.

6.1.1 Formula Expressions

Transformations of a variable are accomplished by choosing "User Defined" from the transform menu. The user is prompted for a dataset and an expression. The formula can be any valid Common Lisp expression and may contain variable names. The formula is mapped over all the variables it contains and applied to each successive value. The result is a new variable which is added to the dataset. For example, the following procedure would produce the ratio of "Nodes Searched" to "Runtime" in the dataset "My Data."

1. Click on Transform Menu
2. Click on User Defined

3. CLASP prompts for a dataset, Click on My Data
4. CLASP prompts for a formula, type: (/ Nodes-Searched Runtime)

A new variable will be created called /-Nodes-Searched-Runtime.

Often a categorical variable will need to be converted to a numeric variable to allow certain statistics to be applied to it. For instance a simulator that reports wind-speed as "low", "medium" or "high", but for analysis, the user prefers the encoding to be 1, 2 and 3. Alternatively, categorical variables which are encoded as integers might need to be converted to a nominal encoding for ease of interpretation. For example, a variable which encodes the completion status of an algorithm as 1 or 0 might be converted to "success" or "failure" for ease of interpretation. This is called *recoding.* Formulas may include the *recode* operator. The form of this expression is:

(recode Variable old-values new-values)

Recoding will translate all the elements of variable in old-values to the equivalent new-value. For instance, if a variable is valued

(A D A C B A B D C)

and the expression

(recode Variable '(A B C D) '(1 2 3 4))

is used in a "Transform" statement, the new variable will have values

(1 4 1 3 2 1 2 4 3)

This kind of transformation can be valuable for running statistics that require numeric data when the original data is symbolic.

6.1.2 Smoothing

Time series and other series plots are sometimes difficult to read because of rapid fluctuations. For example, the time series of mean daily temperature bounces all over the place and can make it difficult to see slower-moving seasonal components. Spectral analysis is sometimes used to decompose series into frequency components, but yields a plot in the frequency domain, not the time domain. Often, it suffices to simply remove the high-frequency component from a time series by applying *smoothing* operators. The basic idea of smoothing is simple: every value in the series is replaced by an average of the value and its neighbors. (Endpoints are handled separately; for example, they might be used as is, or averaged with their neighbor.) The average need not be the arithmetic average, in fact, it is often better to use the median instead of the mean. In either case, values that are much bigger or smaller than their neighbors are "brought into line." The most common mean and median smoothing techniques involve replacing the ith value in the series with the mean or median of the i-1, i, and i+1 values. These are called "3" smooths because they involve three values, a *window* of size 3.

Median smooths handle outlier values differently than mean smooths. Whereas median smooths ignore outliers, leaving only small values, mean smooths average outliers and neighboring values, giving the impression that the neighbors have higher values than they actually do. On the other hand, median smooths create *mesas* or short sequences of identical values.

Mesas can be handled by *resmoothing* that is, smoothing the smoothed series again. One can resmooth a sequence many times, with various sizes of windows, and various kinds of averaging—not only median and mean smoothing, but also weighted smoothing in which each value in a window is multiplied by a weight. One can create long and complicated smoothing plans. For example, 3R2H means "smooth the sequence with 3,median smooths repeatedly until the appearance of the series doesn't change with successive smooths (that's what the R means), then apply a 2,median smooth, and then a hanning operation (that's what H means). A hanning operation multiplies the three values in a window by .25, .5 and .25, respectively, and sums the results. It is used to "clean up" a smooth by removing any leftover spikes.

Many smoothing plans have been developed for different kinds of data, but one that works pretty well in general is 4253H. This is four median smooths, with window sizes 4, 2, 5 and 3, followed by a Hanning operation. Currently, CLASP offers four median smoothing operators with window sizes 2, 3, 4, and 5, respectively; and the general smoothing plan 4253H.

6.2 Partition Clauses

A common manipulation of data is to reorganize it, say by grouping it or looking at a fraction of it. Most of these kinds of manipulation can be accomplished by *partitioning*, which essentially means mapping over the data, grouping it, and doing something with each group. (You might discard some groups in order to look more closely at others.)

It is often necessary to apply a command to a subset of the rows in a dataset. Sometimes this is because the statistics being used are only meaningful for that subset or because different operations must be done to different parts of the dataset. For instance, suppose you collect data on the time it takes an algorithm to solve a problem; it is meaningless to try and compute the mean solution time in cases where an algorithm couldn't find a solution. Instead, you can select from the dataset just those cases where the algorithm succeeded and then take the mean of the runtimes in that selection. In other situations, it might be desirable to break the dataset up into a number of subsets by using one or more key variables. For every row in the dataset, the value of the key variable(s) at that row determine which new partition the row will be assigned to. This is often useful when a factorial analysis of data is being done.

6.2.1 Selection

In CLASP, the "Partition Dataset" command does selection. The form of the command is

partition dataset partition-clause

Partition clauses are in prefix notation (operator first, arguments last) and must be enclosed in parentheses. In fact, they are simply lisp expressions in which symbols denote variables in a dataset. Effectively the symbol is replaced by the value of the corresponding variable for the particular row being processed.

```
(and (== success t)
     (or  (<= nodes-searched 5)
          (< 50 tree-depth 101)))
```

This clause would select those rows of the dataset where the "success" variable was true, and either the "tree-depth" was less than or equal to 5 or the "nodes-searched" was between 50 and 101 exclusive.

6.2.2 Partitioning

In CLASP, the "Partition-On" command does partitioning. The form of the command is

partition-on dataset variables

The dataset is partitioned on the variables. It is not advisable to partition a dataset on a variable that has many different values, since you'll get an unmanageable number of partitions. In particular, continuous variables are very bad, since it is likely that almost every row will end up in its own partition.

This operator makes several partitions from a dataset—one for each distinct value of the variables. Each resulting dataset will contain all the rows of the original dataset which share the same variable values. Here is an example. We start with the following table:

My Data

Key-1	Key-2	Time	Temp
1	A	10	52.1
1	A	12	53.9
2	A	9	52.6
2	B	7	48.2
1	C	10	50.2
2	B	13	47.4
1	C	12	51.5
2	A	6	55.1
2	C	11	46.7

Partitioning the dataset on Key-1 and Key-2 would result in the following 5 datasets

My Data where Key-1 = 1 and Key-2 = A

Key-1	Key-2	Time	Temp
1	A	10	52.1
1	A	12	53.9

My Data where Key-1 = 1 and Key-2 = C

Key-1	Key-2	Time	Temp
1	C	10	50.2
1	C	12	51.5

My Data where Key-1 = 2 and Key-2 = A

Key-1	Key-2	Time	Temp
2	A	9	52.6
2	A	6	55.1

My Data where Key-1 = 1 and Key-2 = A

Key-1	Key-2	Time	Temp
2	B	7	48.2
2	B	13	47.4

My Data where Key-1 = 1 and Key-2 = A

Key-1	Key-2	Time	Temp
2	C	11	46.7

Chapter 7

Functions

CLASP is designed so that the functions that actually compute statistics on data are separated from the functions that manipulate data in the CLASP database and also separated from the graphical interface (CLIM) functions. The purpose of this modularity is to allow users to call statistics functions directly on their own data, from their Lisp Read-Eval-Print loops (REPLs) or even from their own Lisp programs, without having to use the graphical interface or the database. Furthermore, it allows users to manipulate their data using the database functions, calling them from their own REPLs or their own Lisp programs. By using CLIP and CLASP, data can be collected and analyzed completely under program control. (Note that we have called this section "Functions" and referred to its subject as functions, but a few of them are macros, CLOS classes or methods, and other things written in Common Lisp.)

Having divorced the statistical functions from the database functions, we had to decide how data was to be presented to the statistical functions. We decided to use sequences, which have a long history of usefulness in Lisp programs. Some of the simpler functions, such as 'mean,' take sequence keyword arguments, such as `:start` and `:end` to indicate a subsequence, and `:key` to map sequence elements to actual numbers. These all have their standard Common Lisp syntax and semantics. Unless otherwise specified, you should assume that all of these functions take sequences of numbers.

The documentation for specific functions is pulled from their online documentation strings. Those documentation strings occasionally refer you to this manual, which will seem odd since you are reading the manual. However, this approach helps to keep the manual up to date.

7.1 Descriptive Statistic Functions

The functions in this section compute statistics that describe a sample, without making assumptions about the population it was drawn from. All of these functions take sequences of numbers.

The 'data-length' function is usually used to compute the n used in statistical computations, such as the mean. It is described here because it is so simple and doesn't fit anywhere very well.

`data-length` *data &key start end key* [*Function*]

Returns the number of data values in 'data.' Essentially, this is the Common Lisp 'length' function, except it handles sequences where there is a 'start' or 'end' parameter. The 'key' parameter is ignored.

7.1.1 Location

The functions in this subsection describe the "location" of the sample, usually the "center" of the sample, for some reasonable definition of the center.

`mean` *data &rest standard-args &key start end key* [*Function*]

Returns the arithmetic mean of 'data,' which should be a sequence. Signals 'no-data' if there is no data.

The following statistics are often ignored in statistics books, which concentrate on statistics that are used in hypothesis testing, such as the mean. Nevertheless, they are important because they outline the location of the data and help to identify outliers, which are sometimes the most important elements of the data because they are most in need of explanation.

`minimum` *data &rest standard-args &key start end key* [*Function*]

Returns the element of the sequence 'data' whose 'key' is minimum. Signals 'no-data' if there is no data. If there is only one element in the data sequence, that element will be returned, regardless of whether it is valid (a number).

`maximum` *data &rest standard-args &key start end key* [*Function*]

Returns the element of the sequence 'data' whose 'key' is maximum. Signals 'no-data' if there is no data. If there is only one element in the data sequence, that element will be returned, regardless of whether it is valid (a number).

The 'quantile' function is a generalization of medians, quartiles, and percentiles. Even more than the minimum and maximum, it helps to picture the data, identifying central values and outliers.

`quantile` *data q &rest standard-args &key start end key* [*Function*]

Returns the element which is the q'th percentile of the data when accessed by 'key.' That is, it returns the element such that 'q' of the data is smaller than it and 1-'q' is above it, where 'q' is a number between zero and one, inclusive. For example, if 'q' is .5, this returns the median; if 'q' is 0, this returns the minimum (although the 'minimum' function is more efficient). If there is no 'key,' this function returns a single number, which may be the mean of two numbers in 'data.' If there is a 'key,' this function also returns the two data elements whose keys bracket the desired quantile. If the quantile happens to be exact, these data elements may be eq.

`median` *data &rest standard-args &key start end key* [*Function*]

Returns the median of the subsequence of 'data' from 'start' to 'end', using 'key'. Works by calling 'quantile,' and that function has more complete documentation.

Essentially, the 'trimmed-mean' gets rid of the outliers and takes the mean of the central values. It is a robust statistic that still provides good estimates of the center of different kinds of population distributions, such as uniform, double-exponential, and normal.

`trimmed-mean` *data percentage &rest standard-args &key start end key* [*Function*]

Returns a trimmed mean of 'data.' A trimmed mean is an ordinary, arithmetic mean of the data, except that an outlying percentage has been discarded. For example, suppose there are ten elements in 'data,' and 'percentage' is 0.1: the result would be the mean of the middle eight elements, having discarded the biggest and smallest elements. If 'percentage' doesn't result in a whole number of elements being discarded, then a fraction of the remaining biggest and smallest is discarded. For example, suppose 'data' is '(1 2 3 4 5) and 'percentage' is 0.25: the result is $(.75(2) + 3 + .75(4))/(.75{+}1{+}.75)$ or 3. By convention, the 0.5 trimmed mean is the median, which is always returned as a number.

The mode of a distribution may not be at the "center" at all, but is certainly a representative value.

`mode` *data &rest standard-args &key start end key* [*Function*]

Returns the most frequent element of 'data,' which should be a sequence. The algorithm involves sorting, and so the data must be numbers or the 'key' function must produce numbers. Consider 'sxhash' if no better function is available. Also returns the number of occurrences of the mode. If there is more than one mode, this returns the first mode, as determined by the sorting of the numbers.

`mode-for-continuous-data` *data &rest standard-args &key start end key window* [*Function*]

Returns the most frequent element of 'data,' which should be a sequence. The algorithm involves sorting, and so the data must be numbers or the 'key' function must produce numbers. Consider 'sxhash' if no better function is available. Also returns the number of occurrences of the mode. If there is more than one mode, this returns the first mode, as determined by the sorting of the numbers.

Keep in mind that if the data has multiple runs of like values that are bigger than the window size (currently defaults to 10function will blindly pick the first one. If this is the case you probabaly should be calling 'mode' instead of this function.

The following is a generalization of the "mode" function. If the 'k' parameter is quite large, this function can count the number of occurrences of each data value. Of course, that computation could take a long time, as the complexity is roughly $O(kn)$.

`multiple-modes` *data k &rest standard-args &key start end key* [*Function*]

Returns the 'k' most frequent elements of 'data,' which should be a sequence. The algorithm involves sorting, and so the data must be numbers or the 'key' function must produce numbers. Consider #'sxhash if no better function is available. Also returns the number of occurrences of each mode. The value is an association list of modes and their counts. This function is a little more computationally expensive than 'mode,' so only use it if you really need multiple modes.

7.1.2 Spread

The functions in this subsection describe how much a sample spreads out around its center.

The 'sum-of-squares' function is rarely useful in itself, but is a common building block for statistical computations such as the variance or the analysis of variance. It is given by the following formula:

$$\sum_{i=1}^{n}(x_i - \bar{x})^2$$

The variance is this divided by $n - 1$ and the standard deviation is the square root of the variance.

`sum-of-squares` *data &rest standard-args &key start end key* [*Function*]

> Returns the sum of squared distances from the mean of 'data'. Signals 'no-data' if there is no data.

`variance` *data &rest standard-args &key start end key* [*Function*]

> Returns the variance of 'data,' that is, the 'sum-of-squares' divided by n-1. Signals 'no-data' if there is no data. Signals 'insufficient-data' if there is only one datum.

`standard-deviation` *data &rest standard-args &key start end key* [*Function*]

> Returns the standard deviation of 'data,' which is just the square root of the variance. Signals 'no-data' if there is no data. Signals 'insufficient-data' if there is only one datum.

`range` *data &rest standard-args &key start end key* [*Function*]

> Returns the range of the sequence 'data.' Signals 'no-data' if there is no data. The range is given by max - min.

`interquartile-range` *data &rest standard-args* [*Function*]

> The interquartile range is similar to the variance of a sample because both are statistics that measure out "spread out" a sample is. The interquartile range is the difference between the 3/4 quantile (the upper quartile) and the 1/4 quantile (the lower quartile).

7.1.3 Correlation

When several measurements are taken of a system, so that we get a sample of points in two-, three-, or n-dimensions, we can not only describe the collection of values in each dimension, but also any relationships between them. One simple measure is their covariance or correlation, which is simply the normalized equivalent of covariance. (Covariance is rarely reported, since it depends on the units of measure.) Essentially, correlation measures the *linear* association between the values in one dimension and those in another; that is, the extent to which, say, high x values coincide with high y values.

The first question when we start to deal with multi-dimensional samples is representation. The two obvious possibilities are:

1. a sequence of points:

```
((x0 y0) (x1 y1) (x2 y2) ...(xn yn))
```

2. a collection of sequences:

```
(x0 x1 x2 ...xn)
(y0 y1 y2 ...yn)
```

We chose to use the latter representation because the database is organized that way—around *variables*, which are sequences of data values of a particular kind or dimension.

Note that these representations are easily converted from one to the other:

```
(setq pts '((0 5) (1 6) (2 8) (3 9) (4 10)))

(values (mapcar #'first pts) (mapcar #'second pts))
  => (0 1 2 3 4)
     (5 6 8 9 10)

(mapcar #'list (multiple-value-list *))
  => ((0 5) (1 6) (2 8) (3 9) (4 10))
```

If you want to avoid the consing that this involves, the function 'map-into' can let you use and reuse an existing array.

Because the 'covariance' and 'correlation' functions take several sequences, we allowed them each to take 'start' and 'end' parameters. However, there is no 'key' parameter.

Mathematically, the covariance is

$$\sum(x_i - \bar{x})(y_i - \bar{y})$$

which is algebraically equivalent to

$$\sum x_i y_i - (\sum x_i)(\sum y_i)/n$$

but the latter is more efficient to calculate.

`covariance` *sample1 sample2 &rest args &key start1 end1 start2 end2* [*Function*]

> Computes the covariance of two samples, which should be equal-length sequences of numbers. Covariance is the inner product of differences between sample elements and their sample means. For more information, see the manual.

`correlation` *sample1 sample2 &rest args &key start1 end1 start2 end2* [*Function*]

> Computes the correlation coefficient of two samples, which should be equal-length sequences of numbers.

The 'inner-product' function was written to be called by the 'covariance' and 'correlation' functions, but may be useful in its own right.

`inner-product` *sample1 sample2 &key start1 end1 start2 end2* [*Function*]

> Returns the inner product of the two samples, which should be sequences of numbers. The inner product, also called the dot product or vector product, is the sum of the pairwise multiplication of the numbers. Stops when either sample runs out; it doesn't check that they have the same length.

Finally, the computation for correlation doesn't really require all the data; it can get by with just a few "summary" statistics. If you have those summary statistics, you may prefer to call the following function directly. Keeping the summary statistics can be done incrementally in a constant amount of space, which can be very useful if you are collecting data during the run of a program but don't want to keep every value. Instead you can just keep a few running sums. There are a number of other statistical functions that work on summary statistics, and the same comments apply to them.

`correlation-from-summaries` *n x x2 y y2 xy* [*Function*]

> Computes the correlation of two variables given summary statistics of the variables. All of these arguments are summed over the variable: 'x' is the sum of the x's, 'x2' is the sum of the squares of the x's, and 'xy' is the sum of the cross-products, which is also known as the inner product of the variables x and y. Of course, 'n' is the number of data values in each variable.

See section 5.1 for a description of cross-correlation and autocorrelation

`cross-correlation` *sequence1 sequence2 max-lag &optional min-lag* [*Function*]

> Returns a list of the correlation coefficients for all lags from 'min-lag' to 'max-lag,' inclusive, where the 'i'th list element is the correlation of the first (length-of-sequence1 - i) elements of sequence1 with with the last i elements of sequence2. Both sequences should be sequences of numbers and of equal length.

`autocorrelation` *sample max-lag &optional min-lag* [*Function*]

> Autocorrelation is merely a cross-correlation between a sample and itself. This function returns a list of correlations, where the i'th element is the correlation of the sample with the sample starting at 'i.'

7.1.4 Summary

The following function just collects many, probably too many, of the descriptive statistic functions. It's an expensive but convenient way to say a lot about a sample in just ten numbers.

`statistical-summary` *data &rest standard-args &key start end key* [*Function*]

> Compute the length, minimum, maximum, range, median, mode, mean, variance, standard deviation, and interquartile-range of 'sequence' from 'start' to 'end', accessed by 'key'.

7.2 Test Statistic Functions

This section describes functions to test hypotheses and such. The reasoning behind them is somewhat complicated at times, and users should check that they understand the tests and their proper uses by referring to any good introductory statistics book, such as the ones in the references. Most of the tests are described in section 5.2, and will not be further described here.

7.2.1 t Tests

The functions to compute various t-tests are the following. We don't implement a z-test because it is just a special case of the t-test in which the samples are quite large. You can use the 't-test' function on large samples without any significant loss of accuracy or efficiency.

