Discrete Finite State Markov Chains in `Lisp-Stat`

B. Narasimhan Brett I. MacAlpine *

Division of Science and Mathematics
University of Minnesota at Morris
Morris, MN 56267

January 31, 2021

DRAFT COPY

Abstract

This paper describes how `Lisp-Stat` can be used to simulate discrete finite state Markov Chains. `Lisp-Stat`, developed by Luke Tierney of the University of Minnesota, is a powerful object-oriented programming language with statistical capabilities.

The software we have developed will graphically simulate a Markov chain using the `Lisp-Stat` language. It can be used to illustrate the validity of several important theoretical results concerning the properties of a Markov chain. In this paper, we give a complete description of our design and the code used to implement the design. A beneficial side-effect of this detailed description is that some of the objects we have designed may be used in a "plug-and-chug" manner by programmers in other situations.

1 Introduction

This paper describes in detail the software we have written for graphically simulating some Markov Chains. The original motivation for this work came from a question posed by the second author while a student in a course on Probability and Stochastic Processes taught by the first author. Therefore, we suspect that

*Brett Ian MacAlpine is a senior in Mathematics at UMM. This research project was supported by the University of Minnesota Undergraduate Research Opportunities Program.

this software will probably find its best use as a pedagogical tool in introductory courses on Probability and Stochastic Processes.

We had some objectives in writing such a detailed paper. We hope that such detailed description will enable people to use our software effectively and isolate bugs, if any; that this would provide first-time users with a non-trivial example of the power of `Lisp-Stat`[8] and the object-oriented paradigm inherent in it; and that other users can build upon our design and add features or other methods that they may find lacking, in which case, we would be glad to hear about such.

This software is free and can be obtained by anybody from `statlib` by anonymous `ftp` or by e-mail. A sample `ftp` session is given below.

```
% ftp lib.stat.cmu.edu
Connected to TEMPER.STAT.CMU.EDU.
220- temper.stat.cmu.edu FTP server ready.
220 StatLib users please send e-mail address as password.
Name (lib.stat.cmu.edu:naras): statlib
331 Please send your email address, for bug reports and logs.
Password: (Type your e-mail address here)
230 Guest login ok, access restrictions apply.
ftp> cd xlispstat
250 CWD command successful.
ftp> bin
200 Type set to I.
ftp> get markov.shar
ftp> quit
221 Goodbye.
%
```

For more information on the `statlib` server, see the list of answers to "Frequently Asked Questions about S" maintained by Shanti Gomatam[2].

This paper is organized as follows. In section 3, we describe in detail each of the modules that make up the software. In section 4, we present three applications that demonstrate the use of our software. Finally, we make some remarks and touch upon future developments in section 5.

2 Preliminaries

We begin with a brief overview of Markov chains. For an excellent introduction, the reader is referred to Feller[1] or Ross[5].

A *stochastic process* is a collection of random variables, $\{X_t,\ t \in T\}$. The set T, the index set, is commonly called *time*. In this paper, we only consider the situation where T is a countable set; specifically, we shall use $T = \mathbf{N}$, the set of natural numbers. Thus, $\{X_k,\ k = 0, 1, 2, \ldots\}$ will denote the stochastic

process. Such a process is called a *discrete time* process. The set of all possible values of the random variables X_k is called the *state space*of the process. We again restrict ourselves to finite or countable state spaces and denote the states by integers. If $X_k = i$, then we say the process is in state i at time k.

A *Markov chain* is a special kind of stochastic process. It is a stochastic process such that:

$$P\{X_{k+1} = j | X_k = i, X_{k-1} = i_{k-1}, \ldots, X_1 = i_1, X_0 = i_0\} = P_{ij}$$

for states $i_0, i_1, \ldots, i_{k-1}, i, j$ and $k \geq 0$. This can be interpreted as saying that the probability that the process will go to state j given its past history (i.e. the states it has been in for all times before k) is independent of those states, and depends only on the the state it is presently occupying.

Whenever the process is in state i, there is a probability P_{ij} that the process will go from state i to state j. These probabilities can be specified by means of a *transition matrix* $\mathbf{P} = [P_{ij}]$. The ith element in the jth row of this matrix $\mathbf{P}$ is the value P_{ij}:

$$\mathbf{P} = \begin{bmatrix} P_{00} & P_{01} & P_{02} & \ldots \\ P_{10} & P_{11} & P_{12} & \ldots \\ \vdots & \vdots & \vdots & \\ P_{i0} & P_{i1} & P_{i2} & \ldots \\ \vdots & \vdots & \vdots & \end{bmatrix}$$

Because the process must make some sort of transition (perhaps even to the same state), the value $\sum_{j=0}^{\infty} P_{ij}$, which is the sum of the probabilities of all possibilities when the process is currently in state i, must equal 1, for every i.

A *discrete time finite state Markov chain* is a markov chain with a finite state space and discrete time. We shall denote the states by integers $0, 1, \ldots, n$, for some integer n.

3 Code Description

The program `dmc.lsp` is designed to graphically illustrate and run a discrete finite state Markov Chain process. The user provides a transition matrix and an initial state. The program creates a graphical window with buttons to run the Markov Chain. There are two kind of buttons: oval or circular buttons representing states and rectangular buttons that perform some actions. Several additional mouse controls are available. Figure 1 shows the the appearance of such windows. Many routine statistics are available, including sample path plots, number of visits to each state, etc.

We now describe the complete design of our objects and algorithms. The task of describing our code would be much easier and neater if we had a literate programming environment like Knuth's *WEB*[6] for `lisp` (although there

is John Ramsdell's SchemeWEB for scheme[3]). An alternative is to use Dorai Sitaram's SLaTeX[7] package which is geared towards both Scheme and Common Lisp; however, we were unable to get it going using the subset of Common Lisp that `Lisp-Stat` uses. Therefore, we have chosen to break up the code description into fragments that perform a particular function. The indentation of the code is preserved to help the user follow the scope of variables; this is necessary in understanding the logic of a large chunk of code. Where necessary, an indentation level number is provided at the beginning of each such fragment for the benefit of the reader; if no indentation level number is provided, then the level is zero.

3.1 Description of `dmc.lsp`

The file `dmc-proto.lsp` contains the code that creates the markov chain object. It also includes the associated methods for the markov chain object.

```
(require "dmc-opts")
(require "buttons")
(require "stochast")
(require "states")
(provide "dmc")
```

These first few lines lines make sure that the auxiliary code in `dmc-opts.lsp` (see section 3.5), `states` (see section 3.2), `buttons.lsp` (see section 3.3), and `stochast.lsp` (see section 3.4) , which is necessary for `dmc.lsp` to work sanely, is loaded and available when `dmc.lsp` is loaded.

```
(defproto dmc-proto '(initial-state state-objs button-objs
                                    current-state time history
                                    sample-paths)
  nil graph-window-proto)
```

We define a new object `dmc-proto` with some slots for holding its data. The slots are described below.

`initial-state` Holds the state in which the process will start,

`state-objs` Holds a list of all the state button objects which are manipulated by the `dmc-proto` object,

`button-objs` Holds a list of control buttons objects for the state-object,

`current-state` Holds the state in which the markov chain is currently,

`time` Holds the current time, in ticks,

`history` Holds the name of a file into which history is recorded. It is `nil` if no such recording is requested, which is the default.

This object inherits from the `Lisp-Stat` prototype `graph-window-proto` and therefore has all the methods of `graph-window-proto` available to it. Some of these inherited methods enable us to paint particular areas of a window, draw lines, points, etc. Thus, we are spared from dealing with these tedious details—a good example of the power of object-oriented paradigm.

3.1.1 The `:isnew` method

When any object is designed, an `:isnew` method needs to be defined for the object outlining the tasks to be done when an instance of the object is created. Note that many instances of an object prototype can be created in one session. We now describe the `:isnew` method for `dmc-proto`.

```
(defmeth dmc-proto :isnew (transition-matrix initial &optional
                           title)
  "Method args: (transition-matrix initial &optional title)
  Returns an instance of the dmc-proto object given a
  transition matrix.  Initial state is initial. Title
  is used for title if given."
```

The mandatory arguments to the `:isnew` method are a transition-matrix of object type `matrix` and an integer for the initial state. An optional `title` may be specified to distinguish various instances of the `dmc-proto` object. The string that is specified as the last argument provides some basic help for the user who wishes to know more the `:isnew` method. In particular, it tells the user what arguments are necessary for the method to be invoked. For example, in `Lisp-Stat`, after one loads the file `dmc.lsp`, the help facility may be used as follows:

```
> (send dmc-proto :help :isnew)
:ISNEW
Method args: (transition-matrix initial &optional title)
  Returns an instance of the dmc-proto object given a
  transition matrix.  Initial state is initial. Title
  is used for title if given.
NIL
```

We shall be content with this one example and refrain from discussing the help string hereafter. To save space, we have omitted these help strings in the otherwise verbatim reproduction of our code.

```
;;; Indentation level 1
  (unless (stochastic-matrixp transition-matrix)
          (error
           "Transition matrix is not a stochastic matrix!"))
  (setf (slot-value 'time) 0)
```

```
  (if title
      (send self :title title)
    (send self :title "Discrete Markov Chain"))
  (setf (slot-value 'initial-state) initial)
  (setf (slot-value 'current-state) initial)
```

The first line above checks whether the transition matrix is a stochastic matrix using a function defined in the file `stochast.lsp`. If the matrix is not so, an error is signalled. Otherwise, the values for `initial-state` and `current-state` are recorded in the appropriate slots. The clock is reset to zero. If a title is given, it is used; otherwise, a default title DISCRETE MARKOV CHAIN is given.

```
;;; Indentation level 1
  (let* ((n (select (array-dimensions transition-matrix) 0))
         (rows (row-list transition-matrix))
         (tmp nil))
    (dotimes (j n)
             (setf tmp (append tmp
                               (list
                                (send state-proto :new
                                      (format nil "~d" j) j
                                      (select rows j))))))
    (setf (slot-value 'state-objs) tmp))
```