`t-test-one-sample` *data tails &optional (h0-mean 0) &rest standard-args &key start end key* [*Function*]

Returns the t-statistic for the mean of the data, which should be a sequence of numbers. Let D be the sample mean. The null hypothesis is that D equals the 'H0-mean.' The alternative hypothesis is specified by 'tails': ':both' means D /= H0-mean, ':positive' means D > H0-mean, and ':negative' means D < H0-mean. The function also returns the significance, the standard error, and the degrees of freedom. Signals 'zero-variance' if that condition occurs. Signals 'insufficient-data' unless there are at least two elements in the sample.

`t-test` *sample-1 sample-2 &optional (tails :both)* [*Function*]

Returns the t-statistic for the difference in the means of two samples, which should each be a sequence of numbers. Let D=mean1-mean2. The null hypothesis is that D=0. The alternative hypothesis is specified by 'tails': ':both' means D/=0, ':positive' means D>0, and ':negative' means D<0. Unless you're using :both tails, be careful what order the two samples are in: it matters! The function also returns the significance, the standard error, and the degrees of freedom. Signals 'standard-error-is-zero' if that condition occurs. Signals 'insufficient-data' unless there are at least two elements in each sample.

`t-test-matched` *sample1 sample2 &optional (tails :both)* [*Function*]

Returns the t-statistic for two matched samples, which should be equal-length sequences of numbers. Let D=mean1-mean2. The null hypothesis is that D=0. The alternative hypothesis is specified by 'tails': ':both' means D/=0, ':positive' means D>0, and ':negative' means D<0. Unless you're using :both tails, be careful what order the two samples are in: it matters! The function also returns the significance, the standard error, and the degrees of freedom. Signals 'standard-error-is-zero' if that condition occurs. Signals 'insufficient-data' unless there are at least two elements in each sample.

The d-test is a competitor to the t-test. It uses bootstrapping to estimate the sampling distribution of d under the null hypothesis. Thus, it avoids the distributional assumptions of the t-test at the price of a great deal of computation. The t-test will take less than a second of computation, while the d-test may take several seconds. For example, on two samples of size 20 on a TI Explorer II, the t-test took 11.7 milliseconds, while the d-test took 2.9 seconds. Thus, the relative cost of the d-test is very high, but its absolute cost is really quite small.

`d-test` *sample-1 sample-2 tails &optional (bootstrap-times 1000)* [*Function*]

Two-sample test for difference in means. Competes with the unmatched, two-sample t-test. Each sample should be a sequence of numbers. We calculate the mean of 'sample-1' minus the mean of 'sample-2'; call that D. Under the null hypothesis, D is zero. There are three possible alternative hypotheses: D is positive, D is negative, and D is either, and they are selected by the 'tails' parameter, which must be :positive, :negative, or :both, respectively. We count the number of chance occurrences of D in the desired rejection region, and return the estimated probability.

7.2.2 Confidence Intervals

Computing a confidence interval on a parameter depends on the parameter and its theoretical distribution. We have implemented just three at this time.

`confidence-interval-z` *data confidence* [*Function*]

Suppose you have a sample of 50 numbers and you want to compute a 90 percent confidence interval on the population mean. This function is the one to use. Note that it makes the assumption that the sampling distribution is normal, so it's inappropriate for small sample sizes. Use confidence-interval-t instead. It returns three values: the mean and the lower and upper bound of the confidence interval. True, only two numbers are necessary, but the confidence intervals of other statistics may be asymmetrical and these values would be consistent with those confidence intervals. This function handles 90, 95 and 99 percent confidence intervals as special cases, so those will be quite fast. 'Sample' should be a sequence of numbers. 'Confidence' should be a number between 0 and 1, exclusive.

`confidence-interval-t` *data confidence* [*Function*]

Suppose you have a sample of 10 numbers and you want to compute a 90 percent confidence interval on the population mean. This function is the one to use. This function uses the t-distribution, and so it is appropriate for small sample sizes. It can also be used for large sample sizes, but the function 'confidence-interval-z' may be computationally faster. It returns three values: the mean and the lower and upper bound of the confidence interval. True, only two numbers are necessary, but the confidence intervals of other statistics may be asymmetrical and these values would be consistent with those confidence intervals. 'Sample' should be a sequence of numbers. 'Confidence' should be a number between 0 and 1, exclusive.

`confidence-interval-proportion` *x n confidence* [*Function*]

Suppose we have a sample of 'n' things and 'x' of them are "successes." We can estimate the population proportion of successes as x/n; call it 'p-hat.' This function computes the estimate and a confidence interval on it. This function is not appropriate for small samples with p-hat far from 1/2: 'x' should be at least 5, and so should 'n'-'x.' This function returns three values: p-hat, and the lower and upper bounds of the confidence interval. 'Confidence' should be a number between 0 and 1, exclusive.

The following functions compute the very same values, but don't require all of the data, just summaries of it. There is no need for a summary version of 'confidence-interval-proportion,' since it is already summarized.

`confidence-interval-z-summaries` *mean standard-error confidence* [*Function*]

This function is just like 'confidence-interval-z,' except that instead of its arguments being the actual data, it takes the following summary statistics: 'mean', a point estimator of the mean of some normally distributed population; and the 'standard-error' of the estimator, that is, the estimated standard deviation of the normal population. 'Confidence' should be a number between 0 and 1, exclusive.

`confidence-interval-t-summaries` *mean dof standard-error confidence* [*Function*]

This function is just like 'confidence-interval-t,' except that instead of its arguments being the actual data, it takes the following summary statistics: 'mean,' which is the estimator of some t-distributed parameter; 'dof,' which is the number of degrees of freedom in estimating the mean; and the 'standard-error' of the estimator. In general, 'mean' is a point estimator of the mean of a t-distribution, which may be the slope parameter of a regression, the difference between two means, or other practical t-distributions. 'Confidence' should be a number between 0 and 1, exclusive.

7.2.3 Analysis of Variance

The analysis of variance is something like a t-test on many samples simultaneously. In any event, to pass data to these functions, we need a way to specify many groups of numbers. We have implemented two ways:

1. Structural Groups
   In this representation, each group of data (numbers), is its own sequence. Thus, the one-way analysis of variance using structural grouping takes a sequence of sequences. For example, the following is three groups of four numbers each.

   ```
   data => ((1 6 2 5) (8 3 6 5) (8 9 3 5))
   ```

2. Keyed Groups (Variables)
   In this representation, each data value (number) is associated with a key or keys, and data that share the same keys are in the same group. Functions that take data represented this way have two or three arguments, one of which is called 'dv' and contains *all* the data as a flat sequence of numbers. The other arguments are sequences of keys. Keys can be any Lisp object; they are compared with 'eql.' These functions have `-variables` appended to their names because their representation is the same as that of variables in the database. For example, to specify the same three groups of four numbers, the input would be as follows:

   ```
   iv => (a a a a b b b b c c c c)
   dv => (1 6 2 5 8 3 6 5 8 9 3 5)
   ```

   Of course, it doesn't matter what identifiers are used to distinguish groups.

   The one-way anova function needs its input to be sorted in key order, so that the groups are contiguous, as in our example. The two-way anova function can have its inputs in any order, because it does a lot more processing.

The analysis of variance functions return an ANOVA table as their first value. For the one-way ANOVA, it looks like this:

```
((df-group ss-group ms-group f p)
 (df-error ss-error ms-error)
 (df-total ss-total))
```

For a two-way ANOVA, there is a larger ANOVA table because there are two groupings (a row effect and a column effect) and the interaction of the two. The table looks like this:

```
((df-interaction SS-interaction MS-interaction F-interaction p-interaction)
 (df-row         SS-row         MS-row         F-row         p-row)
 (df-column      SS-column      MS-column      F-column      p-column)
 (df-error       SS-error       MS-error)
 (df-total       SS-total       MS-total))
```

A useful function to print these out is 'print-anova-table'; it can tell which kind of table it is by the length of the list.

`anova-one-way-variables` *iv dv &optional (scheffe-tests-p t)* [*Function*]

Performs a one-way analysis of variance (ANOVA) on the input data, which should be two equal-length sequences: 'iv' is the independent variable, represented as a sequence of categories or group identifiers, and 'dv' is the dependent variable, represented as a sequence of numbers. The 'iv' variable must be "sorted," meaning that AAABBCCCCCDDDD is okay but ABCDABCDABDCDC is not, where A, B, C and D are group identifiers. Furthermore, each group should consist of at least 2 elements. The significance of the result indicates that the group means are not all equal; that is, at least two of the groups have significantly different means. If there were only two groups, this would be semantically equivalent to an unmatched, two-tailed t-test, so you can think of the one-way ANOVA as a multi-group, two-tailed t-test. This function returns four values: 1. an ANOVA table; 2. a list a group means; 3. either a Scheffe table or nil depending on 'scheffe-tests-p'; and 4. an alternate value for SST. The fourth value is only interesting if you think there are numerical accuracy problems; it should be approximately equal to the SST value in the ANOVA table. This function differs from 'anova-one-way-groups' only in its input representation. See the manual for more information.

`anova-one-way-groups` *data &optional (scheffe-tests-p t)* [*Function*]

Performs a one-way analysis of variance (ANOVA) on the 'data,' which should be a sequence of sequences, where each interior sequence is the data for a particular group. Furthermore, each sequence should consist entirely of numbers, and each should have at least 2 elements. The significance of the result indicates that the group means are not all equal; that is, at least two of the groups have significantly different means. If there were only two groups, this would be semantically equivalent to an unmatched, two-tailed t-test, so you can think of the one-way ANOVA as a multi-group, two-tailed t-test. This function returns four values: 1. an ANOVA table; 2. a list a group means; 3. either a Scheffe table or nil depending on 'scheffe-tests-p'; and 4. an alternate value for SST. The fourth value is only interesting if you think there are numerical accuracy problems; it should be approximately equal to the SST value in the ANOVA table. This function differs from 'anova-one-way-variables' only in its input representation. See the manual for more information.

`print-anova-table` *anova-table &optional (stream *standard-output*)* [*Function*]

Prints 'anova-table' on 'stream.'

Optionally, the ANOVA functions can also return scheffe-tables, which represent all pairwise comparisons between groups. A scheffe-table is an upper-triangular table represented with list structure. For example, with four groups, it would look like this:

```
(((F-01 p-01) (F-02 p-02) (F-03 p-03))
 ((F-12 p-12) (F-13 p-13))
 ((F-22 p-23)))
```

This table makes more sense when formatted as follows:

	mean-1	mean-2	mean-3
mean-0:	F-01 p-01	F-02 p-02	F-03 p-03
mean-1:		F-12 p-12	F-13 p-13
mean-2:		F-22 p-22	F-23 p-23

The scheffe-tables can be computed and printed by the following functions.

`scheffe-tests` *group-means group-sizes ms-error df-error* [*Function*]

Performs all pairwise comparisons between group means, testing for significance using Scheffe's F-test. Returns an upper-triangular table in a format described in the manual. Also see the function 'print-scheffe-table.' 'Group-means' and 'group-sizes' should be sequences. The arguments 'ms-error' and 'df-error' are the mean square error within groups and its degrees of freedom, both of which are computed by the analysis of variance. An ANOVA test should always be run first, to see if there are any significant differences.

`print-scheffe-table` *scheffe-table &optional group-means (stream *standard-output*)* [*Function*]

Prints 'scheffe-table' on 'stream.' If the original one-way anova data had N groups, the Scheffe table prints as an n-1 x n-1 upper-triangular table. If 'group-means' is given, it should be a list of the group means, which will be printed along with the table.

Two factor analysis of variance is computed by the following functions.

`anova-two-way-groups` *data-array* [*Function*]

Calculates the analysis of variance when there are two factors that may affect the dependent variable. Because the input is represented as an array, we can refer to these two factors as the row-effect and the column effect. Unlike the one-way ANOVA, there are mathematical difficulties with the two-way ANOVA if there are unequal cell sizes; therefore, we require all cells to be the same size, and so the input is a three-dimensional array. The result of the analysis is an anova-table, as described in the manual. This function differs from 'anova-two-way-variables' only in its input representation. See the manual for further discussion of analysis of variance.

`anova-two-way-variables` *dv iv1 iv2* [*Function*]

Calculates the analysis of variance when there are two factors that may affect the dependent variable, specifically 'iv1' and 'iv2.' Unlike the one-way ANOVA, there are mathematical difficulties with the two-way ANOVA if there are unequal cell sizes; therefore, we require all cells to be the same size; that is, the same number of values (of the dependent variable) for each combination of the independent factors. The result of the analysis is an anova-table, as described in the manual. This function differs from 'anova-two-way-groups' only in its input representation. See the manual for further discussion of analysis of variance. If you use 'print-anova-table,' the row effect is 'iv1' and the column effect is 'iv2.'

7.2.4 Linear Regression

CLASP implements six different linear regression functions, the result of two kinds of input representations (data versus summaries) and three quantities of output (minimal, brief, and verbose). The verbose form returns an anova table in the following format:

```
((df-regression SS-regression MS-regression F p)
 (df-error      SS-error      MS-error)
 (df-total      SS-total))
```

`linear-regression-minimal` *dv iv* [*Function*]

Calculates the slope and intercept of the regression line. This function takes two equal-length sequences of raw data. Note that the dependent variable, as always, comes first in the argument list. You should first look at your data with a scatter plot to see if a linear model is plausible. See the manual for a fuller explanation of linear regression statistics.

`linear-regression-minimal-summaries` *n x y x2 y2 xy* [*Function*]

Calculates the slope and intercept of the regression line. This function differs from 'linear-regression-minimal' in that it takes summary statistics: 'x' and 'y' are the sums of the independent variable and dependent variables, respectively; 'x2' and 'y2' are the sums of the squares of the independent variable and dependent variables, respectively; and 'xy' is the sum of the products of the independent and dependent variables. You should first look at your data with a scatter plot to see if a linear model is plausible. See the manual for a fuller explanation of linear regression statistics.

`linear-regression-brief` *dv iv* [*Function*]

Calculates the main statistics of a linear regression: the slope and intercept of the line, the coefficient of determination, also known as r-square, the standard error of the slope, and the p-value for the regression. This function takes two equal-length sequences of raw data. Note that the dependent variable, as always, comes first in the argument list. You should first look at your data with a scatter plot to see if a linear model is plausible. See the manual for a fuller explanation of linear regression statistics.

`linear-regression-brief-summaries` *n x y x2 y2 xy* [*Function*]

Calculates the main statistics of a linear regression: the slope and intercept of the line, the coefficient of determination, also known as r-square, the standard error of the slope, and the p-value for the regression. This function differs from 'linear-regression-brief' in that it takes summary variables: 'x' and 'y' are the sums of the independent variable and dependent variables, respectively; 'x2' and 'y2' are the sums of the squares of the independent variable and dependent variables, respectively; and 'xy' is the sum of the products of the independent and dependent variables. You should first look at your data with a scatter plot to see if a linear model is plausible. See the manual for a fuller explanation of linear regression statistics.

`linear-regression-verbose` *dv iv* [*Function*]

Calculates almost every statistic of a linear regression: the slope and intercept of the line, the standard error on each, the correlation coefficient, the coefficient of determination, also known as r-square, and an ANOVA table as described in the manual. This function takes two equal-length sequences of raw data. Note that the dependent variable, as always, comes first in the argument list. If you don't need all this information, consider using the "-brief," or "-minimal" functions, which do less computation. You should first look at your data with a scatter plot to see if a linear model is plausible. See the manual for a fuller explanation of linear regression statistics.

`linear-regression-verbose-summaries` *n x y x2 y2 xy* [*Function*]

Calculates almost every statistic of a linear regression: the slope and intercept of the line, the standard error on each, the correlation coefficient, the coefficient of determination, also known as r-square, and an ANOVA table as described in the manual. If you don't need all this information, consider using the "-brief" or "-minimal" functions, which do less computation. This function differs from 'linear-regression-verbose' in that it takes summary variables: 'x' and 'y' are the sums of the independent variable and dependent variables, respectively; 'x2' and 'y2' are the sums of the squares of the independent variable and dependent variables, respectively; and 'xy' is the sum of the products of the independent and dependent variables. You should first look at your data with a scatter plot to see if a linear model is plausible. See the manual for a fuller explanation of linear regression statistics.

The verbose functions do not currently return the standard error of the intercept. This will be fixed in the next release.

7.2.5 Contingency Table Analysis

Often, data collection will simply observe the occurrence of discrete events, rather than measuring some dependent variable. The data, then, consists of counts: n events of type A, m events of type B, and so on. Data tables of counts are called *contingency tables.* Hypotheses can then be tested about the distribution of events (their relative frequency) and whether one kind of event is independent of another. The following functions test such hypotheses.

There are generally two kinds of functions, depending on whether the input representation is a contingency table or sequences of events. In the latter case, the functions construct a contingency table by counting events and then call the other functions.

By convention, the cells in a 2x2 contingency table are labeled as follows:

a	b
c	d

This is the semantics for the arguments of 'chi-square-2x2-counts.' By symmetry, it doesn't matter if you swap the 'b' and 'c' arguments.

`chi-square-2x2-counts` *a b c d &optional (yates t)* [*Function*]

Runs a chi-square test for association on a simple 2 x 2 table. If 'yates' is nil, the correction for continuity is not done; default is t. Returns the chi-square statistic and the significance of the value.

`chi-square-2x2` *v1 v2* [*Function*]

Performs a chi-square test for independence of the two variables, 'v1' and 'v2.' These should be categorial variables with only two values; the function will construct a 2x2 contingency table by counting the number of occurrences of each combination of the variables. See the manual for more details.

`chi-square-rxc-counts` *contingency-table* [*Function*]

Calculates the chi-square statistic and corresponding p-value for the given contingency table. The result says whether the row factor is independent of the column factor. Does not apply Yate's correction.

`chi-square-rxc` *v1 v2* [*Function*]

Performs a chi-square test for independence of the two variables, 'v1' and 'v2.' These should be categorial variables; the function will construct a contingency table by counting the number of occurrences of each combination of the variables. See the manual for more details.

7.2.6 Smoothing

The following functions "smooth" data in a non-parametric way, by replacing elements with a function of themselves and their neighbors. They aren't statistics functions in the usual sense, but they operate on sequences and are useful in processing data before computing statistics. For example, smoothing a time series before doing a cross-correlation can produce a clearer result.

`smooth-median-2` *data* [*Function*]

Smooths 'data' by replacing each element with the median of it and its neighbor on the left. A median of two elements is the same as their mean. The end is handled by duplicating the end element. This function is not destructive; it returns a list the same length as 'data,' which should be a sequence of numbers.

`smooth-median-3` *data* [*Function*]

Smooths 'data' by replacing each element with the median of it and its two neighbors. The ends are handled by duplicating the end elements. This function is not destructive; it returns a list the same length as 'data,' which should be a sequence of numbers.

`smooth-median-4` *data* [*Function*]

Smooths 'data' by replacing each element with the median of it, its left neighbor, and its two right neighbors. The ends are handled by duplicating the end elements. This function is not destructive; it returns a list the same length as 'data,' which should be a sequence of numbers.

`smooth-median-5` *data* [*Function*]

Smooths 'data' by replacing each element with the median of it, its two left neighbors and its two right neighbors. The ends are handled by duplicating the end elements. This function is not destructive; it returns a list the same length as 'data,' which should be a sequence of numbers.

`smooth-hanning` *data* [*Function*]

Smooths 'data' by replacing each element with the weighted mean of it and its two neighbors. The weights are 1/2 for itself and 1/4 for each neighbor. The ends are handled by duplicating the end elements. This function is not destructive; it returns a list the same length as 'data,' which should be a sequence of numbers.