Once the transition-matrix is available, the above code creates a list of n state objects, where n is the number of states. Each state stores its own probability vector which means that the transition matrix can be discarded after this point. Note that a `:new` message is sent to the prototype `state-proto` (see section 3.2) n times to create n individual instances of the `state-proto` object. The list of state objects is finally stored in the slot `state-objs`.

```
;;; Indentation level 1
  (let* ((n (send self :no-of-states))
         (ta (send self :text-ascent))
         (td (send self :text-descent))
         (col-sep (send self :text-width "mm"))
         (row-sep (+ ta td))
         (bh (round (* 1.5 (+ ta td))))
         (bw1 (send self :text-width " Reset "))
         (bw2 (send self :text-width " run-dmc "))
         (bw2a (send self :text-width " # "))
         (bw3 (send self :text-width  " Time: mmmmmmmmmm "))
         (state-width (send self :text-width
                            (format nil "m~dm" n)))
         (c (floor (sqrt n)))
```

```
      (r (floor (/ n c)))
      (wx (+ (* 4 col-sep) bw1 bw2 bw2a bw3))
      (cols-width (+ (* c (+ col-sep state-width))
                     col-sep)))
  (when (< cols-width wx)
       (setf c (floor (/ (- wx col-sep)
                         (+ col-sep state-width))))
       (setf r (floor (/ n c))))
  (setf wx (max wx cols-width))
```

It may be wise to look at figure 1 to understand this piece of code. The size of our graph-window and the layout of the state-buttons are calculated. Thus, it is necessary to have information about the ascent and descent of the characters in the font currently being used in the window. The idea is that the window should be as square as possible, thus occupying minimum area of the screen, but at the same time, it should accommodate the control buttons that sit on the first row of window. The variables `c` and `r` are the number of columns, number of rows respectively, in which the state buttons will be arrayed. The variable dmc`wx` is the window width. Some amount of row and column separation is provided for better appearance.

```
;;; Indentation level 2
  (setf (slot-value 'button-objs)
        (list
         (send button-obj-proto :new
               "Reset"
               (list col-sep row-sep bw1 bh)
               #'(lambda () (send self :reset)))
         (send button-obj-proto :new
               "run-dmc"
               (list (+ col-sep bw1 col-sep) row-sep bw2 bh)
               #'(lambda () (send self :run-dmc)))
         (send button-obj-proto :new
               " # "
               (list (+ col-sep bw1 col-sep bw2)
                     row-sep bw2a bh)
               #'(lambda ()
                   (let ((k (get-value-dialog
                             "Number of steps to run?"
                             :initial 1)))
                     (if k
                         (send self :run-
                               (select k 0))))))
         (send button-obj-proto :new
               (format nil "Time: ~d" (slot-value 'time))
```

```
                (list (- wx col-sep bw3) row-sep bw3 bh)
                #'(lambda ())))))
```

The positions of each of the control buttons is determined and objects are created for those buttons. The first two buttons `Reset` and `run-dmc` perform the tasks suggested by their names. There is also a mini-button created that goes with the `run-dmc` button. This button can be used to run the markov chain for a specified number of steps instead of just once. The user is queried for the number of steps in a dialog box. A default of 1 step is supplied. The `time` button is somewhat special. It has no action to perform and its title is dynamic; that is, as time ticks by, its title keeps changing to display the new time. At present, it can display a maximum of a billion time ticks, which is not altogether inadequate.

```
;;; Indentation level 2
    (let* ((state-height (round (* 3 (+ ta td))))
           (controls-height (+ row-sep
                               (round (* 1.5 (+ ta td)))))
           (wy (+ (if (eql (* c r) n)
                      (* r (+ row-sep state-height))
                    (* (+ r 1)
                       (+ row-sep state-height))) row-sep))
           (tmp (slot-value 'state-objs)))
      (send self :size wx (+ wy controls-height))
      (call-next-method)
      (send self :delete-method :do-motion)
```

Next, the height of the window is determined and the window is informed of its new size. Since we wouldn't want anything done when the mouse is moved over the window, we trash the `:do-motion` method. The `call-next-method` makes sure that the inherited `:isnew` method of `graph-window-proto` gets executed when the `:isnew` method is invoked for `dmc-proto`. This is what actually creates the window (recall that our prototype inherited from `graph-window-proto`).

```
;;; Indentation level 2
      (dotimes (i n)
               (let* ((y (+
                          (* (floor (/ i c))
                             (+ state-height row-sep))
                          row-sep))
                      (x (+ (* (mod i c)
                               (+ state-width col-sep))
                            col-sep)))
                 (send (select tmp i)
                       :loc-and-size
```

```
                    (list x (+ y controls-height)
                          state-width state-height))))))
```

It is time now to determine the locations for the state buttons. The state buttons, which are available as a list in the variable `tmp` are given their locations which they store in their own respective slots. Later, we shall see that by defining a `:redraw` method (see section 3.2.7) for the state buttons, we can let it draw itself in the window, thus freeing ourselves from those decisions for now.

```
;;; Indentation level 1
  (send (select (slot-value 'state-objs) initial)
        :you-are-current-state self)
  (send self :add-some-menu-items))
```

The initial state object is sent a message that it is the current state which makes it take the appropriate action. Finally, some additional items are installed in the menu for the window; see the description of the `:add-some-menu-items` for details. This concludes the `:isnew` method for our `dmc-proto`.

3.1.2 The `:current-state-obj` method

```
(defmeth dmc-proto :current-state-obj ()
  (select (slot-value 'state-objs)
          (slot-value 'current-state)))
```

This method returns the object representing the current state. Note that the current state number and the current state object are different.

3.1.3 The `:resize` method

```
(defmeth dmc-proto :resize ()
  (let* ((but-objs (slot-value 'button-objs))
         (wsize (send self :size))
         (n (send self :no-of-states))
         (ta (send self :text-ascent))
         (td (send self :text-descent))
         (col-sep (send self :text-width "mm"))
         (row-sep (+ ta td))
         (bh (round (* 1.5 (+ ta td))))
         (bw1 (send self :text-width " Reset "))
         (bw2 (send self :text-width " run-dmc "))
         (bw2a (send self :text-width " # "))
         (bw3 (send self :text-width  " Time: mmmmmmmmmm "))
         (state-width (send self :text-width
                            (format nil "m~dm" n)))
         (wx (max (first wsize) (+ (* 4 col-sep)
```

```
                                bw1 bw2 bw2a bw3)))
        (c (floor (/ (- wx col-sep)
                     (+ col-sep state-width))))
        (r (floor (/ n c))))

    (send (select but-objs 0) :loc-and-size
          (list col-sep row-sep bw1 bh))
    (send (select but-objs 1) :loc-and-size
          (list (+ col-sep bw1 col-sep) row-sep bw2 bh))
    (send (select but-objs 2) :loc-and-size
          (list (+ col-sep bw1 col-sep bw2) row-sep bw2a bh))
    (send (select but-objs 3) :loc-and-size
          (list (- wx col-sep bw3) row-sep bw3 bh))
```

Whenever a window is resized, the window layout would need to be adjusted. Otherwise, the same layout that we initially created would be used every time the resizing occurs. The `:resize` method adjusts the number of rows and columns in which the state buttons are arrayed while making sure that the window does not become too small or too large. This `:resize` method has a lot in common with the `:isnew` method (see section 3.1.1). The main difference is that now we don't have to determine the window size ourselves; the resizing of the window by the user forces a window dimension on us. However, we have to guard against the window being too narrow for the control buttons to fit. Therefore, there is a minimum size beyond which the window cannot be made any smaller. We ensure this when we define the variable `wx` representing the window width. Once the window width is known, the number of columns `c` and the number of rows `r` are determined for the layout. The control buttons may now have new locations in the window, and so they must be sent a `:loc-and-size` message with the appropriate coordinates.

```
;;Indentation level 1
    (let* ((state-height (round (* 3 (+ ta td))))
           (controls-height (+ row-sep
                               (round (* 1.5 (+ ta td)))))
           (wy (+ (if (eql (* c r) n)
                      (* r (+ row-sep state-height))
                    (* (+ r 1) (+ row-sep state-height)))
                  row-sep))
           (tmp (slot-value 'state-objs)))
      (send self :size wx (+ wy controls-height))
      (dotimes (i n)
               (let* ((y (+
                          (* (floor (/ i c))
                             (+ state-height row-sep))
                          row-sep))
```

```
            (x (+ (* (mod i c)
                     (+ state-width col-sep))
                  col-sep)))
       (send (select tmp i)
             :loc-and-size
             (list x (+ y controls-height)
                   state-width state-height)))))))
```

After the control button locations have been determined, the state button locations are calculated. Then each of the state objects is also sent a `:loc-and-size` message with the new locations. Note that the window also cannot be reduced to a very small height either since the size is reset to a value that will accommodate both the control buttons and the state buttons.

3.1.4 The `:do-click` method

```
(defmeth dmc-proto :do-click (x y m1 m2)
  (catch 'done
    (dolist (obj (slot-value 'button-objs))
    (when (send obj :my-click x y m1 m2)
  (send obj :do-click)
  (throw 'done nil)))
    (dolist (obj (slot-value 'state-objs))
    (when (send obj :my-click x y m1 m2)
  (send obj :do-click self)
  (throw 'done nil)))))
```

Once the window is created, the user may click on certain buttons installed in the window to invoke various actions. When such a click occurs, the code has to determine which button was the focus of the click. This `:do-click` method is invoked whenever such a click occurs; the `Lisp-stat` system provides this. It is the responsibility of the programmer to pack into this `:do-click` method whatever needs to be done. In our case, we have to determine two things: was the click on a control button? or was it on on a state button? The code first polls the control buttons and then the state buttons to see if any one of them owns up to the click. If so, that object is sent a message to perform the click-action. If none of the objects owns up to the click, nothing is done. Notice that if any of the buttons owns up to the click, none of the remaining ones are polled. The `catch` and `throw` construct ensure this fact.