`smooth-4253h` *data* [*Function*]

Smooths 'data' by successive smoothing: 4,median; then 2,median; then 5,median; then 3,median; then hanning. The ends are handled by duplicating the end elements. This function is not destructive; it returns a list the same length as 'data,' which should be a sequence of numbers.

7.3 Data Manipulation Functions

This section describes the representation and functions for CLASP's internal data manipulation code. These functions are called by the graphical interface when data is loaded, manipulated, transformed, saved, and so forth. The functions in this section could allow you to manipulate your own data under program control, or they could be the basis for new manipulation functions.

The basic class definitions and methods are the following:

`data` *[Class]*

'data' is the basic data class for clasp. Dataset, variable etc. are all subclasses of data.

`name` *object* *[Method]*

data 'name' returns the name of a data object.

`description` *object* *[Method]*

data 'description' returns the description of a data object.

`id` *object* *[Method]*

data 'id' returns the internal id of a data object.

`dataset` *[Class]*

The 'dataset' is the top-level user class for data representation in CLASP. A dataset is a collection of variables.

`variable` *[Class]*

'variable's are ordered collections of values. Valid data types for variables are numbers (integers, rationals and floats), symbols and strings. Symbols and strings can be used to represent for categorical data).

7.3.1 Dataset Functions

With the following functions, we start creating and manipulating data, variables, and datasets. These are the fairly low-level functions for creating, accessing and destroying the data structures.

`variable-value` *the-variable &key order-by where where-args distinct* *[Function]*

The `variable-value` function returns the value of a variable, expressed as a list. The `:order-by` keyword can be used to specify another variable to sort on. The `:order-by` variable must be in the same dataset as *the-variable*. By default, the row-order of the dataset is used.

`make-dataset` *&key (name nil) (description nil)* *[Function]*

Creates a new dataset and fills the slots with 'name', 'data', 'variable-list' and 'description'.

`make-dataset-from-rows` *name data column-names &key description* [*Function*]

Creates a new data set and fills the slots with 'name', 'data' and 'description'. 'data' is a list of lists where each interior list represents one row of the data.

`make-dataset-from-columns` *name data column-names &key description* [*Function*]

Creates a new data set and fills the slots with 'name', 'data' and 'description'. 'data' is a list of lists where each interior list represents one column of the data.

`rename-dataset` *dataset new-name* [*Function*]

rename-dataset changes the name of 'dataset to 'new-name', insuring the new name is unique. 'dataset' can either be a dataset or the name of a dataset, and 'new-name' must be a string.

`delete-dataset` *dataset* [*Function*]

Destroys a dataset and removes it from the dataset lists

`get-dataset` *the-dataset &optional (active-datasets-only-p t) (not-found-error-p t)* [*Function*]

Given a name or id, finds the dataset. If 'active-datasets-only-p' is nil, will search through all current clasp datasets, otherwise, only the active datasets are searched.

`activate-dataset` *dataset* [*Function*]

Takes a dataset on the available list *clasp-datasets* and adds it to the active list *clasp-active-datasets*

`deactivate-dataset` *dataset* [*Function*]

Takes a dataset off of the *clasp-active-datasets* list.

`make-variable` *&key name type value description dataset* [*Function*]

make-variable creates a new variable. 'name' is a string. If 'type' is one of :number, :symbol or :string, then that will be used as the type, otherwise the type will be inferred from (type-of 'type'). 'value' is the data the variable should be initialized with. 'description' is a description of the variable. 'dataset' is the dataset to which the variable belongs.

`add-variable-to-dataset` *dataset data variable-name &optional (description nil)* [*Function*]

add-variable-to-dataset adds the 'data', which must be a list of values, to 'dataset' using 'variable-name' as the name of the new variable and 'description' as a description.

`add-variables-to-dataset` *dataset data-list variable-names &optional (descriptions nil)* [*Function*]

add-variables-to-dataset adds a number of variables to 'dataset'. For each variable, i, its value (which must be a list) is the ith element of 'data-list', its name is the ith element of 'variable-names', and its description is the ith element of descriptions.

`rename-variable` *variable new-name* [*Function*]

rename-variable changes the name of 'variable' to 'new-name', insure the new name is unique. 'variable' can either be a variable or the name of a variable, and 'new-name' must be a string.

`delete-variable` *variable* [*Function*]

Destroys a variable. 'variable' can either be a variable or the name of a variable.

`get-variable` *the-variable &optional dataset (active-datasets-only-p t) (not-found-error-p t)* [*Function*]

Given a name or id, finds the variable. If 'dataset' is specified, that is the dataset which will be searched for 'variable'. Otherwise, a search will be made across datasets. If this is the case, then 'active-datasets-only-p' will determine whether all datasets currently loaded are searched, or just the active ones. Note, since variable names must only be unique within a dataset, not across datasets, care must be taken to request the correct variable. It is advisable to call get-variable with the dataset optional variable set.

`add-rows-to-dataset` *dataset data-list variables* [*Function*]

This function will add 'data-list' to the dataset 'dataset'. 'data-list' is a list of lists representing rows of data. The rows must be of equal length. 'variables' is a list of variables and specifies what order the new rows are in with respect to the variables in the dataset. Therefore, each element of 'data-list' must match variables in both length and the types of its elements.

7.3.2 Input/Output

The following functions do input and output of data, so that they can be saved between sessions and processed by other programs. CLASP supports two ways of representing datasets in files: a special CLASP format and a general non-CLASP format that should be readable by most other statistics programs.

The CLASP file format is read by 'load-dataset' and saved by 'save-dataset.' The loading function takes a filename as its only argument, while the saving function takes a dataset and a filename. This format is also produced by CLIP, because we want CLIP and CLASP to have a seamless interface. The CLASP format is as follows:*

- The first line of the file is a string (surrounded by double-quotes) containing the name of the dataset.
- The following n lines of the file each have a single string with the name of one of the n variables in the dataset.
- At any point after the variable names, comments may be inserted by starting a line with a semicolon (;).
- The rest of the lines of the file each contain a Common Lisp list representing one row of data. The order of the elements must match the order of the variable names listed at the top of the file.

The non-CLASP file format is read by 'import-dataset' and saved by 'export-dataset.' They take extra arguments because the file representation allows user-specified separator characters between data values. Some statistics packages use commas between values, others use tabs or spaces. The separator character is specified by `:separator`, defaulting to a

*For examples of CLASP format, see Section 2.4 and Appendix B.

comma (`#\,`). Furthermore, the file representation allows, optionally, the names of the variables to appear in the file. You can allow them to appear using `:include-labels-p`, which defaults to nil. The non-CLASP format is as follows:

- The first line is, optionally, a line of variable names, expressed as symbols, and separated by separator characters.
- The rest of the file is lines of data values, separated by separator characters.

Both 'load-dataset' and 'import-dataset' will signal an error if 'filename' does not specify an extant file. No other error checking is currently done, and so, for example, if some of the data rows have too few elements, as in the case of missing data, the dataset will be incorrect. In addition, the datatype of each variable will be inferred from its value in the first row of the data.

Both 'save-dataset' and 'export-dataset' will signal an error if 'filename' specifies an already extant file.

`load-dataset` *filename* [*Function*]

> Reads a dataset in from a file in standard CLASP format. Specifically, the first line is a string containing the name of the dataset. The following n lines are strings containing the names of the variables in the dataset, and the following m lines are lists containing the rows of the dataset.

`save-dataset` *data filename* [*Function*]

> Writes a dataset out to a file in CLASP format.

`import-dataset` *filename &key (separator ,) include-labels-p* [*Function*]

> Reads a dataset in from a file, where 'filename' is a string or pathname. If 'filename' doesn't exist, an error is signaled. The data on each line in the file must be separated by 'separator' characters or whitespace. No error checking is done to insure that each line of the file contains the same number of data. If 'include-labels-p' is non-nil, the first line will be used a variable names, otherwise they will be named variable, variable-1, variable-2, etc. The filename will be used for the dataset name.

`export-dataset` *dataset filename &key (separator #\,) include-labels-p* [*Function*]

> Writes out the values in 'dataset' in row-major order to the file 'filename.' Each row in 'filename' will be one row of data in 'dataset.' The data in a row will be separated by the character 'separator'. If include-labels-p is non-nil, the variable-names will be written out on the first line. 'export-dataset' will signal an error if a file of name 'filename' already exists.

7.3.3 Manipulation

Finally, we get to functions that transform or otherwise manipulate existing data.

`create-new-column-from-expression` *dataset expression* [*Function*]

> create-new-column-from-expression adds a new variable to 'dataset', using 'expression' to calculate the values. 'expression' may be any numerical expression. If variables are included in 'expression', they should be referenced by name. For instance, if a dataset has two variables, "hits" and "misses", then

```
(create-new-column-from-expression dataset
                                    (/ hits (+ hits misses)))
```

will add a new variable with the hit-to-attempt ratio.

When create-new-column-from-expression is called, the rows of 'dataset' are mapped over, and for each row, substitutions are made into 'expression' for any variable references. The value returned is added to the new variable in the appropriate row of 'dataset'. Because the dataset is mapped over, aggregating functions (such as mean or data-length) cannot be used. A new function, 'recode', has been implemented to allow for mapping of categorical variables to new values.

`create-new-column-from-function` *dataset new-name function &rest arg-variables* *[Function]*

create-new-column-from-function adds a new variable to 'dataset', using 'function' to calculate the values. 'function' must be a valid lisp function, while 'arg-variables' should be a list of variables that are used to create the arguments to the function. The rows of 'dataset' are mapped over, calling 'function' with values extracted from each of the 'arg-variables' which are then. This produces a new column of data which is added as a variable to 'dataset'.

`recode` *value old-values new-values* *[Function]*

Maps 'value' from it's position in 'original-values' to the value at the equivalent position in 'new-values'. For instance, (recode 2 '(1 2 3) '(a b c)) would return 'b. This function can be used with create-new-column-from-expression to recode a categorical variable. From the program interface, it is recommended that "recode-list" be used instead.

`recode-list` *list old-values new-values* *[Function]*

Maps each element of 'list' from it's position in 'old-values' to the value at the equivalent position in 'new-values'.

Partitioning

Often it is desirable to operate on a portion of a dataset, or to separate out different parts of a dataset according to the values of some of the variables. When a dataset is broken up this way, it is called partitioning, and the parts of the dataset created are called partitions. There are two functions used for partitioning in CLASP, partition-dataset and partition-on.

`partition-dataset` *dataset filter arg-variables &optional include-variables* *[Function]*

partition-dataset produces a new dataset whose rows pass the filter. `:filter` should be a function that accepts values extract from a row of `:arg-variables`. 'include-variables' indicates what variables to include in the new dataset.

`partition-on` *dataset variables &optional include-variables* *[Function]*

partition-on separates the dataset into different partitions for each unique set of values of its variables. 'include-variables' indicates what variables to include in the new dataset.

`dataset-to-rows` *data &key columns filter arg-variables* *[Function]*

Takes a dataset *data* and extracts tuples representing the rows of data in the dataset. The `:columns` keyword takes a list of variables (which must all be in the dataset *data*.) By default, all variables are extracted. The `:filter` and `:arg-variables` are the same as for partition-dataset (see Section 7.3.3).

`dataset-to-columns` *data &key columns filter arg-variables* [*Function*]

Takes a dataset 'data' and extracts tuples representing the columns of data in the dataset. The `:filter` and `:arg-variables` are the same as for partition-dataset (see Section 7.3.3).

`transpose` *data* [*Function*]

Takes a list of lists, 'data', and makes a list of tuples t where t[i] contains all the i'th elements of the lists in 'data'.

7.4 Density and Combinatorial Functions

Given a statistic, such as the t-statistic, the final step in hypothesis testing is to compute the p-value of the statistic—that is, the probability that a value for the statistic as extreme or more so could occur by chance. In statistics books, p-values are found by looking in tables in the appendices, but our computer programs can't do that. Instead, they must calculate the p-value.

The following functions are how CLASP calculates p-values. Some of them may be completely useless to you unless you are implementing new statistical functions. (If you do, let us know; perhaps they can be incorporated into the next release.) Some are useful if you want to graph theoretical distributions or otherwise experiment with them. Finally, there are some functions that calculate combinatorial probabilities, such as the binomial.

Most of the following functions are implemented from *Numerical Recipes in C* [11]. The code has been extensively tested and the values compared to tables of statistics or, in some cases, to figures in the *Numerical Recipes* book.

`beta` *z w* [*Function*]

Returns the value of the Beta function, defined in terms of the complete gamma function, G, as: G(z)G(w)/G(z+w). The implementation follows Numerical Recipes in C, section 6.1.

The complete Beta function is not used in CLASP, but may be helpful to some users.

`gamma-incomplete` *a x* [*Function*]

This is an incomplete gamma function, what *Numerical Recipes in C* calls "gammp." This function also returns, as the second value, g(a,x). See the manual for more information.

The incomplete gamma function, notated p(a,x) is mathematically defined as

$$p(a, x) = g(a, x)/G(a)$$

where $g(a, x)$ is what Mathematica$^{\mathrm{tm}}$ calls the incomplete gamma function and is mathematically defined as

$$\int_0^x t^{a-1} e^{-t} dt$$

and G is the complete gamma function, defined as

$$\int_0^\infty t^{a-1} e^{-t} dt$$

Yes, it's very confusing; apparently, there is no standard naming scheme for these functions. Our implementation follows *Numerical Recipes in C*, section 6.2 and is the function that they call "gammp." We did not implement the function they call "gammq," since it is just 1.0-gammp.

`error-function` x *[Function]*

> Computes the error function, which is typically used to compute areas under the Gaussian probability distribution. See the manual for more information. Also see the function 'gaussian-cdf.' This implementation follows *Numerical Recipes in C*, section 6.2

The error function is not well described in *Numerical Recipes* or other books that we checked, and it mystified us for some time. Once we understood it, it made perfect sense, so we offer the following explanation.

The error function is defined as

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\, dt$$

The error function is related to the integral of the Gaussian probability density function (pdf), as we can see in the following proof. First, define $A(x)$ to be the area under the Gaussian pdf and less than x in absolute value. (This is the complement of a two-tailed test of the significance of x.)

$$A(x) = \int_{-x}^{x} \frac{1}{\sqrt{2\pi}} e^{-t^2/2} dt$$

Because the Gaussian is symmetrical, if $x > 0$, we can just integrate from 0 to x and double it:

$$A(x) = 2\int_0^x \frac{1}{\sqrt{2\pi}} e^{-t^2/2} dt$$

Now we can show that A is closely related to erf. The proof works by a change of variable, substituting $t = u\sqrt{2}$. All the rest is just algebra.

$$\begin{aligned}
A(x) &= 2\int_0^x \frac{1}{\sqrt{2\pi}} e^{-t^2/2} dt \\
&= 2\int_0^{x/\sqrt{2}} \frac{1}{\sqrt{2\pi}} e^{-(u\sqrt{2})^2/2} du\sqrt{2} \\
&= 2\int_0^{x/\sqrt{2}} \frac{1}{\sqrt{\pi}} e^{-u^2} du \\
&= \frac{2}{\sqrt{\pi}} \int_0^{x/\sqrt{2}} e^{-u^2} du \\
&= \mathrm{erf}(x/\sqrt{2})
\end{aligned}$$

That change of variable is the key to understanding erf: erf is the integral (from zero outwards) of a function that is shaped exactly like the Gaussian, except that the axis has been "stretched" by a factor of $\sqrt{2}$. So, to find $A(x)$, we just compute $\mathrm{erf}(x/\sqrt{2})$.

Note, by defining erf to be the integral from 0 to x, we have $\mathrm{erf}(-x) = -\mathrm{erf}(x)$. With this in mind, we can compute other integrals of the Gaussian using erf. Define F(x) to be the cumulative distribution function of the Gaussian. Then

$$F(x) = \frac{1}{2}(1 + \mathrm{erf}(x/\sqrt{2}))$$

Indeed, erf can be used to compute the cdf of Gaussians with mean m and standard deviation s by standardizing and dividing by the square root of 2.

`gaussian-cdf` *x &optional (mean 0.0) (sd 1.0)* [*Function*]

> Computes the cumulative distribution function for a Gaussian random variable (defaults: mean=0.0, s.d.=1.0) evaluated at 'x.' The result is the probability of getting a random number less than or equal to 'x,' from the given Gaussian distribution.

`error-function-complement` *x* [*Function*]

> This function computes the complement of the error function, "erfc(x)," defined as 1-erf(x). See the documentation for 'error-function' for a more complete definition and description. Essentially, this function on z/sqrt2 returns the two-tailed significance of z in a standard Gaussian distribution. This function implements the function that *Numerical Recipes in C* calls erfcc, see section 6.3; that is, it's the one using the Chebyshev approximation, since that is the one they call from their statistical functions. It is quick to compute and has fractional error everywhere less than 1.2x10^{-7}.

`gaussian-significance` *x tails &optional mean sd* [*Function*]

> Computes the significance of 'x' in a Gaussian distribution with mean='mean' (default 0.0) and standard deviation='sd' (default 1.0); that is, it returns the area which farther from the mean than 'x' is. The null hypothesis is roughly that 'x' is zero; you must specify your alternative hypothesis (H1) via the 'tails' parameter, which must be :both, :positive or :negative. The first corresponds to a two-tailed test: H1 is that 'x' is not zero, but you are not specifying a direction. If the parameter is :positive, H1 is that 'x' is positive, and similarly for :negative.

Departing from the Gaussian distribution, we can compute the significance of other kinds of statistics.

`poisson-cdf` *k x* [*Function*]

> Computes the cumulative distribution function for a Poisson random variable with mean 'x' evaluated at 'k.' The result is the probability that the number of Poisson random events occurring will be between 0 and k-1 inclusive, if the expected number is 'x.' The argument 'k' should be an integer, while 'x' should be a float. The implementation follows *Numerical Recipes in C*, section 6.2

`chi-square-significance` *x dof* [*Function*]

> Computes the complement of the cumulative distribution function for a Chi-square random variable with 'dof' degrees of freedom evaluated at 'x.' The result is the probability that the observed chi-square for a correct model should be greater than 'x.' The implementation follows *Numerical Recipes in C*, section 6.2. Small values suggest that the null hypothesis should be rejected; in other words, this computes the significance of 'x.'

The incomplete "beta" function is not particularly useful by itself, but is useful in defining the cumulative distributions for Student's t and the F distribution. It is mathematically defined as

$$I_x(a,b) = \frac{1}{B(a,b)} \int_0^x t^{a-1}(1-t)^{b-1}dt$$

where $B(a, b)$ is the complete "beta" function.

`beta-incomplete` *a b x* [*Function*]

This function is useful in defining the cumulative distributions for Student's t and the F distribution. All arguments must be floating-point numbers; 'a' and 'b' must be positive and 'x' must be between 0.0 and 1.0, inclusive.

`students-t-significance` *t-statistic dof tails* [*Function*]

Student's distribution is much like the Gaussian distribution except with heavier tails, depending on the number of degrees of freedom, 'dof.' As 'dof' goes to infinity, Student's distribution approaches the Gaussian. This function computes the significance of 't-statistic.' Values range from 0.0 to 1.0: small values suggest that the null hypothesis—that 't-statistic' is drawn from a t distribution—should be rejected. The 't-statistic' parameter should be a float, while 'dof' should be an integer. The null hypothesis is roughly that 't-statistic' is zero; you must specify your alternative hypothesis (H1) via the 'tails' parameter, which must be :both, :positive or :negative. The first corresponds to a two-tailed test: H1 is that 't-statistic' is not zero, but you are not specifying a direction. If the parameter is :positive, H1 is that 't-statistic' is positive, and similarly for :negative. This implementation follows *Numerical Recipes in C*, section 6.3.

`f-significance` *f-statistic numerator-dof denominator-dof &optional one-tailed-p* [*Function*]

This function occurs in the statistical test of whether two observed samples have the same variance. A certain statistic, F, essentially the ratio of the observed dispersion of the first sample to that of the second one, is calculated. This function computes the tail areas of the null hypothesis: that the variances of the numerator and denominator are equal. It can be used for either a one-tailed or two-tailed test. The default is two-tailed, but one-tailed can be computed by setting the optional argument 'one-tailed-p' to true. For a two-tailed test, this function computes the probability that F would be as different from 1.0 (larger or smaller) as it is, if the null hypothesis is true. For a one-tailed test, this function computes the probability that F would be as LARGE as it is if the first sample's underlying distribution actually has SMALLER variance that the second's, where 'numerator-dof' and 'denominator-dof' is the number of degrees of freedom in the numerator sample and the denominator sample. In other words, this computes the significance level at which the hypothesis "the numerator sample has smaller variance than the denominator sample" can be rejected. A small numerical value implies a very significant rejection. The 'f-statistic' must be a non-negative floating-point number. The degrees of freedom arguments must be positive integers. The 'one-tailed-p' argument is treated as a boolean. This implementation follows *Numerical Recipes in C*, section 6.3 and the 'ftest' function in section 13.4. Some of the documentation is also drawn from the section 6.3, since I couldn't improve on their explanation.

Some useful combinatorial functions are the following. The 'gamma-ln' function is the natural logarithm of the gamma function, which is the continuous form of the factorial function.

`gamma-ln` *x* [*Function*]

Returns the natural logarithm of the Gamma function evaluated at 'x.' Mathematically, the Gamma function is defined to be the integral from 0 to Infinity of t^x exp(-t) dt. The implementation is copied, with extensions for the reflection formula, from *Numerical Recipes in C*, section 6.1. The argument 'x' must be positive. Full accuracy is obtained for x>1. For x<1, the reflection formula is used. The computation is done using double-floats, and the result is a double-float.

`factorial-exact` *n* [*Function*]

Returns the factorial of 'n,' which should be an integer. The result will returned as an integer or bignum. This implementation is exact, but is more computationally expensive than 'factorial,' which is to be preferred.

`factorial` *n* [*Function*]

Returns the factorial of 'n,' which should be a non-negative integer. The result will returned as a floating-point number, single-float if possible, otherwise double-float. If it is returned as a double-float, it won't necessarily be integral, since the actual computation is (exp (gamma-ln (1+ n))) Implementation is loosely based on *Numerical Recipes in C*, section 6.1. On the TI Explorer, the largest argument that won't cause a floating overflow is 170.

`factorial-ln` *n* [*Function*]

Returns the natural logarithm of n!; 'n' should be an integer. The result will be a single-precision, floating point number. The implementation follows *Numerical Recipes in C*, section 6.1

`binomial-coefficient` *n k* [*Function*]

Returns the binomial coefficient, 'n' choose 'k,' as an integer. The result may not be exactly correct, since the computation is done with logarithms. The result is rounded to an integer. The implementation follows *Numerical Recipes* in C, section 6.1

`binomial-coefficient-exact` *n k* [*Function*]

This is an exact but computationally intensive form of the preferred function, 'binomial-coefficient.'

`binomial-probability` *p n k* [*Function*]

Returns the probability of 'k' successes in 'n' trials, where at each trial the probability of success is 'p.' This function uses floating-point approximations, and so is computationally efficient but not necessarily exact.

`binomial-probability-exact` *p n k* [*Function*]

This is an exact but computationally intensive form of the preferred function, 'binomial-probability.'

`binomial-cdf` *p n k* [*Function*]

Suppose an event occurs with probability 'p' per trial. This function computes the probability of 'k' or more events occurring in 'n' trials. Note that this is the complement of the usual definition of cdf. This function approximates the actual computation using the incomplete beta function, but is preferable for large 'n' (greater than a dozen or so) because it avoids summing many tiny floating-point numbers. The implementation follows *Numerical Recipes in C*, section 6.3.

Distribution	Slots	Legal value
Uniform-distribution	Minimum	Real number
	Maximum	Real number
Normal-distribution	Mean	Real number
	Standard-deviation	Non-negative real number
Binomial-distribution	P	Real number from 0 to 1
	N	Positive integer
Poisson-distribution	Mean	Real number
Gamma-distribution	IA	Positive real number

Table 7.1: Distribution classes and parameters

`binomial-cdf-exact` *p n k* [*Function*]

This is an exact but computationally intensive form of the preferred function, 'binomial-cdf.'

The following function uses binary search to find a critical value of a statistic, given a distribution function for the statistic.

`find-critical-value` *p-function p-value &optional (x-tolerance 1.0e-5) (y-tolerance 1.0e-5)* [*Function*]

Returns the critical value of some statistic. The function 'p-function' should be a unary function mapping statistics—x values—to their significance—p values. The function will find the value of x such that the p-value is 'p-value.' The function works by binary search. A secant method might be better, but this seems to be acceptably fast. Only positive values of x are considered, and 'p-function' should be monotonically decreasing from its value at x=0. The binary search ends when either the function value is within 'y-tolerance' of 'p-value' or the size of the search region shrinks to less than 'x-tolerance.'

7.5 Sampling

CLASP provides random sampling from uniform, normal, binomial, poisson and gamma distributions. Distributions are CLOS classes . The distribution classes are defined as follows:

Sampling from a distribution is a two step process. The first step is to create the distribution, and the second step is to draw a sample from the distribution. To create a distribution, use `make-instance`. To draw a sample from a distribution use either `sample-to-list` or `sample-to-dataset`.

The following example shows how to create a standard-normal distribution and then make a dataset from two samples of length 50 drawn from the distribution.

```
;;; create a standard normal distribution
(setf dist (make-instance normal-distribution :mean 0
                                              :standard-deviation 1))
```

```
;;; draw two 50 element samples from the distribution and create a
;;; dataset from the result
(sample-to-dataset dist 50 2)
```

`sample-to-list` *((dist distribution) size-of-sample)* [*Method*]

Gets a sample from the distribution 'dist' of size 'size-of-sample'

`sample-to-dataset` *((dist distribution) size-of-samples &optional number-of-samples dataset-name variable-names* [*Method*]

Gets a sample from the distribution 'dist' of size 'size-of-sample' and creates a dataset out of it. If 'number-of-samples' is > 1 then multiple samples are taken from the distribution and each one forms one column of the dataset.

7.6 Graphics

All graphics in the CLIM CLASP are obtained via the SCIGRAPH graphing system from BBN. Please refer to the documentation available with that system to use graphical functions in CLASP. SCIGRAPH is available via anonymous ftp from *cambridge.apple.com* in the directory /pub/clim/clim-1-and-2/scigraph.

All graphics in Macintosh CLASP are obtained via the EKSL plotter graphing system from EKSL.

Appendix A

Computational Methods

Essentially, this chapter acts as extended documentation for some of the more complicated statistical functions. Statistical functions, by their very nature, take collections of numbers, do some calculations that are often mysterious and unintuitive, and produce some other numbers that are difficult to check by hand. Consequently, these functions are hard for programmers to debug and hard for users to trust. This chapter will try to explain the calculations for some of the more arcane functions.

A.1 Quantiles

Quantiles seem like they ought to be simple, since they are just a generalization of the median, which splits the data in half. All we need to do is split the data into q and $1-q$. But there are other valid ways of viewing the quantile, which lead to other ways of dividing the data. As Freund says in his discussion of quartiles, "there is ample room for arbitrariness" [5, p. 52].

x_0	x_1	x_2	x_3	x_4
2	3	5	8	13

Consider the preceding five-element sample, which we will use throughout our discussion of quantiles. Clearly, x_2 is the median. What is x_1? There are three reasonable answers:

- **First Quartile** x_1 splits the sample so that three times as many elements are above it as below it; splitting the sample in a ratio of 1:3 is one definition of the first quartile.
- **First Quintile** One fifth of the sample is less than x_1, and so it is the first quintile.
- **Second Quintile** Two fifths of the sample is less than or equal to x_1, and so it is the second quintile.

There are two kinds of definitions here, resulting in different computational methods. The first is what we will call the "bisection" method because we look at how the data are split, and the second is the "cumulative" method, because we refer to the cumulative distribution function of the data. Of course, in the cumulative method, we also have to think about whether we want "$<$" or "$\leq$." These methods will be discussed in turn. Only the bisection method is implemented in CLASP, so some readers may want to read only subsection A.1.1; readers who want to understand why we chose the bisection method over the cumulative method will want to read this whole section.

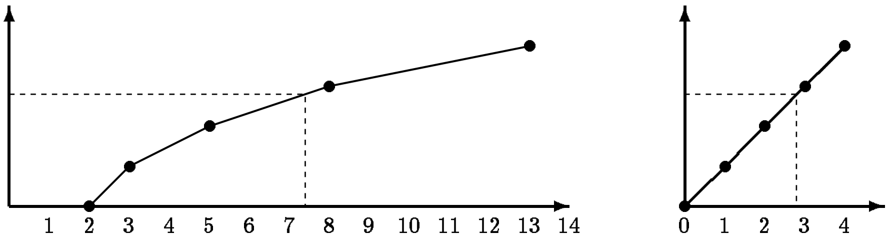

Figure A.1: In these graphs, the vertical axis is q, the quantile, and the horizontal axis is either x (at left) or i (at right). The bisection method maps q directly onto $i \in [0, n-1]$, as shown at right. The graph at the left shows how q maps onto the data values, x_i.

In either case, the way to think about the quantile is as a function from the quantile $q \in [0,1]$ to the indices of the data, $i \in [0, n-1]$, where n is the number of sample items. This makes sense because the actual data values have little influence on the quantile; their relative ranks matter more. Their magnitudes will matter when the quantile falls between two values and we interpolate.

A.1.1 Bisection

The bisection method is very easy, because it maps q directly onto i in the obvious manner, $i = q(n-1)$, as shown in Figure A.1. The median conforms easily to this definition: $0.5(5-1) = 2$, and x_2 is the median of our example. Similarly, $0.25(5-1) = 1$ and x_1 is the first quartile. Using the bisection method, $q = 0.0$ is the smallest element in the sample and $q = 1.0$ is the largest.

What about when i is not an integer? For example, suppose $q = 0.7$, so the element "index" is 2.8, as illustrated in Figure A.1. Letting $i = \lfloor q(n-1) \rfloor$, we can linearly interpolate between the two bracketing values, x_i and x_{i+1}, which have exact quantiles of $q_i = i/(n-1)$ and $q_{i+1} = (i+1)/(n-1)$. The linear interpolation is a weighted sum of x_i and x_{i+1} where the weights are such that q is the same weighted sum of q_i and q_{i+1}. Let the weights be λ and $(1-\lambda)$ for elements $i+1$ and i, respectively.

$$\begin{aligned}
q &= \lambda q_{i+1} + (1-\lambda) q_i \\
q &= \lambda \left(\frac{i+1}{n-1}\right) + (1-\lambda)\left(\frac{i}{n-1}\right) \\
q(n-1) &= \lambda(i+1) + (1-\lambda) i \\
q(n-1) &= \lambda + i \\
\lambda &= q(n-1) - i \\
\lambda &= q(n-1) - \lfloor q(n-1) \rfloor
\end{aligned}$$

In other words, λ is the fractional part left over after we compute the floor of $q(n-1)$, so we get it for free in our calculation of i. We then compute:

$$x_{q(n-1)} = \lambda x_{i+1} + (1-\lambda) x_i$$

In our example with $q = 0.7$, $\lambda = 2.8 - \lfloor 2.8 \rfloor$, and $x_{2.8} = (0.2)5 + (0.8)8 = 7.4$.

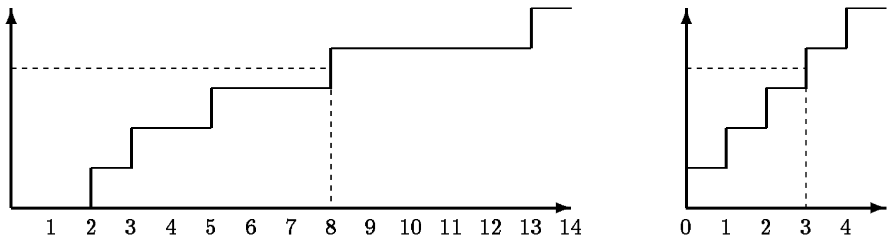

Figure A.2: Inverting the discrete CDF for our sample. In both graphs, each vertical step is $1/n$, since each element is that fraction of the whole. On the left, we map directly to the x_i, while on the right we just map to i, which has a very easy structure, since each horizontal step is of unit length.

A.1.2 Cumulative

The cumulative distribution function (CDF) of a random variable $\mathbf{X}$ is defined as $F(x) = \Pr(\mathbf{X} \leq x)$. As such, the CDF maps from the domain of the random variable to the interval $[0, 1]$. The quantile function is simply the inverse of the CDF! For example, we know that if $\mathbf{X}$ has a standard normal distribution, $\Pr(\mathbf{X} < -1.96) = 0.025$, so -1.96 is the $q = 0.025$ quantile of the standard normal.

Figure A.2 shows two cumulative distribution functions for our sample. They are step functions because our sample is finite, and each element adds $1/n$ to the running total. The function on the left has the x_i as its domain, while the function on the right has i as its domain. The idea is to invert this function, so start at q on the vertical axis, go right until we hit the function, and then go down until we hit the horizontal axis, just as we did in Figure A.1. In the graph on the right, we hit i such that:

$$\begin{array}{ccccc} i/n & < & q & < & (i+1)/n \\ i & < & nq & < & (i+1) \end{array}$$

So, $i = \lfloor qn \rfloor$. Compare this with the bisection method, which gives $i = \lfloor q(n-1) \rfloor$; the algorithmic difference is obvious, but the semantic difference is subtle.

Note that the cumulative method is the inverse of the bisection method with respect to non-integer values of i: with bisection, we knew what to do when $q(n-1)$ was an integer, and we had to resolve what to do when it is not, while with the cumulative method, we have not resolved what to do when qn is an integer. When qn is an integer, the horizontal line hits the top of one vertical segment and the bottom of the next vertical segment. To which vertical segment does q belong? This question is identical to whether we define quantiles in terms of "$<$" or "$\leq$." In other words, is $i/n < q$ or is $i/n \leq q$? This is a fairly arbitrary decision, but we can reduce its effects by interpolating between values.

Suppose we want to interpolate with the cumulative method. Remember that, currently, the cumulative method always returns one of the elements of the sample. Indeed, this is a major flaw of the cumulative method, because the cumulative method cannot yet compute the median when n is even. In our example, $n = 5$, so it computes the median as x_i where $i = \lfloor qn \rfloor = 2$, which is correct. But if $n = 4$, the median is defined to be the average of the middle two elements and is not equal to any element of the sample.

Linear interpolation in the cumulative method can be visualized by connecting the vertical lines in our cumulative distribution function (see Figure A.3). Unfortunately, because

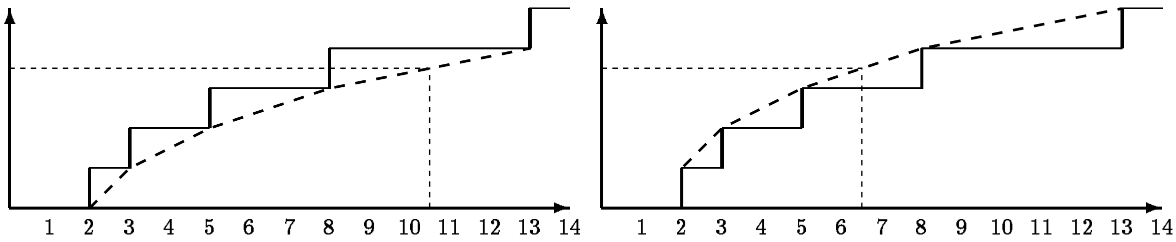

Figure A.3: The ways of interpolating in the cumulative method: either connecting the bottoms of the vertical segments (left) or the tops (right).

of the "$<$" versus "$\leq$" problem, it can be done in two different ways, either connecting the bottoms of the vertical lines ("$<$") or the tops ("$\leq$"). Once we find the right interval, the interpolation is done the same as in the bisection method, computing the weight λ from the fractional part of $i = qn$. In our $q = 0.7$ example, the interpolated value is either 10.5 (halfway between the 0.8 quantile, $x_3 = 8$, and the 1.0 quantile, $x_4 = 13$) or 6.5 (halfway between the 0.8 quantile, $x_2 = 5$, and the 0.8 quantile, $x_3 = 8$), as is shown in Figure A.3. Contrast these values with the 7.4 obtained with the bisection method.

Note in figure A.3 that when we connect the bottoms of the segments, the last vertical segment (over $x_4 = 13$) belongs to the CDF, while when we connect the tops, the first vertical segment (over $x_0 = 2$) belongs to the CDF. This produces an essential asymmetry in each curve. Most elements influence two $n$ths of the curve (the interval on either side), but the extremes are different. One extreme influences only one interval, while the other extreme influences one interval and completely determines another. For example, when connecting the bottoms of the verticals, it seems that $x_4 = 13$ has more than its fair share of influence. One solution to this asymmetry is to count the quantile from the nearer end. This amounts to connecting the tops of the segments for $q < 0.5$ and the bottoms of the segments for $q > 0.5$ or vice versa. Thus either both extreme elements get undue influence (relative to middle elements) or neither does.

A.1.3 Discussion

We have presented two ways of viewing the quantile. One views the quantile as bisecting a line from 0 to $n-1$, with data elements sitting on the line at the integer positions. The other views the quantile with reference to the CDF and leads to unending problems all stemming from how to treat the vertical steps in that CDF.

One problem that still remains with the cumulative method is how to find the median. In our example, element $x_2 = 5$ is either at quantile $q = .4$ or $q = .6$, and so it cannot be the median. A solution can certainly be found, but it doesn't fall easily out of the cumulative method.

For these reasons, we have chosen to implement the bisection method as simpler and more elegant than any version of the cumulative method. Figure A.4 shows all three functions in one large graph; you can see how the three methods will disagree on every quantile (except the extremes), and sometimes even interpolate in different intervals. Nevertheless, the disagreements will be fairly small and will shrink as the samples get larger.

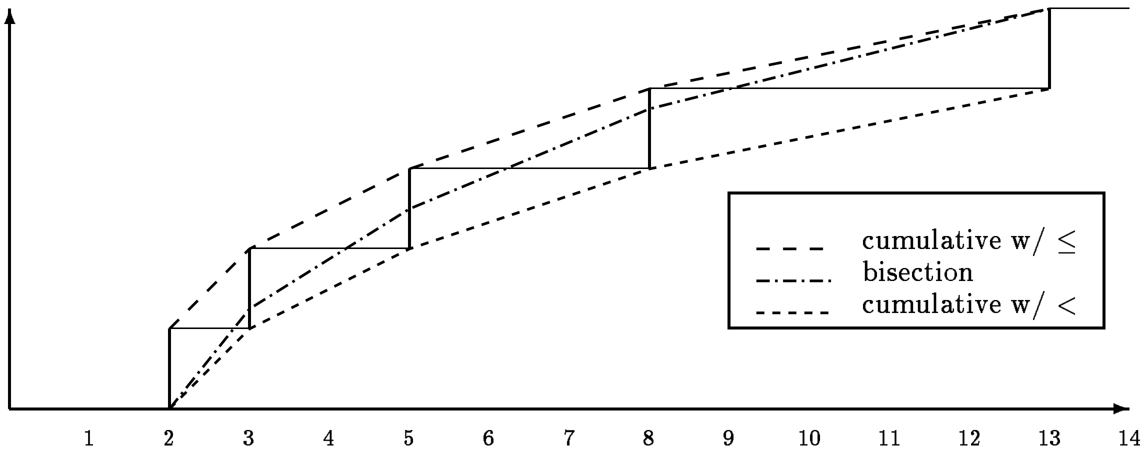

Figure A.4: A graphical comparison of the bisection method (middle line) the cumulative method with "<" (bottom line) and the cumulative method with "≤" (top line).