3.1.5 The `:no-of-states` method

```
(defmeth dmc-proto :no-of-states ()
  (length (slot-value 'state-objs)))
```

This is just an accessor method that returns the number of states in the markov chain. An accessor method for a slot is one that returns the value stored in that particular slot.

3.1.6 The `print-transition-matrix` method

```
(defmeth dmc-proto :print-transition-matrix ()
  (print-matrix (send self :transition-matrix)))
```

Using the `Lisp-stat` function `print-matrix`, the transition matrix is printed in a nice form. However, it should be remarked that when a markov chain has a large number of states, it is hard to make sense of this printout.

3.1.7 The `:transition-matrix` method

```
(defmeth dmc-proto :transition-matrix ()
  (let ((n (send self :no-of-states))
        (tmp nil))
    (dolist (obj (slot-value 'state-objs))
            (setf tmp
                  (append tmp
                          (coerce
                           (send obj :prob-vec) 'list))))
    (matrix (list n n) (coerce tmp 'list))))
```

This method returns the transition matrix. Recall that the transition matrix is not stored as a matrix anywhere. However, since each state object has a copy of its transition probability vector, the transition matrix can be recreated. This method collects all such probability vectors and builds them into a matrix. We don't envision this method to be used with great frequency.

3.1.8 The `:run-dmc` method

```
(defmeth dmc-proto :run-dmc (&optional k)
  (if k
      (dotimes (j k)
               (send self :goto-next-state)
               (pause *run-step-delay*))
    (send self :goto-next-state)))
```

This method runs the Markov Chain k steps, or if k is not supplied as an optional argument, one step. In the latter case, the markov chain merely goes to the next state. It uses the `:goto-next-state` method (see section 3.1.10) to accomplish the transition.

3.1.9 The `:run-dmc-silently` method

```
(defmeth dmc-proto :run-dmc-silently (k)
  (let* ((state-objs (slot-value 'state-objs))
         (cs (slot-value 'current-state))
         (obj (select state-objs cs))
         (next-obj nil))
    (case (slot-value 'history)
         (nil (dotimes (j k)
                       (setf (slot-value 'time)
                             (+ (slot-value 'time) 1))
                       (setf next-obj
                             (select state-objs
                                     (send obj :next-state)))
                       (send next-obj :slot-value
                             'no-of-visits
                             (+
                              (send next-obj
                                    :slot-value 'no-of-visits)
                              1))
                       (setf obj next-obj)))
         (t (dotimes (j k)
                     (setf (slot-value 'time)
                           (+ (slot-value 'time) 1))
                     (format *history-file-handle* "~d~%"
                             (send obj :next-state))
                     (setf next-obj
                           (select state-objs
                                   (send obj :next-state)))
                     (send next-obj :slot-value 'no-of-visits
                           (+
                            (send next-obj
                                  :slot-value 'no-of-visits)
                            1))
                     (setf obj next-obj))))
    (setf (slot-value 'current-state)
          (send obj :slot-value 'state-no)))
  (send (select (slot-value 'button-objs) 3)
        :title (format nil "Time: ~d" (slot-value 'time)))
  (send self :redraw))
```

This method runs the markov chain for k steps silently. This is useful when the user wants to run the markov chain for a large number of steps without watching its complete history. If history recording is on, then all the intermediate state visits are recorded in the appropriate file. Each state object dutifully increments

the `no-of-visits` slot whenever it is visited. Finally, when the loop is over, the window is redrawn to show the new time and the new current state. This method is truly silent in that if sample paths window is on, it is not updated.

3.1.10 The `:goto-next-state` method

```
(defmeth dmc-proto :goto-next-state ()
  (let* ((time (slot-value 'time))
         (state-objs (slot-value 'state-objs))
         (cs (slot-value 'current-state))
         (obj (select state-objs cs))
         (next-state (send obj :next-state))
         (next-obj (select state-objs next-state))
         (w (slot-value 'sample-paths)))
    (when w
          (send w :add-lines
                (list (list time
                            (+ time 1)) (list cs next-state)))
          (if (> *time-window* 0)
              (send w :range 0
                    (max 0 (- time *time-window* -1))
                    (max 50 (+ time 1)))
            (send w :adjust-to-data)))

    (setf (slot-value 'current-state) next-state)
    (send self :tick-time)

    (send obj :you-are-not-current-state self)
    (send next-obj :you-are-current-state self))
  (when (slot-value 'history)
        (format *history-file-handle* "~d~%"
                (slot-value 'current-state))))
```

This method sends the Markov chain to the next state. There are a few things that need to be done: the clock has to be incremented, the next state has to be determined and finally the markov chain has to be sent to the next state. The user might have requested sample paths be drawn, in which case the sample paths window must be updated. The sample paths window can display a fixed time-frame at a time, or accumulate the complete history of the process, which is the case if the global variable `*time-window*` is zero. In the latter situation, the user can use a scroll-bar in the bottom of the window to view the whole history. If `*time-window` is non-zero, then the history for the last `*time-window*` units of time is displayed. This is accomplished by adjusting the range for the abscissa to give the appearance of new lines appearing on the right and old ones

disappearing from the left. If history recording was requested, the code above makes sure it records the transitions in the specified file.