A.2 Analysis of Variance

The heart of the analysis of variance is, of course, calculating different kinds of variance in the data. Calculations of sums-of-squares (variance is just a sum-of-squares divided by the number of degrees of freedom) can generally be done in two different ways:

$$\sum_{i=1}^{n}(x_i - \bar{x})^2 = \sum_{i=1}^{n} x_i^2 - 1/n \left(\sum_{i=1}^{n} x_i\right)^2$$

These expressions are algebraically equivalent, as can easily be shown. We will do the proof carefully once, so that when we talk about different kinds of variance below, we can give sketches of the algebra, skipping steps that are similar to this proof. Here's the careful proof.

$$\sum (x_i - \bar{x})^2$$

We dropped the indices of summation, since they will always be i from 1 to n. The next step is to expand the squaring inside the summation.

$$\sum (x_i^2 - 2x_i\bar{x} + \bar{x}^2)$$

We now split this into three different sums.

$$\sum x_i^2 - \sum 2x_i\bar{x} + \sum \bar{x}^2$$

Factor out constant quantities.

$$\sum x_i^2 - 2\bar{x} \sum x_i + \bar{x}^2 \sum$$

The middle sum is just the total of the numbers, call it t, and the last sum is the number of numbers, n

$$\sum x_i^2 - 2\bar{x}(t) + \bar{x}^2(n)$$

Note that $\bar{x} = t/n$, so let's eliminate $\bar{x}$.

$$\sum x_i^2 - 2t/n(t) + (t/n)^2(n)$$

Combine factors in the last two terms.

$$\sum x_i^2 - 2t^2/n + t^2/n$$

Combine the last two terms.

$$\sum x_i^2 - t^2/n$$

This last expression is just the right side of the equation, once we substitute the definition of t back in.

$$\sum x_i^2 - 1/n \left(\sum x_i\right)^2$$

Which formula should we use in our implementation? There are several issues in answering that question:

Efficiency The first expression requires computing the mean, which is n additions and a division. Then it requires n subtractions, squarings and additions. The second expression requires n squarings and additions to compute the first sum, n additions, a squaring and a division for the second, and a final subtraction. We can compare them as follows:

	Adds	Subs	Squares	Divides
$\sum(x_i - \bar{x})^2$	2n	n	n	1
$\sum x_i^2 - t^2/n$	2n	1	n+1	1

Basically, there's only a hair's breadth difference between the two. On most machines, squaring is more expensive than even quite a few subtractions, but the efficiency issue will clearly be our least important.

Data Representation The first expression clearly requires two passes over the data—once to compute the mean and the second time to compute the variance. The second expression requires only one pass over the data, because we can compute the total while we're summing the x_i^2; we can even count the data to compute n if that's necessary. For some data representations, this issue will dominate.

In CLASP, there are two ANOVA functions, one in which the values are structurally grouped (sequences of sequences) and another in which each value is paired with a key that tells what group it's in; for one-way ANOVA, these functions are 'anova-one-way-groups' and 'anova-one-way-variables' respectively. With the structural representation, it's easy to do several passes over a group. The other representation requires figuring out the groups, which is not too hard, but may tip the scales towards the one-pass expression.

Numerical Accuracy If a data value lies quite close to the mean, the difference will be very small, and squaring it could produce a numerical underflow. At the very least, the first expression (subtract and square) will tend to cancel the significant digits and square the uncertain digits, resulting in numerical instability. The second expression will preserve the significant digits for as long as possible, but because it is adding up squared data values, it runs the risk of numerical overflow. (Numerical overflow and underflow will only happen if the data are floating point; if they are exact (integer or rational), the exactness will be preserved by the CLASP functions.) Numerical accuracy suggests that the second expression is better.

Since there are two ways of computing the variance, couldn't we compute *both* and use them to improve our numerical accuracy? For example, we could signal an error if they are very different, or we could simply take their average as the "true" variance. We could do this, but it seems excessive for most purposes. However, we will shortly see a way to exploit this redundancy.

The one-way ANOVA computes three sums of squares, called the sums of squares total, group, and error, or SST, SSG and SSE, respectively. By definition, they are the following:

$$\begin{aligned} SST &= \sum_{i=1}^{I}\sum_{j=1}^{J_i}(x_{ij} - M)^2 \\ SSG &= \sum_{i=1}^{I}\sum_{j=1}^{J_i}(M_i - M)^2 \\ SSE &= \sum_{i=1}^{I}\sum_{j=1}^{J_i}(x_{ij} - M_i)^2 \end{aligned}$$

The notation is a little complicated, but not too bad. There are I groups, and they all might be different sizes, denoted J_i. Thus, each of these summations has one term for each data value, x_{ij}. The first sum of squares, SST, subtracts the data values from the grand mean, denoted M. The second sum of squares, SSG, subtracts the mean of the ith group, denoted M_i, from the grand mean. The third sum of squares, SSE, subtracts the data values from the group means.

There is a fundamental identity, called the ANOVA identity, that states:

$$SST = SSG + SSE$$

This identity doesn't look obvious from the formulas above; however, it becomes more obvious when the computing formulas are derived. We will denote the grand total by T and the number of data values by N, so the grand mean is just $M = T/N$. Similarly, we will denote the group totals by T_i, so the group means are $M_i = T_i/J_i$.

$$\begin{aligned} SST &= \sum\sum(x_{ij} - M)^2 \\ & \sum\sum x_{ij}^2 - 2M\sum\sum x_{ij} + \sum\sum M^2 \end{aligned}$$

$$
\begin{aligned}
& \sum\sum x_{ij}^2 - 2MT + M^2N \\
& \sum\sum x_{ij}^2 - T^2/N \\
SSG \;=\; & \sum\sum (M_i - M)^2 \\
& \sum J_i(M_i - M)^2 \\
& \sum J_iM_i^2 - 2J_iM_iM + M^2 \\
& \sum J_iM_i^2 - 2M\sum J_iM_i + M^2\sum J_i \\
& \sum J_i(T_i/J_i)^2 - 2MT + M^2N \\
& \sum T_i^2/J_i - T^2/N \\
SSE \;=\; & \sum\sum (x_{ij} - M_i)^2 \\
& \sum\sum x_{ij}^2 - 2\sum\sum x_{ij}M_i + \sum\sum M_i^2 \\
& \sum\sum x_{ij}^2 - 2\sum M_iT_i + \sum J_iM_i^2 \\
& \sum\sum x_{ij}^2 - 2\sum T_i^2/J_i + \sum J_i(T_i/J_i)^2 \\
& \sum\sum x_{ij}^2 - \sum T_i^2/J_i
\end{aligned}
$$

SST	SSG	SSE
$\sum\sum(x_{ij} - M)^2$	$\sum\sum(M_i - M)^2$	$\sum\sum(x_{ij} - M_i)^2$
$\sum\sum x_{ij}^2 - 2M\sum\sum x_{ij} + \sum\sum M^2$	$\sum J_i(M_i - M)^2$	$\sum\sum x_{ij}^2 - 2\sum\sum x_{ij}M_i + \sum\sum M_i^2$
$\sum\sum x_{ij}^2 - 2MT + M^2N$	$\sum J_iM_i^2 - 2J_iM_iM + M^2$	$\sum\sum x_{ij}^2 - 2\sum M_iT_i + \sum J_iM_i^2$
$\sum\sum x_{ij}^2 - T^2/N$	$\sum J_iM_i^2 - 2M\sum J_iM_i + M^2\sum J_i$	$\sum\sum x_{ij}^2 - 2\sum T_i^2/J_i + \sum J_i(T_i/J_i)^2$
	$\sum J_i(T_i/J_i)^2 - 2MT + M^2N$	$\sum\sum x_{ij}^2 - \sum T_i^2/J_i$
	$\sum T_i^2/J_i - T^2/N$	

We see a lot of recurring expressions here, so we can see the relationships of these expressions better by making the following substitutions:

$$
\begin{aligned}
A &\Rightarrow \sum\sum x_{ij}^2 \\
B &\Rightarrow T^2/N \\
C &\Rightarrow \sum T_i^2/J_i
\end{aligned}
$$

With those substitutions, we get the following formulas for the sums of squares:

$$
\begin{aligned}
SST &= A - B \\
SSG &= C - B \\
SSE &= A - C
\end{aligned}
$$

It's now obvious why the ANOVA identity is true.

What's the point of all this? First, the code for calculating the sums of squares will calculate A, B, and C, and that code wouldn't make any sense to a reader without this explanation. Second, we can calculate any one of these sums of squares by the two-pass method and compare it to the one-pass method as a test for numerical accuracy. The largest sum is SST and so it is the one most liable to produce floating overflow, especially with these calculating formulas, since A will be bigger than SST. SST also computes with the largest differences (subtracting data from the grand mean rather than a group mean), and so is the least likely to suffer from numerical inaccuracy by the two-pass method. Therefore, we will calculate SST by the two-pass method as well as by $A - B$. Rather than enforce some arbitrary precision, we will simply report both values for SST to the users, who can judge the numerical accuracy for themselves.

A.3 Two Way Analysis of Variance

Unlike the one way ANOVA, there are mathematical difficulties with the two way ANOVA if there are unequal cell sizes. Our method for this case differs fron the standard two-anova by (1) the use of cell means as single scores, (2) the division of squared quantities by the number of cell means contributing to the quantity that is squared and (3) the multiplication of a "sum of squares" by the harmonic mean of the sample sizes. The reference used is [8].

A.4 Linear Regression

Linear regression fits a linear model to the data using a least squares fit. If the linear model has more than one independent (x) variable, we will call it multiple linear regression. Multiple linear regression is implemented by matrix methods, which are covered in the next section. This section is devoted to the special case of linear regression in which we fit a line to bivariate data. Thus, we only have two variables, x and y. We will give the computational formulas we implemented. The main references we used are Devore [4], Freund [5], and Bohrnstedt and Knoke [1].

The linear regression statistics can be calculated from just six summary statistics of the original data. Since authors use different notation for the regression statistics and the summary statistics, we have also decided to invent our own notation. This will be useful later, when we attempt to resolve discrepancies between the formulas given by different authors. The six summary statistics are the following. The second column is the symbol used in the Common Lisp code, the third column is the mathematical notation we are inventing. These are all easy summary statistics, computable in one pass over the data, which makes them useful for bootstrapping.

	CL symbol	math symbol		definitions
1	n	n		the number of data points, (x_i, y_i)
2	x	x^+	$\sum_{i=1}^{n} x_i$	sum of the x's
3	y	y^+	$\sum_{i=1}^{n} y_i$	sum of the y's
4	x2	$(x^2)^+$	$\sum_{i=1}^{n} (x_i)^2$	sum after squaring
5	y2	$(y^2)^+$	$\sum_{i=1}^{n} (y_i)^2$	sum after squaring
6	xy	$(xy)^+$	$\sum_{i=1}^{n} x_i y_i$	AKA dot product of $\vec{x}$ and $\vec{y}$

CLASP implements linear regression functions that return three amounts of information: minimal, brief, and verbose. The functions take two different kinds of argument lists: two sequences of data, or the six summary statistics. Thus, there are six linear regression functions, depending on how you want to supply your arguments and how much computation you want done. All of the ones that take data sequences just calculate the summary statistics and pass them on to the ones that take summary statistics, so we will only discuss the latter in this section.

From these summary statistics, we will calculate the following sums of squares. The first formula on each line is the intuitive definition in terms of variation or covariation around the means and the second formula is an easy computing formula in terms of summary statistics.

$$\begin{aligned}
\mathrm{SSX} &= \sum(x_i-\bar{x})^2 &&= (x^2)^+ - (x^+)^2/n \\
\mathrm{SSY} &= \sum(y_i-\bar{y})^2 &&= (y^2)^+ - (y^+)^2/n \\
\mathrm{SSXY} &= \sum(x_i-\bar{x})(y_i-\bar{y}) &&= (xy)^+ - x^+y^+/n
\end{aligned}$$

Notice that all of these formulas involve a division by n. Since division is an expensive arithmetic operation, second only to square root, we try to avoid it wherever possible. Consequently, we found it efficient to compute the following instead.

$$\begin{aligned}
\mathrm{NSSY} &= n(x^2)^+ - (x^+)^2 \\
\mathrm{NSSY} &= n(y^2)^+ - (y^+)^2 \\
\mathrm{NSSXY} &= n(xy)^+ - x^+y^+
\end{aligned}$$

Now we are ready to give computational formulas for the slope and intercept of the regression line. Our formula for slope matches Devore's, and is algebraically equivalent to the other authors' formulas. Indeed, it was Devore's use of multiplying through by n that inspired us to do so everywhere.

$$\begin{aligned}
\text{slope} &: \frac{\mathrm{NSSXY}}{\mathrm{NSSX}} \\
\text{intercept} &: \frac{y^+ - (\text{slope})x^+}{n}
\end{aligned}$$

The slope and intercept are the only statistics calculated by the function 'linear-regression-minimal.' That function doesn't need the 'y2' variable, $(y^2)^+$, but takes it anyhow, just for consistency with the other linear regression functions.

The next statistics all depend on the sum of squares of the residuals—the remaining error not accounted for by the regression model. Most authors denote this quantity with SSE. All define it the same way, based on the difference between y_i and the y value of the line at the corresponding x_i, denoted $\hat{y}_i$.

$$\mathrm{SSE} = \sum(y_i-\hat{y}_i)^2 = \sum\left(y_i - [(\text{slope})x_i + \text{intercept}]\right)^2$$

Just as with the analysis of variance, there is an identity relating the total sum of squares, usually denoted SST, to the regression sum of squares, denoted SSR, and the error sum of squares, SSE. That identity is

$$\mathrm{SST} = \mathrm{SSR} + \mathrm{SSE}$$

By definition, SST=SSY, so we will use SSY to avoid introducing any new symbols. Because of this identity, and the fact that we already have SSY, we should compute SSR or SSE, whichever is easier, and then obtain the other by subtraction.

The authors all give different computing formulas for SSE. Translating into our notation, their computing formulas are:

$$\begin{aligned}
\text{BK} &\quad \mathrm{SSY}(1-r^2) \\
\text{Devore} &\quad (y^2)^+ - (\text{intercept})y^+ - (\text{slope})(xy)^+ \\
\text{Freund} &\quad \mathrm{SSY} - (\text{slope})\mathrm{SSXY}
\end{aligned}$$

It's hard to believe that these are all algebraically equivalent! Still, because of the subtractions in Freund' and BK's formulas, it's easy to see that SSR=(slope)SSXY. Therefore, our code will compute the following:

$$\begin{aligned}\text{NSSR} &= \text{(slope)NSSXY}\\ \text{NSSE} &= \text{NSSY} - \text{NSSR}\end{aligned}$$

At last, we are ready to compute the coefficient of determination, denoted r^2. We are also ready to compute the standard error of the slope, which is used in t-tests on the slope and also in building confidence intervals on the slope. Fortunately, all the authors agree on how to compute these statistics.

$$\begin{aligned}r^2 &= \frac{\text{NSSR}}{\text{NSSY}}\\ \text{std. err. slope} &= \sqrt{\frac{\text{NSSE}}{(n-2)\text{NSSX}}}\end{aligned}$$

Note that the degrees of freedom for the t-distribution of the slope is $n-2$. Also note how we have taken advantage of multiplying through by n in all our sums of squares.

The next step is to perform a t-test on the slope parameter to test the hypothesis that the slope is non-zero. This t-test is equivalent to testing whether the correlation coefficient, r, is non-zero and equivalent to testing whether the F statistic (mean square regression divided by mean square error) is greater than one. The t-statistic is:

$$t = \frac{\text{slope}}{\text{std. err. slope}}$$

and is tested with $n-2$ degrees of freedom. This computes a p-value which is also returned.

In summary, the "brief" linear regression functions return the following information: slope, intercept, coefficient of determination, standard error of the slope, and the p-value.

The "verbose" linear regression functions return even more statistics, but most are redundant with others. They return the correlation coefficient, r, but the user can get that by simply taking the square root of the coefficient of determination, r^2. It also returns an ANOVA table, computing the F statistic and p-value rather than the t-statistic and p-value that the "brief" functions do. The ANOVA table looks like this:

Source of Variation	degrees of freedom	Sum of squares	Mean Square	F	p-value
Regression	1	SSR	MSR = SSR/1	MSR/MSE	p
Error	$n-2$	SSE	MSE = SSE/$(n-2)$		
Total	$n-1$	SSY			

In addition to all this, the "verbose" functions return the standard error of the intercept. Unfortunately, there are several conflicting formulas for the standard error of the intercept, which we have not yet resolved as of this release. For now, we return nil, deeming less information better than bad information.

A.5 Singularities in Multiple Regression

This section describes the theory behind an error message that you may encounter in computing multiple regression models using CLASP/. If you never encounter the message or simply don't care, you may skip this section. The error message is something like the following:

```
There are columns which appear to be colinear in this regression.
```

What this message means is that there is a redundancy among your independent (predictor) variables, so that the solution to the regression is underconstrained. To be more precise, the listed variables (their indices in the regression), are a linear combination of some of the others. For example, in the message given above, the fifth independent variable seems to be redundant given the other independent variables. This results in underconstraining the solution, because the coefficient of that variable can any value, just by compensating in the coefficients of the other variables. In linear algebra terms, the independent variables are not linearly independent.

The solution, naturally, is to eliminate enough variables to restore the linear independence of the set of independent variables. Any one of the variables in the linearly dependent set can be deleted, so this decision is left to you. Deleting the listed variables is, of course, an easy option, but you may prefer to delete other.

Actual linear dependence is unlikely in real data, barring an encoding mistake, simply because experimenters rarely measure the same thing twice or vary two parameters that are so similar. Nevertheless, near singularities can occur when several variables are *nearly* linearly dependent, which can certainly happen in any experimental situation. For example, consider measuring two variables that happen to be highly correlated, which means $x \approx cy$, for some correlation constant c. Near singularities can cause numerical instability in solving the regression, leading to constants that are very large as the vectors that they multiply nearly cancel. It is better to eliminate such near cancelations by removing one of the variables from the regression.

That's the short answer. If you'd like more information, keep reading.

A.5.1 Singularity Example

Suppose that you are attempting to predict z by the variables x and y. We'll first consider an example without singularities and another with singularities. For now, ignore the fact that multiple regression is doing a least-squares fit to the data; we will be looking for exact solutions. Here's a concrete example:

$$\begin{array}{c} \begin{array}{cc} x & y \end{array} \\ \begin{bmatrix} 1 & 2 \\ 4 & 5 \end{bmatrix} \end{array} \begin{array}{c} \\ \begin{bmatrix} a \\ b \end{bmatrix} \end{array} \begin{array}{c} \\ = \end{array} \begin{array}{c} z \\ \begin{bmatrix} 3 \\ 6 \end{bmatrix} \end{array}$$

What this means is that we want to find two coefficients, a and b, such that $ax + by = z$, which is our regression equation, predicting z as a linear function of x and y, which are variables in our dataset.

We can solve these simultaneous equations to find that $a = -1$ and $b = 2$ is the solution to the regression. Note that if we had the inverse of the data matrix, we could multiply this equation by that inverse to get our solution vector in one matrix operation.

Now, let's repeat that example except in a case where there is dependency among the columns of the matrix.

$$\begin{array}{cc} x & y \\ \end{array} \qquad\qquad z$$
$$\begin{bmatrix} 1 & 2 \\ 4 & 8 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} 3 \\ 12 \end{bmatrix}$$

Because this is such a small example, it's easy to see that the dependency is that $y = 2x$, but in a large dataset, it's much harder to see these problems. It's especially hard if several variables are involved, say if $y = 2x + 4w - 3v$. When we try to solve these simultaneous equations, we find cancelations and therefore no unique solution. (If we were to solve it by inverting the data matrix, we would find that the matrix was not invertible—that is, it is singular.) Because of the dependency, any pair of coefficients (a, b) satisfying $a + 2b = 3$ satisfies these equations, so we have a family of solutions. Two solutions are $(3, 0)$ and $(1, 1)$.

If we eliminate one of the redundant variables, say y, we immediately find $a = 3$ as the unique solution. Note that this happens to be the same as taking the $b = 0$ element of family of solutions.

A.5.2 Singular Value Decomposition

The discussion in this section is a condensed extract from Numerical Recipes in C [12], sections 2.6 and 15.4. Any errors are my own; please refer to that excellent book and references therein for the whole truth.

There are many ways of solving multiple regression equations, which we can state as follows, where $\mathbf{X}$ is our data matrix whose columns are predictor variables, $\mathbf{Y}$ is our data vector whose sole column is our dependent variable, and $\mathbf{b}$ is the unknown vector of regression coefficients. (To be slightly more exact, $\mathbf{X}$ has an additional column made up entirely of ones; the parameter fitted to this "variable" is the intercept of the regression model. The variable $\mathbf{X}$ is sometimes called the *design matrix.*)

$$\mathbf{Xb} = \mathbf{Y}$$

Clearly, one way is to compute the inverse of $\mathbf{X}$, say by Gauss-Jordan elimination. Unfortunately, computing the inverse has some problems. First of all, Gauss-Jordan is not the most numerically stable method, and secondly, it fails utterly in the face of singularities or near-singularities. Instead, Numerical Recipes advises using Singular Value Decomposition (SVD), which, as you might guess, allows one to identify sources of singularities in the matrix. Their precise words are:

> Generally speaking, we recommend that you always use SVD techniques instead of using the normal equations. SVD's only significant disadvantage is that it requires an extra array of size $N \times M$ to store the whole design matrix. ...SVD can be significantly slower than solving the normal equations; however, its great advantage, that it (theoretically) *cannot fail*, more than makes up for the speed disadvantage. ([12], page 677–678, their emphasis)

The SVD technique avoids computing the inverse of $\mathbf{X}$, but instead decomposes $\mathbf{X}$ into a form that makes the inverse easy to compute, if desired, and, more importantly, allows one to identify and eliminate singularities before the inversion occurs. The decomposition is into matrices $\mathbf{UWV^T}$, such that $\mathbf{W}$ is a diagonal matrix and the inverse of $\mathbf{X}$ is:

$$\mathbf{X}^{-1} = \mathbf{V} \cdot [\text{diag}(1/w_i)] \cdot \mathbf{U}^T$$

The center matrix is another diagonal matrix whose diagonal elements are the reciprocals of the diagonals elements of $\mathbf{W}$. What this decomposition means is that in order to find the inverse of $\mathbf{X}$, we have to take the transposes of $\mathbf{V}$ and $\mathbf{U}$ (that is, flip them across the diagonal) and take the reciprocals of the diagonal elements of $\mathbf{W}$.

We can certainly transpose matrices without introducing any numerical instabilities, so all the trouble comes in taking reciprocals of the w_i. Any $w_i = 0$ is immediately a problem (a "singular value") and, indeed, any w_i that is near zero will blow up when its reciprocal is taken. Similarly, any errors in the w_i are magnified when the reciprocal is taken. (In practice, CLASP/ does not create the inverse of $\mathbf{X}$, it uses the decomposition to directly solve for the coefficient vector, $\mathbf{b}$, using backsubstitution.)

In short, if w_i is "small," the corresponding column of $\mathbf{X}$ is a suspect variable. If, when taking the reciprocal, we replace $1/w_i$ with zero, the result regression equation ignores the ith variable. Thus, by adopting a reasonable but arbitrary threshold for "small," CLASP/ can identify singular values in regression, report these to you, and find a solution anyhow. When CLASP/ reports that it has "zeroed variables" for you, it is doing to so prevent a singularity in the solution, thereby giving you the best anwer it can. You should probably follow up to make sure you understand how these linear dependencies arose in your data.

Let's see how this works in practice. How would SVD solve the singular example above? It turns out SVD would zero the first variable, a, and would find $(0.6, 1.2)$ as the solution. (SVD could just as easily have zeroed b.) This solution is clearly in the family of solutions, since $a + 2b = 3$. SVD's solution is, in some sense, a "minimal" solution, in keeping both suspicious coefficients as small as possible, driving them towards zero rather than towards infinity, which they would do if the singular $\mathbf{X}$ were inverted.

A.5.3 Implementation

The coefficients are only a small part of what CLASP/ computes for a linear regression. This section describes all the computed quantities and how they are computed. It uses some of the variable names from the code, so acts as extended documentation of that code, but familiarity with the code will not be necessary to understand the description.

- coefficients: This is computed using the technique advised by Numerical Recipes. The design matrix $\mathbf{X}$ is created and filled from the variables. It is then decomposed into $\mathbf{U}$, $\mathbf{W}$, and $\mathbf{V}$ using their `svdcmp` function. The small diagonal elements of $\mathbf{W}$ are zeroed; this may result in printout to the screen. The coefficient vector is determined by backsubstitution, using their `svbksb` function.

The following terms should be familiar to those who understand the analysis of variance (ANOVA). The main difference is that we only look at the amount of variance explained by the regression model, rather than the amount explained by each independent variable.

- sum squared error: This quantity is the sum of the squares of the deviations of the fitted model from the data, abbreviated SSE. This quantity is also called chi-square, and is the amount of variation in the dependent variable that is *not* explained by the model. SSE is computed by going through the design matrix row by row, multiplying the x_i by the corresponding coefficient to get $\hat{y}_i$, subtracting that from the true y_i, squaring the result and summing.

- mean squared error: Often denoted MSE or even $\hat{\sigma}^2$, this is simply the mean of the SSE. There are $N - K$ degrees of freedom, where N is the number data points (the number of rows of the design matrix) and K is the number of parameters that have been estimated. Thus, MSE=SSE/(N-K). MSE estimates the variance of the error term in the model:
$$y = b_0 + \sum b_i x_i + \epsilon$$
By the nature of regression, the expected value of the error, $E(\epsilon)$ is zero (that is, regression yields an unbiased estimate of the coefficient vector $\mathbf{b}$). MSE is an estimate of the variance of ϵ, hence the notation $\hat{\sigma}^2$, where σ^2 is the true population variance.

- sum of squares of regression: Sometimes abbreviated SSR, this quantity is the amount of variation in the dependent variable that is accounted for by the regression model. It is computed by subtraction from the total variation, using the identity SSR+SSE=SST, where SST is computed using the standard computing formula:
$$\mathrm{SST} = \sum_{i=1}^{N} Y_i^2 - \frac{1}{N}\left(\sum_{i=1}^{N} Y_i\right)^2$$

- mean square of regression. This is the mean of SSR, computed using $K - 1$ degrees of freedom, where K is the number of parameters estimated by the regression. K is one more than the number of independent variables, because regression estimates one coefficient for each independent variable and also estimates the y-intercept of the model, called b_0 above. Therefore, MSR is SSR divided by the number of independent variables. (Some textbooks denote the number of parameters by K and others the number of independent variables by K, so be careful in trying to match our notation with your favorite statistics text.)

- R^2, pronounced R-squared. This is the fraction of variance accounted for by the model, and is computed simply as SSR/SST. R^2 is always between zero and one, and the closer it is to one, the better the model fits the data.

- F. This is the ratio of the explained variance to the unexplained variance, computed as MSR/MSE, and is the primary test statistic for the regression model. The numerator and denominator degrees of freedom are determined by MSR $(K - 1)$ and MSE $(N - K)$.

The following computations are unique to multiple linear regression.

- The Covariance Matrix, COV, sometimes denoted $\sigma(\hat{\mathbf{b}})$. This matrix is the covariance among the fitted *parameters* (the coefficients), not among the independent variables. Therefore, its entries are estimates of population values and are not sample statistics. This matrix measures the error in the parameters. For example, the diagonal elements are the variances of the coefficients. The off-diagonal elements measure the extent to which the error in one parameter is correlated with the error of another.* CLASP/ uses the diagonal elements of this matrix to compute the standard error of each coefficient. The matrix is computed using the `svdvar` routine from Numerical Recipes. The matrix is not actually reported by CLASP/, but it could be.

- t statistics for the coefficients. For each coefficient b_i, we can compute a corresponding t_i that can be used in a t test to determine if that coefficient is zero. If it is zero, it should presumably be deleted from the model, thereby simplifying the model. Note that each t_i tests whether b_i is zero assuming that the rest of the model is unchanged. The ts should not be used to test whether several parameters are zero; one may become non-zero when another is removed from the model. Similarly, F tests whether all the bs are simultaneously zero.

  As with all t tests against a null hypothesis of zero, the t statistic is $t_i = b_i/\text{s.d.}(b_i)$. The standard deviation of b_i is the square root of its variance, and its variance is MSE$\cdot$COV(i, i). CLASP/ uses the parameter covariance matrix and MSE both computed above, to find t_i for each b_i.

- Standardized coefficients, also called *betas*. These are the coefficients that regression would find if the input were standardized values instead of raw values. A standardized value is expressed in terms of its deviation from the mean: $\tilde{x} = (x - \bar{x})/\text{s.d.}(x)$. The effect that standardizing has on the coefficients can easily be calculated by converting a model using standardized coefficients into the form of a model using non-standardized coefficients:

$$\begin{aligned}
\tilde{y} &= \sum_i \beta_i \tilde{x}_i + \beta_0 \\
\frac{y - \bar{y}}{s_y} &= \sum_i \beta_i \frac{x_i - \bar{x}_i}{s_{x_i}} + \beta_0 \\
\frac{y}{s_y} &= \sum_i \beta_i \frac{x_i}{s_{x_i}} + \text{constants} \\
y &= \sum_i \beta_i x_i \frac{s_y}{s_{x_i}} + \text{constants}
\end{aligned}$$

  This last equation is the regression model using raw values x_i and y, instead of the standardized values $\tilde{x}_i$ and $\tilde{y}$. Therefore, the coefficient of x_i is b_i, and so:

$$\beta_i = b_i \frac{s_{x_i}}{s_y} = b_i \sqrt{\frac{\text{VAR}(x_i)}{\text{VAR}(y)}}$$

  This is the computation that CLASP/ makes; computing the variance the dependent variable and of each independent variable and multiplying the b_i by the square root of the ratio.

- Correlation Matrix. This is a matrix of the correlations of each independent variable with each of the others and also with the dependent variable. It doesn't use any prior computations and is almost entirely separate from the rest of the regression computations, except that we use some of these correlations, below.

*From the **UWV** decomposition of the design matrix, the principal axes of the error hyper-ellipsoid can be recovered. The principal axes, of course, are uncorrelated, since they are orthogonal. Refer to Numerical Recipes, section 15.6 for the details.

- The portion of regression sum of squares due to each variable. Like the Analysis of Variance, regression can be viewed as breaking up the sum of squares of the dependent variable into parts. This calculation breaks SSR into $K - 1$ parts, one for each coefficient, as follows

$$\mathrm{SSR} = \sum_j b_j \left[\left(\sum_i x_{ij} y_i \right) - \frac{1}{N} \left(\sum_i y_i \right) \left(\sum_i x_i \right) \right]$$

  CLASP/ calculates these quantities using exactly the formula above.

- R Squared. This quantity is a list of the squares of the correlation coefficients between each independent variable and the other independent variables and also the dependent variable. That is, it is the squares of the elements of the correlation matrix computed earlier.

A.6 Regression Analysis: Matrix Method†

To implement 'linear regression' in CLASP the decision was made to use matrix algebra methods for the calculation of the statistics.‡ The basis of the method is in the ability to calculate the sum of squares and cross products for the independent and dependent variables through matrix multiplication, then using these values to calculate the regression coefficients, correlation coefficients, and percentage variance of the dependent variable, Y. The advantage of using matrix algebra over direct raw score calculation is not obvious when considering an example with only two independent variables, however the complexity of the direct calculations becomes apparent as the number of independent variables increase. The following formulae illustrate the calculations involved using the direct raw score method, for two independent, X, variables, with $\sum x_i$ the sum of deviation scores for the i'th X variable, and $\sum x_i y$ the sum of the cross products of the deviation scores of Y and the i'th X variable.

$$b_1 = \frac{(\sum x_2^2)(\sum x_1 y) - (\sum x_1 x_2)(\sum x_2 y)}{(\sum x_1^2)(\sum x_2^2) - (\sum x_1 x_2)^2}$$

$$b_2 = \frac{(\sum x_1^2)(\sum x_2 y) - (\sum x_1 x_2)(\sum x_1 y)}{(\sum x_1^2)(\sum x_2^2) - (\sum x_1 x_2)^2}$$

As can be seen these equations expand with the introduction of each new independent variable, making them very unwieldy. In order to perform the calculations with matrices we first introduce a new variable, X_0, a unit vector (a vector of ones), of the same size as the vectors X_i, with this the matrix $\mathbf{Z}$ is built, with the columns being defined as $X_0 \cdots X_i$ and Y. Using the equation

$$Y = a + b_1 X_1 + b_2 X_2 + \cdots + b_k X_k + e$$

in matrix form, where $\mathbf{Y}$ is a $N \times 1$ column vector, $\mathbf{X}$ is a $N \times k$ matrix with each column corresponding to X_i, where X_0 is the unit vector described above, a is the first element of the $1 \times k$ row vector of regression coefficients, $\mathbf{b}$, and $\mathbf{e}$ is the $1 \times k$ column vector of the error terms of the equation, given by the form $Y = \hat{Y} + e$, where $\hat{Y}$ is the predicted value of Y from the regression equation, produces the following equation:

$$\begin{array}{ccccc} Y & = a + & b & X & + \quad e \\ \begin{bmatrix} Y_1 \\ Y_2 \\ Y_3 \\ \vdots \\ Y_N \end{bmatrix} & & \begin{bmatrix} a \\ b_1 \\ b_2 \\ \vdots \\ b_N \end{bmatrix} & \begin{bmatrix} X_{10} & X_{11} & X_{12} & \cdots & X_{1k} \\ X_{20} & X_{21} & X_{22} & \cdots & X_{2k} \\ X_{30} & X_{31} & X_{32} & \cdots & X_{3k} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ X_{N0} & X_{N1} & X_{N2} & \cdots & X_{Nk} \end{bmatrix} & \begin{bmatrix} e_1 \\ e_2 \\ e_3 \\ \vdots \\ e_N \end{bmatrix} \end{array}$$

To calculate the values of $\mathbf{b}$ and $\mathbf{e}$ we use the sum of squares and cross product matrix produced by multiplying the transpose of the matrix $\mathbf{Z}$, $\mathbf{Z}'$, by $\mathbf{Z}$, $\mathbf{Z}'\mathbf{Z}$. As can be seen the result of this multiplication contains the summed squares of the individual variables along the diagonal, with $\sum X_0^2 = \sum X_0$, and the cross products along the off diagonals.

†This discussion is a condensation of chapters 2 and 3 from Pedhazur's book, *Multiple Regression in Behavioral Research* [10].

‡Although we still provide this method, it is obsolete, the primary method for computing Multiple Regression can be found in Section A.5.2.

$$\mathbf{Z'Z} = \begin{bmatrix} \sum X_0 & \sum X_1 & \sum X_2 & \cdots & \sum X_k & \sum Y \\ \sum X_1 & \sum X_1^2 & \sum X_1X_2 & \cdots & \sum X_1X_k & \sum X_1Y \\ \sum X_2 & \sum X_1X_2 & \sum X_2^2 & \cdots & \sum X_2X_k & \sum X_2Y \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \sum X_k & \sum X_kX_1 & \sum X_kX_2 & \cdots & \sum X_k^2 & \sum X_kY \\ \sum Y & \sum YX_1 & \sum YX_2 & \cdots & \sum YX_k & \sum Y^2 \end{bmatrix}$$

From this we get the two component matrices $\mathbf{X'X}$, X transpose X, and $\mathbf{X'Y}$, X transpose Y,

$$\mathbf{X'X} = \begin{bmatrix} \sum X_0 & \sum X_1 & \sum X_2 & \cdots & \sum X_k \\ \sum X_1 & \sum X_1^2 & \sum X_1X_2 & \cdots & \sum X_1X_k \\ \sum X_2 & \sum X_1X_2 & \sum X_2^2 & \cdots & \sum X_2X_k \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \sum X_k & \sum X_kX_1 & \sum X_kX_2 & \cdots & \sum X_k^2 \end{bmatrix} \quad \mathbf{X'Y} = \begin{bmatrix} \sum Y \\ \sum X_1Y \\ \sum X_2Y \\ \vdots \\ \sum X_kY \\ \sum Y^2 \end{bmatrix}$$

which are then used to determine the matrix $\mathbf{b}$, using the following equation:

$$b = (X'X)^{-1}X'Y$$

where $(\mathbf{X'X})^{-1}$ is the inverse of the matrix $\mathbf{X'X}$. Recall that the inverse of a matrix is a matrix such that $\mathbf{X}^{-1}\mathbf{X} = \mathbf{I}$, where $\mathbf{I}$ is the identity matrix. So multiplying by the inverse of a matrix is equivalent to division. The significance of the coefficients can be tested using a standard t test, using the formula $t = \frac{b}{s_b}$, where s_b is the error term of the coefficient. The value of t indicates whether or not the coefficient is significantly different from zero, with a critical value at the 0.05 level of 1.96 for a two tailed test. A two tailed test is used as it does not matter whether the coefficient is positive or negative. With $\mathbf{b}$ the sum of squares of the regression and the sum of squares of the residual are calculated, with:

$$ss_{reg} = \mathbf{b'X'Y} - \frac{(\sum Y)^2}{N}$$

$$ss_{res} = \mathbf{e'e} = \sum e^2 = \mathbf{Y'Y\text{-}b'X'Y}$$

where $\mathbf{e'e}$ is the sum of squared error terms for the regression. The sum of squares of the regression indicate what portion of the total squared deviation scores, $\sum y^2$, of the dependent variable, are due to the effects of the independent variables. The sum of squares of the residual indicate the proportion which is unaccounted for by the independent variables. Recall that

$$ss_{total} = ss_{reg} + ss_{res} = \sum y^2 = \mathbf{Y'Y} - \frac{(\sum Y)^2}{N}$$

using these values the significance of the regression can be tested.