3.1.11 The `:time` method

```
(defmeth dmc-proto :time (&optional value)
  (if value
      (let ((obj (select (slot-value 'button-objs) 3)))
        (setf (slot-value 'time) 0)
        (send obj :title
              (format nil "Time: ~d" (slot-value 'time)))
        (send obj :redraw self))
    (slot-value 'time)))
```

This method retrieves the current time, or sets it if `value` is supplied. When time is set, it must be ensured that the clock button in the window is displaying the correct time. Therefore, this method send the button a new title with the current time included in it.

3.1.12 The `:tick-time` method

```
(defmeth dmc-proto :tick-time ()
  (setf (slot-value 'time) (+ (slot-value 'time) 1))
  (let ((obj (select (slot-value 'button-objs) 3)))
    (send obj :title
          (format nil "Time: ~d" (slot-value 'time)))
    (send obj :redraw self)))
```

This method increments the time by 1. Again, it makes certain the the new time is shown on the clock button in the window.

3.1.13 The `:current-state` method

```
(defmeth dmc-proto :current-state ()
  (slot-value 'current-state))
```

This is just an accessor method for the slot `current-state`. The returned value is an integer between 0 and $n-1$, where n is the number of states.

3.1.14 The `:describe` method

```
(defmeth dmc-proto :describe ()
  (let ((time (slot-value 'time)))
    (format t "-----------------------------------------~%")
    (format t "~%State         No of visits     Proportion~%")
    (format t "-----------------------------------------~%")
```

```
    (dolist (obj (slot-value 'state-objs))
            (if (> time 0)
                (send obj :describe time)
              (send obj :describe)))
  (format t "-----------------------------------------~%")))
```

This method prints the statistics for each state in the Markov chain. It prints how many times each state has been visited and the proportion of visits if the time is greater than 0. To make the summary compact, the number of visits is not printed for those states yet unvisited.

3.1.15 The `:reset` method

```
(defmeth dmc-proto :reset ()
  (send self :time 0)
  (dolist (obj (slot-value 'state-objs))
          (send obj :reset))
  (if (slot-value 'sample-paths)
      (send (slot-value 'sample-paths) :clear))
  (when (slot-value 'history)
        (send self :record-history nil))
  (let ((obj (send self :current-state-obj)))
    (setf (slot-value 'current-state)
          (slot-value 'initial-state))
    (send (send self :current-state-obj)
          :you-are-current-state self)
    (send obj :you-are-not-current-state self)))
```

The Markov chain is reset by this method. Time is set to zero. The statistics for each state are reset by the `:reset` method for state items. If the `sample-paths` window is currently displayed, it is reset with the `:clear` method. If the `history` slot is open, it is closed. Otherwise it is set to `nil`. The value of the current state is returned to the value of the initial state, which is then highlighted by `:you-are-current-state`. The former current state is returned to its normal colors by `:you-are-not-current-state`.

3.1.16 The `:redraw` method

```
(defmeth dmc-proto :redraw ()
  (dolist (obj (slot-value 'button-objs))
          (send obj :redraw self))
  (dolist (obj (slot-value 'state-objs))
            (send obj :redraw self)))
```

Whenever a window is covered and uncovered, the window is sent a `:redraw` message, which is responsible for ensuring the the window is redrawn correctly.

If no such method is defined, then the window contents will not be current. The `:redraw` method defined above send each control button object and each state object the `:redraw` method along with a pointer to the window itself. Those methods, which are described later, redraw themselves appropriately in the window.

3.1.17 The `:add-some-menu-items` method

```
(defmeth dmc-proto :add-some-menu-items ()
  (let (
        (run-item
         (send menu-item-proto :new "Run Quietly"
               :action
               #'(lambda ()
                   (let ((tmp (get-value-dialog
                               "How many steps?" :initial 1)))
                     (when tmp
                           (send self :run-dmc-silently
                                 (select tmp 0)))))))

        (sample-paths-item
         (send menu-item-proto :new "Sample Paths"
               :action
               #'(lambda ()
                   (send self :sample-paths
                         (not
                          (slot-value 'sample-paths))))))

        (desc-item
         (send menu-item-proto :new "Describe"
               :action
               #'(lambda () (send self :describe))))

        (s-color
         (send menu-item-proto :new "Normal State Color"
               :action
               #'(lambda () (send self :set-state-color))))

        (c-color
         (send menu-item-proto :new "Current State Color"
               :action
               #'(lambda ()
                   (send self :set-c-state-color))))
```

```
    (s-text-color
     (send menu-item-proto :new "State Text Color"
           :action
           #'(lambda ()
               (send self :set-state-text-color))))

    (c-text-color
     (send menu-item-proto
           :new "Current State Text Color"
           :action
           #'(lambda ()
               (send self :set-c-state-text-color))))

    (hist-item
     (send menu-item-proto :new "History"
           :action
           #'(lambda ()
               (send self :record-history
                     (not (send self :history)))))))

  (send self :menu (send menu-proto :new "Menu"))
  (send (send self :menu) :append-items run-item
        sample-paths-item hist-item desc-item
        (send dash-item-proto :new)
        s-color c-color s-text-color c-text-color)))
```

This method adds some items to the window menu. Those items include the following:

Run Quietly This item allows the user to invoke the `:run-dmc-silently` method silently using the mouse. The user is queried for the number of steps in a pop-up dialog box. A default of 1 step is supplied.

Sample Paths This item enables and disables sample paths window. If enabled, a line plot of the sample paths is displayed against time.

Describe This item enables the user to get a summary of the number of visits to each state in xlispstat command window.

Normal State Color This item enables the user to change the normal state color to suit his/her preference.

Current State Color This item enables the user to change the current state color to suit his/her preference.

State Text Color This item enables the user to change the color of the text in the normal state buttons.

Current State Text Color This item enables the user to change the text in current state button.

The color changing items are separated from the other items by a dashed line and all these items are installed in the menu.

3.1.18 The `:history` method

```
(defmeth dmc-proto :history (&optional val)
  (if val
      (setf (slot-value 'history) val)
    (slot-value 'history)))
```

This method sets and retrieves the value in the history slot. The value is set if `val`, which should be boolean, is supplied.

3.1.19 The `:record-history` method

```
(defmeth dmc-proto :record-history (on)
  (let ((hist-item
         (select (send (send self :menu) :items) 2)))
    (case on
          (nil (setf (slot-value 'history) nil)
               (format *history-file-handle*
                       "End Time: ~d~%" (slot-value 'time))
               (close *history-file-handle*)
               (send hist-item :mark nil))
          (t (let ((fname (get-string-dialog
                           "Name of output file: "
                           :initial *history-file*)))
               (when fname
                     (setf (slot-value 'history) t)
                     (setf *history-file-handle*
                           (open fname :direction :output))
                     (format *history-file-handle*
                             "Start Time: ~d~%"
                             (slot-value 'time))
                     (send hist-item :mark t)))))))
```

This method disables or enables the recording of history. When it is enabled, the user is prompted for a file name into which the state visits can be recorded. A default name of "history.mc" is provided. When the file is opened for output, the current time is recorded and the history item in the menu is marked with a check mark. When the history recording is disabled, the check mark is removed and the output file is closed. The ending time is also recorded.

3.1.20 The `:sample-paths` method

```
(defmeth dmc-proto :sample-paths (on)
  (let ((menu-item (select (send (send self :menu) :items) 1))
        (boss self))
    (case on
          (nil (send (slot-value 'sample-paths) :close))
          (t (let ((w (plot-lines
                       (list (slot-value 'time))
                       (list (slot-value 'current-state))))
                   (y-lim
                    (select (get-nice-range
                             0 (send self :no-of-states)
                             5) 1)))
               (send w :range 1 0 y-lim)
               (send w :title
                     (concatenate 'string
                                  (send boss :title)
                                  ": Sample Paths"))
               (if (eql *time-window* 0)
                   (send w :has-h-scroll 1000)
                 (send w :range 0 0 *time-window*))
               (send menu-item :mark t)
               (setf (slot-value 'sample-paths) w)
               (defmeth w :close ()
                 (send menu-item :mark nil)
                 (send boss :slot-value 'sample-paths nil)
                 (send self :remove)))))))
```

This method turns the display of the sample paths on or off. When the display is enabled, the sample paths window is given a title which if formed by concatenating the title of the `dmc-proto` instance with the string ": Sample Paths". The x and y range are adjusted so that the sample paths window need not be resized all the time. The menu item is marked with a check and the identity of the sample paths window is recorded in the slot `sample-paths`. The sample paths window's `:close` method is redefined so that when it is closed with a mouse, the check against the menu item is removed and the slot value `sample-paths` is also set back to `nil`. This ensures a sane exit. When sample paths is disabled, then all we need to do is to send the sample paths window a `:close` message.

3.1.21 The `:set-color` methods

We shall not discuss the color methods because they are not fully implemented. It is hoped at some future date that users will be able to set the color interac-

tively using a package like the first author's `color.lsp`.

3.2 Description of `states.lsp`

The file `states.lsp` contains the code for the state button objects. These buttons are slightly different from the control buttons because they are oval in shape, and moreover, they perform different actions. It is possible to have one object prototype for both the control and the state buttons and distinguish between them using a slot that indicates the nature of the button, but we decided to do it separately. When the user clicks the mouse on any state button, the user gets a summary of the number of visits to that state.

```
(defproto state-proto '(title state-no loc-and-size no-of-visits
                              pvec))
```

The `state-proto` prototype is defined with slot names `title`, to hold the name of the state, `state-no`, to hold the number corresponding to the state, `loc-and-size`, to hold the location and size of the state button in the `dmc-proto` window, `no-of-visits`, to hold the number of times the Markov chain has been to the state, and `pvec`, to hold the list of transition probabilities in each state. The location and size are stored as a list of four values x, y, width and height. Recall the x value increases horizontally to the right while the y value increases vertically down in a graph window.