The key value of interest from the regression is the F statistic of the R^2 value, which is the percentage of the variance accounted for by the regression. The value of R^2 can be calculated using either of the following two equations:

$$R^2 = \frac{\mathbf{b'X'Y} - \frac{(\sum Y)^2}{N}}{\mathbf{Y'Y} - \frac{(\sum Y)^2}{N}} = \frac{ss_{reg}}{\sum y^2}$$

$$R^2 = 1 - \frac{\mathbf{e'e}}{\mathbf{Y'Y} - \frac{(\sum Y)^2}{N}} = 1 - \frac{ss_{res}}{\sum y^2}$$

where $\sum y^2$ is the sum of squared deviations for the dependent variable Y. The value of F indicates the probability that such a value could occur due to chance. F is calculated as:

$$F = \frac{ss_{reg}/df_{reg}}{ss_{res}/df_{res}} = \frac{R^2/k}{(1 - R^2)/(N - k - 1)}$$

where k is the number of independent variables and N is the number of observations. Additionally the coefficients of correlation between each X_i and Y, as well as between the X_i's can be calculated, with the squared coefficient for each of the X_i's indicating its contribution to the variance of Y.

It is possible, as with other statistics, to produce results which are significant, but not meaningful, as well as meaningful, but not significant. Evaluating the result of a linear regression, especially when there is interaction between the independent variables, can be a very difficult task. Often a small value of R^2 can discourage a researcher, even though its value is significant, and, by the same token, a large value of R^2 does not immediately confer meaning to the results of the regression. In cases where the results of the analysis are not clear, or appear to be at a border line, a useful strategy can be to reduce the number of independent variables being used, looking for a change in the analysis as the interactive effects are reduced. The covariance of the independent variables, which is indicated by the correlation between the X_i's, is a good indicator of the interaction between the independent variables. In general it is best to refer to a statistics text when evaluating the results of a linear regression.

Appendix B

Clip Examples

B.1 More Experiment Definitions Using CLIP

The following three examples, taken from different domains, illustrate experiment control and data collection using CLIP. The third example includes output files (in CLASP format) for time-series clips.

B.1.1 Measuring an Airport Gate Maintenance Scheduler's Performance

This extended example of CLIP code was provided by Zachary Rubinstein. The system being studied is an AI scheduler developed by David Hildum for a domain called Airport Resource Management (ARM). The task is to schedule gate maintenance, providing and coordinating vehicles for servicing planes at terminal gates. The experiments shown here were part of a baseline study designed to measure the performance characteristics of several priority rating schemes under resource-constrained scenarios. The dependent variables are delay and fragmentation (how much shuttling between gates occurs). The independent variables are the competing heuristic rating schemes for determining the order in which resource requests are scheduled. For clarity, many of the implementation details have been omitted.

The example uses several super clips to organize the collection of similar kinds of data, such as all measures of delay. All data collection occurs *post hoc* and is written to a single summary file. Note that two experiments are defined, each of which uses the same clips. The function `delays-information`, called by the clip `delay`, provides an interesting example of the interface between clips and the system being instrumentated.

The `delay` and `fragmentation` clips are examples of *composite* clips. The values returned by the body of a composite clip are one-to-one mapped onto the components of the clip. Each component is itself a clip which accepts an argument and has a body that returns a value. This value is what is output to the data file. The number of values a composite clip produces is equal to the number of its components.

The components of a composite clip have a clip definition automatically generated for them. The default action for this automatically generated component clip is to return its single argument as a single value. For example, the `delay clip` below generates a clip for each of its components automatically. The clip designer can also override this default behavior by writing a clip for that component that does something with the argument before returning it.

```
(define-simulator arm :start-system arm:run-arm)

;;; ----------------------------------------------------------------------
;;; CLIP Definitions

(defclip airport ()
  "Return the Currently Loaded Airport"
  ()
  (arm::get-loaded-airport))

(defclip timetable ()
  "Return the Currently Loaded timetable"
  ()
  (arm::get-loaded-timetable))


(defclip delay ()
  "Composite clip to calculate the various delay clips."
  (:components (total-delay number-of-delays average-delay
                std-dev-delay maximum-delay duration-ratio))

  (delays-information))


(defclip fragmentation ()
  "Composite fragmentation clip to calculate the various fragmentation clips."
  (:components (total-servicing-time
                total-sig-frag-time
                total-setup-time
                total-travel-time
                ratio-of-frag-to-servicing
                ratio-of-setup-to-servicing
                ratio-of-travel-to-servicing))

  (resource-fragmentation-information 'arm:baggage-truck-planner-unit))



;;; ----------------------------------------------------------------------
;;; Experiments

(define-experiment limit-resources-numerically-over-rating-schemes
                       (&key (rating-schemes *defined-rating-schemes*)
                             (airport-delimiter 10)
                             (airports user::*10-FTS-AIRPORTS*)
                             (timetable-name :20-flights))
  "Using the various rating schemes, constrain the resources
and report the delays."
  :locals ((report-filename nil))
```

```
  :variables ((airport-name in airports)
             (rating-scheme in rating-schemes))
  :instrumentation (timetable
                  total-delay number-of-delays average-delay
                  std-dev-delay maximum-delay duration-ratio
                  total-servicing-time total-sig-frag-time
                  total-setup-time total-travel-time
                  ratio-of-frag-to-servicing
                  ratio-of-setup-to-servicing
                  ratio-of-travel-to-servicing)
  :before-experiment (progn (arm:zack-set-default-demo-parameters)
                          (arm:execute-load-timetable
                            :LOAD-SPECIFIC-FILE timetable-name))
  :before-trial (setf report-filename
                    (delay-initialize-trial rating-scheme
                      airport-delimiter airport-name))
  :after-trial (delay-report report-filename
                          airport-delimiter rating-scheme))


;;; ---------------------------------------------------------------------
;;; Example Invocation

#+COMMENT
(run-experiment 'dss:limit-resources-numerically-over-rating-schemes
                :args '(:rating-schemes
                         (:zack-1 :default :obo-minest-heuristic)
                         :airport-delimiter :20-ref
                         :airports (:detroit-17-bts :detroit-16-bts
                                    :detroit-15-bts :detroit-14-bts
                                    :detroit-13-bts :detroit-12-bts)
                         :timetable-name :20-flights)
                :output-file "hillary:reports;20-REF-FTS-DELAY-STATS.TEXT")


;;; ---------------------------------------------------------------------
;;; Experiments

(define-experiment run-over-rating-schemes
                        (&key (rating-schemes *defined-rating-schemes*)
                              (airport-delimiter 10))
  "Using the various rating schemes, constrain the resources, introduce
orders, and report the delays."
  :locals ((report-stream nil))
  :variables ((rating-scheme in rating-schemes))
  :instrumentation (airport timetable
                  total-delay number-of-delays
                  average-delay std-dev-delay
```

```
                    maximum-delay duration-ratio
                    total-servicing-time total-sig-frag-time
                    total-setup-time total-travel-time
                    ratio-of-frag-to-servicing
                    ratio-of-setup-to-servicing
                    ratio-of-travel-to-servicing)
  :before-trial (setf report-stream
                      (delay-initialize-trial rating-scheme
                        airport-delimiter))
  :after-trial (write-current-experiment-data ))


;;; ----------------------------------------------------------------------
;;; Example Invocation

#+COMMENT
(run-experiment 'dss:run-over-rating-schemes
                     :ARGS `(:RATING-SCHEMES
                              ,(cons :DEFAULT
                                     (cons :ZACK-1
                                           (remove :OPPORTUNISTIC
                                                   *defined-rating-schemes*
                                                   :TEST #'eq)))
                              :airport-delimiter 10)
                     :OUTPUT-FILE "hillary:reports;HILDUM-EXP-1.TEXT")



;;; ----------------------------------------------------------------------
;;; Delay Information

(defun delays-information ()
  (LOOP WITH local-delay-time = 0
        AND network = nil
        AND release-time = nil
        AND due-date = nil
        AND start-time = nil
        AND finish-time = nil
        AND avg-delay = 0
        AND std-dev-delay = 0
        AND max-delay = 0
        FOR order IN (find-units 'order
                       (make-paths '(scheduler * order)) :ALL)
        DO
        (setf network (order$network order))
        (setf release-time (order$release-time order))
        (setf due-date (order$due-date-time order))
        (setf start-time (task-network$start-time network))
```

```
(setf finish-time (task-network$finish-time network))
(setf local-delay-time (- finish-time due-date))
SUM (- finish-time start-time) INTO total-actual-duration
SUM (- due-date release-time) INTO total-desired-duration
WHEN (not (= local-delay-time 0))
COUNT local-delay-time INTO number-of-delays
AND
SUM local-delay-time INTO total-delay-time
AND
COLLECT local-delay-time INTO delays
FINALLY
(unless (zerop number-of-delays)
  (setf avg-delay (float (/ total-delay-time number-of-delays)))
  (setf std-dev-delay
        (LOOP FOR delay IN delays
              SUM (expt (- delay avg-delay) 2)
              INTO sum-of-squares
              FINALLY (return
                        (sqrt
                          (float
                            (/ sum-of-squares
                              (1- number-of-delays)))))))
  (setf max-delay (apply #'max delays)))
(return (values total-delay-time
                number-of-delays
                avg-delay
                std-dev-delay
                max-delay
                (float (/ total-actual-duration
                          total-desired-duration))))))
```

B.1.2 PHOENIX Real-Time-Knob Experiment

This section describes an experiment done in the PHOENIX fire-fighting simulation system. We give some background on the Phoenix system and on the experiment design to help the reader better interpret the experiment definition that follows. For more detail on the experiment and its results see Hart & Cohen, 1992 [6].

The PHOENIX System

PHOENIX is a multi-agent planning system that fights simulated forest-fires. The simulation uses terrain, elevation, and feature data from Yellowstone National Park and a model of fire spread from the National Wildlife Coordinating Group Fireline Handbook (National Wildlife Coordinating Group 1985). The spread of fires is influenced by wind and moisture conditions, changes in elevation and ground cover, and is impeded by natural and man-made boundaries such as rivers, roads, and fireline. The Fireline Handbook also prescribes many of the characteristics of our firefighting agents, such as rates of movement and effectiveness of various firefighting techniques. For example, the rate at which bulldozers dig fireline varies with the terrain. PHOENIX is a real-time simulation environment PHOENIX agents must think and act as the fire spreads. Thus, if it takes too long to decide on a course of action, or if the environment changes while a decision is being made, a plan is likely to fail.

One PHOENIX agent, the Fireboss, coordinates the firefighting activities of all field agents, such as bulldozers and watchtowers. The Fireboss is essentially a thinking agent, using reports from field agents to form and maintain a global assessment of the world. Based on these reports (e.g., fire sightings, position updates, task progress), it selects and instantiates fire-fighting plans and directs field agents in the execution of plan subtasks.

A new fire is typically spotted by a watchtower, which reports observed fire size and location to the Fireboss. With this information, the Fireboss selects an appropriate fire-fighting plan from its plan library. Typically these plans dispatch bulldozer agents to the fire to dig fireline. An important first step in each of the three plans in the experiment described below is to decide where fireline should be dug. The Fireboss projects the spread of the fire based on prevailing weather conditions, then considers the number of available bulldozers and the proximity of natural boundaries. It projects a bounding polygon of fireline to be dug and assigns segments to bulldozers based on a periodically updated assessment of which segments will be reached by the spreading fire soonest. Because there are usually many more segments than bulldozers, each bulldozer digs multiple segments. The Fireboss assigns segments to bulldozers one at a time, then waits for each bulldozer to report that it has completed its segment before assigning another. This ensures that segment assignment incorporates the most up-to-date information about overall progress and changes in the prevailing conditions.

Once a plan is set into motion, any number of problems might arise that require the Fireboss's intervention. The types of problems and mechanisms for handling them are described in Howe & Cohen, 1990 [7], but one is of particular interest here: As bulldozers build fireline, the Fireboss compares their progress to expected progress. If their actual progress falls too far below expectations, a plan failure occurs, and (under the experiment scenario described here) a new plan is generated. The new plan uses the same bulldozers to fight the fire and exploits any fireline that has already been dug. We call this error recovery method vreplanning. PHOENIX is built to be an adaptable planning system that can recover from plan failures. Although it has many failure-recovery methods, replanning is the focus of the experiment described in the next section.

Identifying the Factors that Affect Performance

We designed an experiment with two purposes. A confirmatory purpose was to test predictions that the planner's performance is sensitive to some environmental conditions but not others. In particular, we expected performance to degrade when we change a fundamental relationship between the planner and its environment—the amount of time the planner is allowed to think relative to the rate at which the environment changes—and not be sensitive to common dynamics in the environment such as weather, and particularly, wind speed. We tested two specific predictions: 1) that performance would not degrade or would degrade gracefully as wind speed increased; and 2) that the planner would not be robust to changes in the Fireboss's thinking speed due to a bottleneck problem described below. An exploratory purpose of the experiment was to identify the factors in the Fireboss architecture and PHOENIX environment that most affected the planner's behavior, leading to a causal model.

The Fireboss must select plans, instantiate them, dispatch agents and monitor their progress, and respond to plan failures as the fire burns. The rate at which the Fireboss thinks is determined by a parameter called the Real Time Knob. By adjusting the Real Time Knob we allow more or less simulation time to elapse per unit CPU time, effectively adjusting the speed at which the Fireboss thinks relative to the rate at which the environment changes.

The Fireboss services bulldozer requests for assignments, providing each bulldozer with a task directive for each new fireline segment it builds. The Fireboss can become a bottleneck when the arrival rate of bulldozer task requests is high or when its thinking speed is slowed by adjusting the Real Time Knob. This bottleneck sometimes causes the overall digging rate to fall below that required to complete the fireline polygon before the fire reaches it, which causes replanning. In the worst case, a Fireboss bottleneck can cause a thrashing effect in which plan failures occur repeatedly because the Fireboss can't assign bulldozers during replanning fast enough to keep the overall digging rate at effective levels. We designed our experiment to explore the effects of this bottleneck on system performance and to confirm our prediction that performance would vary in proportion to the manipulation of thinking speed. Because the current design of the Fireboss is not sensitive to changes in thinking speed, we expect it to take longer to fight fires and to fail more often to contain them as thinking speed slows.

In contrast, we expect PHOENIX to be able to fight fires at different wind speeds. It might take longer and sacrifice more area burned at high wind speeds, but we expect this effect to be proportional as wind speed increases and we expect PHOENIX to succeed equally often at a range of wind speeds, since it was designed to do so.

Experiment Design

We created a straightforward fire fighting scenario that controlled for many of the variables known to affect the planner's performance. In each trial, one fire of a known initial size was set at the same location (an area with no natural boundaries) at the same time (relative to the start of the simulation). Four bulldozers were used to fight it. The wind's speed and direction were set initially and not varied during the trial. Thus, in each trial, the Fireboss receives the same fire report, chooses a fire-fighting plan, and dispatches the bulldozers to implement it. A trial ends when the bulldozers have successfully surrounded the fire or after

120 hours without success. The experiment's first dependent variable then is Success, which is true if the fire is contained, and false otherwise. A second dependent variable is shutdown time (SD), the time at which the trial was stopped. For successful trials, shutdown time tells us how long it took to contain the fire. Two independent variables were wind speed (WS) and the setting of the Fireboss's Real Time Knob (RTK). A third variable, the first plan chosen by the Fireboss in a trial (FPLAN), varied randomly between trials. It was not expected to influence performance, but because it did, we treat it here as an independent variable.

WS The settings of WS in the experiment were 3, 6, and 9 kilometers per hour. As wind speed increases, fire spreads more quickly in all directions, and most quickly downwind. The Fireboss compensates for higher values of wind speed by directing bulldozers to build fireline further from the fire.

RTK The default setting of RTK for PHOENIX agents allows them to execute 1 CPU second of Lisp code for every 5 minutes that elapses in the simulation. We varied the Fireboss's RTK setting in different trials (leaving the settings for all other agents at the default). We started at a ratio of 1 simulation-minute/cpu-second, a thinking speed 5 times as fast as the default, and varied the setting over values of 1, 3, 5, 7, 9, 11, and 15 simulation-minutes/cpu-second. These values range from 5 times the normal speed at a setting of 1 down to one-third the normal speed at 15. The values of RTK reported here are rescaled. The normal thinking speed (5) has been set to RTK=1, and the other settings are relative to normal. The scaled values (in order of increasing thinking speed) are .33, .45, .56, .71, 1, 1.67, and 5. RTK was set at the start of each trial and held constant throughout.

FPLAN The Fireboss randomly selects one of three plans as its first plan in each trial. The plans differ mainly in the way they project fire spread and decide where to dig fireline. SHELL is aggressive, assuming an optimistic combination of low fire spread and fast progress on the part of bulldozers. MODEL is conservative in its expectations, assuming a high rate of spread and a lower rate of progress. The third, MBIA, generally makes an assessment intermediate with respect to the others. When replanning is necessary, the Fireboss again chooses randomly from among the same three plans. We adopted a basic factorial design, systematically varying the values of WS and RTK. Because we had not anticipated a significant effect of FPLAN, we allowed it to vary randomly.

RTK Experiment Clips

All data collection in this experiment occurs at the end of each trial. Many of these clips invoke system specific calls to the simulator and to agents' state memories, again, as in the first example, illustrating the interface between clips and the system under study. Of particular interest here is the script invoked to recreate the same experimental scenario for each trial (see the `:script` keyword to `define-experiment` below). This script has a set of instructions to the Phoenix simulator's event-scheduler that introduce a fixed set of environmental changes that are part of the experimental control.

```
(define-experiment real-time-knob-experiment (use-exp-style)
  "Simple experiments for exercising and testing the real time knob."
```

```
  :before-experiment (real-time-knob-experiment-init-before-experiment
                      use-exp-style)
  :before-trial (real-time-knob-experiment-init-before-trial)
  :after-trial (real-time-knob-experiment-after-trial)
  :after-experiment (real-time-knob-experiment-reset-after-experiment)
  :variables ((real-time-knob in '(1 3 5 7 9 11))
              (wind-speed in '(3 9 12)))
  :instrumentation (number-of-bulldozers
                    plan-to-contain-fire       ; FPLAN
                    all-plans-to-contain-fire
                    fires-started
                    shutdown-time              ; SD
                    number-of-fires-contained
                    total-fire-line-built
                    r-factor
                    area-burned
                    agents-lost
                    all-agent-instrumentation
                    fireboss-instrumentation
                    bulldozer-instrumentation)

  :script
  ((setup-starting-conditions "12:29"
     (progn
       (send (fire-system)
             :alter-environment-parameter 'wind-direction 315)
       (send (fire-system)
             :alter-environment-parameter 'wind-speed (wind-speed))))
   (start-fire "12:30"
     (send (fire-system) :start-fire 700 (point 50000 40000)))))

;;; -------------------------------------------------------------------
;;; Utility functions used by the :before- and :after- forms to
;;; initialize and reset the experiment environment

(defun real-time-knob-experiment-init-before-experiment (use-exp-style)
  (when use-exp-style
    (setf (interaction-style t) 'experiment))
  ;; Don't allow fires to skip over fire lines.
  (send (fire-simulation) :set-spotting-scale-factor 0)
  ;; Get rid of flank attack, etc.
  (modify-knowledge-base))

(defun real-time-knob-experiment-init-before-trial ()
  (gc-immediately :silent t))

(defun real-time-knob-experiment-after-trial ()
  ;; This is now done in the `shutdown-trial' method.
```

```
  (write-current-experiment-data))

(defun real-time-knob-experiment-reset-after-experiment ()
  (setf (interaction-style t) 'normal))

;;; ----------------------------------------------------------------------

(defmacro pct (part wh)
  `(if (zerop ,wh) 0 (* 100.0 (/ ,part ,wh))))

(defmacro /-safe (dividend divisor)
  `(if (zerop ,divisor) 0 (/ ,dividend ,divisor)))

;;; ----------------------------------------------------------------------
;;; Instrumentation definitions...

(defclip number-of-bulldozers ()
  (:report-key "Number of bulldozers")
  (length (find-agent-by-name 'bulldozer :multiple-allowed t)))

(defclip area-burned ()
  (:report-key "Area burned (sq. km)")
   (send (real-world-firemap) :fire-state))

(defclip shutdown-time ()
  (:report-key "Shutdown time (hours)")
    (current-time))

;;; ----------------------------------------------------------------------
;;; Fire clips

(defclip fires-started ()
  (:report-key "Fires started")
  (let ((cnt 0))
    (map-over-fires (fire) (:delete-fire-frames nil)
       (incf cnt))
    (values cnt)))

(defclip fires-contained ()
  (:report-key "Fires contained")
  (mapcan #'(lambda (fire)
              (when (eq (f:get-value* fire 'status) 'under-control)
                (list fire)))
          (f:get-values* 'actual-fire 'instance+inv)))

(defclip number-of-fires-contained ()
  (:report-key "Fires extinguished")
  (length (fires-contained)))
```

```
;;; ----------------------------------------------------------------------
;;; Fireline clips

(defun expand-extent-in-all-directions-by (extent distance-in-meters)
  (let ((temp-point (point distance-in-meters distance-in-meters)))
    (extent (point-clip (point-difference (extent-upper-left extent)
                        temp-point))
            (point-clip (point-sum (extent-lower-right extent) temp-point)))))

(defun length-of-built-line-in-extent (extent)
  (let ((line-length 0))
    (dofiremap (point :upper-left (extent-upper-left extent)
                      :lower-right (extent-lower-right extent))
      (do-feature-edges (edge point (real-world-firemap) :edge-type :dynamic)
        (incf line-length (feature-edge-length edge))))
    (values line-length)))

(defclip r-factor ()
  (:report-key "R factor")
    (/-safe (total-fire-line-built) (total-perimeter)))

(defclip total-perimeter ()
  ()
  (reduce #'+ (fires-contained)
          :key #'(lambda (fire)
                   (fast-polyline-length
                     (fire-perimeter-polyline (fire-origin fire)
                                              *fire-perimeter-resolution*)
                     t))))

(defclip total-fire-line-built ()
  (:report-key "Fireline Built (meters)")
  (reduce #'+ (fires-contained)
          :key #'(lambda (fire)
                   (length-of-built-line-in-extent
                     (expand-extent-in-all-directions-by
                       (accurate-fire-extent (fire-center-of-mass fire)
                                             (point-on-polyline-furthest-from
                                               (fire-center-of-mass fire)
                                               (fire-boundary fire)
                                               nil))
                       *fire-sector-extension*)))))

;;; ----------------------------------------------------------------------
;;; Agent clips

(defclip agents-lost ()
```

```
  ()
  (mapcan #'(lambda (agent)
              (when (eq (f:get-value (send agent :self-frame) 'status) :dead)
                (list (name-of agent))))
          (all-agents)))

;;; "all-agent-instrumentation" is a mapping clip that collects information
;;; from its components, all of the Phoenix agents defined in this
;;; scenario.  Each of the :component clips is run for each agent found
;;; by the :mapping function.  Values returned by the :component clips
;;; are written out to the data file in sequence.

(defclip all-agent-instrumentation ()
  "Records the utilization of all the agents."
  (:components (agent-overall-utilization
                agent-cognitive-utilization
                agent-message-handling-time-pct
                agent-action-selection-time-pct
                agent-error-recovery-cost
                agent-error-recovery-percentage-of-cognitive-time
                number-of-frames-on-timeline)
   :map-function (cons
                  (find-agent-by-name 'fireboss)
                  (find-agent-by-name 'bulldozer :multiple-allowed t))))

(defclip agent-overall-utilization (agent)
  (:report-key "~a overall utilization")
  (pct (task-cumulative-cpu-time agent) (current-time)))

(defclip agent-cognitive-utilization (agent)
  (:report-key "~a cognitive utilization")
  (pct (phoenix-agent-cumulative-action-execution-time agent)
       (current-time)))

(defclip agent-message-handling-time-pct (agent)
  (:report-key "~a message handling pct")
  (pct (phoenix-agent-cumulative-message-handling-time agent)
      (current-time)))

(defclip agent-action-selection-time-pct (agent)
  (:report-key "~a action selection pct")
  (pct (phoenix-agent-cumulative-next-action-selection-time agent)
      (current-time)))

(defclip agent-error-recovery-cost (agent)
  (:report-key "~a error recovery cost")

  (f:using-frame-system ((name-of agent))
```

```
    (reduce #'+ (gather-recovery-method-instances (name-of agent))
            :key #'determine-recovery-cost)))

(defclip agent-error-recovery-percentage-of-cognitive-time (agent)
  (:report-key "~a ER % of cognitive time")
  (pct (agent-error-recovery-cost (name-of agent))
       (phoenix-agent-cumulative-action-execution-time agent)))

(defclip number-of-frames-on-timeline (agent)
  (:report-key "~a number of frames on timeline")
  (f:using-frame-system ((name-of agent))
    (unwind-protect
        (let ((cnt 0))
          (labels ((count-frame (frame)
                     (unless (f:get-value frame 'counted)
                       (incf cnt)
                       (f:put-value frame 'counted t)
                       (count-frames-after frame)
                       (count-frames-below frame)))
                   (count-frames-below (start-frame)
                     (dolist (frame (tl-has-components start-frame))
                       (count-frame frame)))
                   (count-frames-after (start-frame)
                     (dolist (frame (tl-next-actions start-frame))
                       (count-frame frame))))

        (dolist (frame (tl-has-start-actions (f:get-value 'initial-timeline
                                                           'instance+inv)))
          (count-frame frame)))
          (values cnt))
      (f:map-frames #'(lambda (frame)
                        (f:delete-all-values frame 'counted))))))

;;; ----------------------------------------------------------------------
;;; Fireboss clips

(defclip fireboss-instrumentation ()
  "Instrumentation for the fireboss."
  (:components (agent-total-envelope-time)
   :map-function (list (find-agent-by-name 'fireboss))))

(defclip agent-total-envelope-time (agent)
  (:report-key "~a total envelope time")
  (f:using-frame-system ((name-of agent))
    (reduce #'+
            (f:pattern-match #p(instance {f:value-in-hierarchy-of
                                           '(ako instance) 'plan-envelope})))))
```

```
;;; ------------------------------------------------------------------
;;; Bulldozer clips

(defclip bulldozer-instrumentation ()
  "Mapping clip mapping reflexes-executed :component over all bulldozers."
  (:components (reflexes-executed)
   :map-function (find-agent-by-name 'bulldozer :multiple-allowed t)))

(defclip reflexes-executed (agent)
  (:report-key "~a reflexes executed")
  (reduce #'+ (standard-agent-model-reflexes agent)
          :key #'reflex-execution-count))

(defclip count-of-deadly-object-in-path-messages ()
  (:enable-function (trace-message-patterns
                     '(:message-type :msg-reflex :type :error
                       :reason :deadly-object-in-path))
   :disable-function (untrace-message-patterns
                      '(:message-type :msg-reflex :type :error
                        :reason :deadly-object-in-path)))
  (message-pattern-count '(:message-type :msg-reflex :type :error
                           :reason :deadly-object-in-path)))

;;; ------------------------------------------------------------------
;;; Plan clips record which plan(s) was used during a trial to fight the fire.

(defun the-first-fire-started ()
  (first (last (f:get-values* 'actual-fire 'instance+inv))))

(defclip plan-to-contain-fire ()
  (:report-key "Plan to Contain Fire")

  (get-primitive-action
    (tl-entry-of-plan-to-contain-fire)))

(defun tl-entry-of-plan-to-contain-fire ()
  (f:using-frame-system ((name-of (find-agent-by-name 'fireboss)))
    (let (top-level-actions)
      (dolist (possible (f:get-values* 'act-deal-with-new-fire 'instance+inv))
        (when (equal (f:framer (the-first-fire-started))
                (variable-value 'fire :action possible))
          (push possible top-level-actions)))
      (f:get-value*
        (first (sort top-level-actions
                     #'>
                     :key #'(lambda (x)
                              (f:get-value* x 'creation-time))))
        'has-end-action))))
```

```
(defclip all-plans-to-contain-fire ()
  (:report-key "All plans to Contain Fire")
  (f:using-frame-system ((name-of (find-agent-by-name 'fireboss)))
    (mapcar #'(lambda (act-deal-with-our-fire)
                (get-primitive-action (f:get-value* act-deal-with-our-fire
                                                     'has-end-action)))
            (sort
              (mapcan #'(lambda (act-deal-with-new-fire)
                          (when (equal (f:framer (the-first-fire-started))
                                       (variable-value 'fire :action
                                          act-deal-with-new-fire))
                            (list act-deal-with-new-fire)))
                      (f:get-values* 'act-deal-with-new-fire 'instance+inv))
              #'< :key #'(lambda (x) (f:get-value* x 'creation-time))))))
```

B.1.3 Example from a Transportation Planning Simulation

This example comes from a baseline experiment in bottleneck prediction at a shipping port in a transportation planning simulator called TRANSSIM. On each day it predicts the occurrence of bottlenecks at a single port in the shipping network, then captures data about actual bottlenecks for later comparison. The experiment collects time-series data, a fragment of which is shown after the example. The summary output file (data collected after each trial) is also shown. Note the use of the keywords to `define-simulator` (`:schedule-function`, `:deactivate-scheduled-function`, `:seconds-per-time-unit` and `:timestamp`) that define the time-series clips.

```
;;; -----------------------------------------------------------------
;;; Clips for the collection of time series data over the course of a trial

(defun current-day ()
  "Return the current day of simulated time from 0."
  (/ (current-time) 24))

(defclip port-state-snapshot ()
  "Record state information for a port at the end of each day."
  (:output-file "port-state"
   :schedule (:period "1 day")
   :map-function (list (port 'port-1))
   :components (ships-en-route ships-queued ships-docked
                expected-ship-arrivals predicted-queue-length)))

(defclip ships-en-route (port)
  "Record the number of ships en route to a port."
  ()
  (length (apply #'append (mapcar 'contents (incoming-channels port)))))

(defclip ships-queued (port)
  "Record the number of ships queued at a port."
  ()
  (length (contents (docking-queue port))))

(defclip ships-docked (port)
  "Record the number of ships docked at a port."
  ()
  (length (apply #'append (mapcar 'contents (docks port)))))

(defclip expected-ship-arrivals (port) ()
  (progn
    (update-prediction)
    (let ((prediction-units (find-units 'prediction '(port-model) :all)))
      (float (/ (round (* 10000
                        (expected-value (first prediction-units)))) 10000)))))
```

```
(defclip port-predicted-value (port) ()
  (let ((prediction-units (find-units 'prediction '(port-model) :all)))
    (generate-prediction (expected-value (first prediction-units)))))

(defclip port-actual-change (port) ()
  (let ((prediction-units (find-units 'prediction '(port-model) :all)))
    (length
      (set-difference
        (ships-previously-in-port port 0)
        (ships-previously-in-port
          port (days-in-future (first prediction-units)))))))

(defclip predicted-queue-length (port) ()
  (let ((prediction-units (find-units 'prediction '(port-model) :all)))
    (pred-queue-length (first prediction-units))))

;;; ----------------------------------------------------------------------
;;; Clips for the collection of data at the end of a trial

(defclip prediction-score ()
  (:output-file "score"
   :components (score-for-day)
   :map-function '(2 5)))

(defclip score-for-day (day)
 ()
 (compute-score-for-day day))

;;; ----------------------------------------------------------------------
;;; Experiment and simulator definitions

(define-simulator transsim
  :system-name "TransSim"
  :start-system (simulate nil)
  :reset-system reset-transsim-experiment
  :schedule-function schedule-function-for-clips
  :deactivate-scheduled-function transsim::reset
  :seconds-per-time-unit 3600
  :timestamp current-day)

(define-experiment test-experiment ()
  :simulator transsim
  :instrumentation (prediction-score port-state-snapshot)
  :variables ((prediction-threshold in '(0.6 0.75 0.9))
              (eta-variance-multiplier in '(0.2 0.4 0.6))
              (prediction-point in '(5)))
  :after-trial
    ;; Other options here include building CLASP datasets,
```

```
    ;; exporting to some database, massaging the data or some
    ;; combination fo these.
    (write-current-experiment-data))

;;; ------------------------------------------------------------------
;;; Utilities

(defun reset-transsim-experiment (prediction-threshold
                                  eta-variance-multiplier
                                  prediction-point)
  (setf *prediction-threshold* prediction-threshold)
  (format t "~&Prediction threshold = ~a" prediction-threshold)
  (setf *eta-variance-multiplier* eta-variance-multiplier)
  (format t "~&ETA variance multiplier = ~a" eta-variance-multiplier)
  (setf *prediction-point* prediction-point)
  (format t "~&Prediction point = ~a" prediction-point)
  (initialize-simulation))

(defun schedule-function-for-clips (function time period name)
  (if period
    (transsim::schedule-recurring-event (transsim::event-actuator :external)
                                        :function function
                                        :time time
                                        :period period
                                        :type (or name :instrumentation))
    (transsim::schedule-event (transsim::event-actuator :external)
                              :function function
                              :time time
                              :type (or name :instrumentation))))

(defun rexp (&key &optional number-of-trials)
  (run-experiment 'test-experiment :output-file "ed-buffer:out"
                                   :number-of-trials number-of-trials))
```

Shown below is the summary output file produced for 9 trials. This file is in CLASP format. It begins with an informative header string, followed by a series of strings, each of which is a column name in the data table. Rows of the table follow, stored as lists. Each list contains one element per column name.

```
"
*********************************************************************
****
**** Experiment: Test-Experiment
**** Machine: Miles
**** TransSim version: Unknown
**** Date: 10/1/93 11:26
**** Scenario: None
**** Script-name: None
```

```
**** First trial number: 1
**** Last trial number: 9
**** Number of trials: 9
**** Max trial length: Unknown hours
**********************************************************************

The key follows:"
"Trial"
"Prediction-Threshold"
"Eta-Variance-Multiplier"
"Prediction-Point"
"Score-For-Day 2"
"Score-For-Day 5"
(1 0.75 0.2 5 100 100 )
(2 0.75 0.3 5 100 100 )
(3 0.75 0.4 5 100 100 )
(4 0.8 0.2  5 100 100 )
(5 0.8 0.3  5 100 100 )
(6 0.8 0.4  5 100 100 )
(7 0.85 0.2 5 100 100 )
(8 0.85 0.3 5 100 100 )
(9 0.85 0.4 5 100 100 )
```

This is a fragment of the time-series data (from the first trial only) showing the flat file structure produced by component and time-series clip relationships.

```
"
*********************************************************************
****
**** Experiment: Test-Experiment
**** Machine: Miles
**** TransSim version: Unknown
**** Date: 10/5/93 15:17
**** Scenario: None
**** Script-name: None
**** First trial number: 1
**** Last trial number: 12
**** Number of trials: 12
**** Max trial length: Unknown hours
*********************************************************************

The key follows:"
"Trial"
"Timestamp"
"Prediction-Threshold"
"Eta-Variance-Multiplier"
"Prediction-Point"
"Ships-En-Route port-1"
```

```
"Ships-Queued port-1"
"Ships-Docked port-1"
"Expected-Ship-Arrivals port-1"
"Predicted-Queue-Length port-1"

(1 0 0.6 0.2 5 0 0 0 0.0 0 )
(1 1 0.6 0.2 5 0 0 0 0.0 0 )
(1 2 0.6 0.2 5 0 0 0 0.0 0 )
(1 3 0.6 0.2 5 0 0 0 0.0 0 )
(1 4 0.6 0.2 5 0 0 0 0.0 0 )
(1 5 0.6 0.2 5 2 0 0 0.0597 0 )
(1 6 0.6 0.2 5 2 0 0 0.3357 0 )
(1 7 0.6 0.2 5 2 0 0 0.7643 0 )
(1 8 0.6 0.2 5 2 0 0 0.9961 0 )
(1 9 0.6 0.2 5 2 0 0 0.9977 0 )
(1 10 0.6 0.2 5 4 0 0 0.683 0 )
(1 11 0.6 0.2 5 3 0 1 0.2361 0 )
(1 12 0.6 0.2 5 3 0 1 0.6389 1 )
(1 13 0.6 0.2 5 3 0 1 1.4062 1 )
(1 14 0.6 0.2 5 3 0 1 1.9885 1 )
(1 15 0.6 0.2 5 3 0 0 2.071 1 )
(1 16 0.6 0.2 5 3 0 0 1.5729 0 )
(1 17 0.6 0.2 5 1 1 1 0.5053 1 )
(1 18 0.6 0.2 5 1 1 1 0.8974 2 )
(1 19 0.6 0.2 5 1 1 1 0.9996 2 )
(1 20 0.6 0.2 5 1 1 0 1.0 1 )
(1 21 0.6 0.2 5 1 0 1 1.0 1 )
(1 22 0.6 0.2 5 0 1 1 0.0 1 )
(1 23 0.6 0.2 5 0 1 1 0.0 1 )
(1 24 0.6 0.2 5 0 1 0 0.0 0 )
(1 25 0.6 0.2 5 2 0 1 0.1331 0 )
(1 26 0.6 0.2 5 2 0 1 0.355 0 )
(1 27 0.6 0.2 5 2 0 1 0.6276 1 )
(1 28 0.6 0.2 5 2 0 0 0.946 0 )
(1 29 0.6 0.2 5 4 0 0 1.2677 0 )
(1 30 0.6 0.2 5 4 0 0 2.0143 1 )
(1 31 0.6 0.2 5 4 0 0 2.8832 2 )
(1 32 0.6 0.2 5 4 0 0 3.3096 2 )
(1 33 0.6 0.2 5 4 0 0 2.4576 1 )
(1 34 0.6 0.2 5 2 1 1 1.9172 3 )
(1 35 0.6 0.2 5 1 2 1 0.9961 3 )
```

References

[1] George W. Bohrnstedt and David Knoke. *Statistics for Social Data Analysis*. F. E. Peacock Publishers, Itasca, Illinois, second edition, 1988.

[2] James V. Bradley. *Distribution-Free Statistical Tests*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1968.

[3] Paul R. Cohen. *Empirical Methods in Artificial Intelligence*. MIT Press, forthcoming.

[4] Jay L. Devore. *Probability and Statistics for Engineering and the Sciences*. Brooks/Cole Publishing Company, Monterey, California, first edition, 1982.

[5] John E. Freund. *Modern Elementary Statistics*. Prentice-Hall, Englewood Cliffs, NJ, seventh edition, 1988.

[6] David M. Hart and Paul R. Cohen. Predicting and explaining success and task duration in the phoenix planner. In *Proceedings of the First International Conference on AI Planning Systems*, pages 106–115. Morgan Kaufmann, 1992.

[7] Adele E. Howe and Paul R. Cohen. Responding to environmental change. In *DARPA Workshop on Innovative Approaches to Planning, Scheduling and Control*, pages 85–92. Morgan Kaufmann, 1990.

[8] Geoffrey Keppel. *Design and Analysis: A Researchers Handbook*. Prentice-Hall, Englewood Cliffs, NJ, seventh edition, 1973.

[9] Edward W. Minium. *Statistical Reasoning in Psychology and Education*. John Wiley and Sons, 1978.

[10] Elazar J. Pedhazur. *Multiple Regression in Behavioral Research*. Holt, Rinehart and Winston, 1973.

[11] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, 1988.

[12] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, second edition, 1992.

Index