3.2.1 The `:isnew` method

```
(defmeth state-proto :isnew (name sno vec)
  (let ((n (length vec)))
    (dolist (j (iseq 1 (- n 1)))
            (setf (elt vec j) (+ (elt vec (- j 1)) (elt vec j))))
    (setf (slot-value 'pvec) vec)
    (setf (elt (slot-value 'pvec) (- n 1)) 1))
  (setf (slot-value 'no-of-visits) 0)
  (setf (slot-value 'title) name)
  (setf (slot-value 'state-no) sno))
```

A new instance of `state-proto` is returned by this method. The method is called with arguments `name`, a title, `sno`, the state number and `vec`, the transition probability vector. To make it easier for us to generate the states to go to from this state, we store the distribution function. Since we are dealing with floating point numbers, a precautionary measure is taken by setting the last entry to 1.

3.2.2 The `:loc-and-size` method

```
(defmeth state-proto :loc-and-size (&optional loc-size)
  (if loc-size
      (setf (slot-value 'loc-and-size) loc-size)
    (slot-value 'loc-and-size)))
```

This method can be used to set or retrieve the location and size of the state in the window where it resides.

3.2.3 The `prob-vec` method

```
(defmeth state-proto :prob-vec ()
  (let ((n (length (slot-value 'pvec)))
        (tmp (copy-vector (slot-value 'pvec))))
    (dolist (j (iseq (- n 1) 1))
            (setf (elt tmp j) (- (elt tmp j) (elt tmp (- j 1)))))
    tmp))
```

This method recreates the transition probability vector from the distribution function. The method is usually used only for printing the transition matrix when the user so desires.

3.2.4 The `:no-of-visits` method

```
(defmeth state-proto :no-of-visits ()
  (slot-value 'no-of-visits))
```

This is an accessor method for the `no-of-visits` slot.

3.2.5 The `:you-are-current-state` method

```
(defmeth state-proto :you-are-current-state (w)
  (let ((nv (slot-value 'no-of-visits)))
    (setf (slot-value 'no-of-visits) (+ nv 1)))
  (send self :redraw w))
```

When the Markov chain goes to a new state, that state is sent this message. Therefore, this method increments the number of visits and asks itself to be redrawn. The net effect of this is to highlight this state as the current state in the window.

3.2.6 The `you-are-not-current-state` method

```
(defmeth state-proto :you-are-not-current-state (w)
  (send self :redraw w))
```

When the markov chain makes a transition from state l to state k, the state l is sent this message. Thus, this message restores the state to a normal state color. This is accomplished by just sending itself a `:redraw` message, since the `:redraw` method checks whether a state is the current state or not before drawing.

3.2.7 The `:redraw` method

```
(defmeth state-proto :redraw (window)
  (let* ((dc (send window :draw-color))
         (cs (eql (send window :current-state)
                  (slot-value 'state-no)))
         (tc (if cs
                 *current-state-text-color*
               *normal-state-text-color*))
         (bc (if cs
                 *current-state-color*
               *normal-state-color*))
defmeth          (ls (send self :loc-and-size))
         (x (first ls))
         (y (second ls))
         (w (third ls))
         (h (fourth ls))
         (cx (+ x (round (* .5 w))))
         (cy (+ y (round (* .5 h)))))
    (send window :draw-color bc)
    (send window :paint-oval x y w h)
    (send window :draw-color tc)
    (send window :draw-text
          (format nil "~d" (slot-value 'state-no))
          cx (+ cy (round (* .5 (send window :text-ascent))))
          1 0)
    (send window :draw-color dc)
    (send window :frame-oval x y w h)
    nil))
```

This method is in charge of redrawing the state button in the window appropriately. We have to determine whether the state is the current state or not. If it is the current state, then the state has to be redrawn with its proper colors. Otherwise, it has to be redrawn with the usual colors. The appropriate drawing color is chosen and an oval is painted at the location in the window where the state button resides. The state title, which is usually the state number, is also redrawn in the appropriate text color in the center of the state button. The default drawing color is restored afterwards.

3.2.8 The `:my-click` method

```
( state-proto :my-click (x y m1 m2)
  (let* ((ls (send self :loc-and-size))
         (left-x (first ls))
         (bot-y (second ls))
         (rt-x (+ left-x (third ls)))
         (top-y (+ bot-y (fourth ls))))
    (if (and (< left-x x rt-x) (< bot-y y top-y))
        t
      nil)))
```

The method `:my-click` checks if the mouse was clicked returns `t` if the mouse is clicked in a state's button. The coordinates of where the mouse click occurred are passed as `x` and `y`. The method merely checks whether the click was in a rectangle encompassing the state button by using information about its location and size in the window. If so, it returns `t` else it returns `nil`.

3.2.9 The `:do-click` method

```
(defmeth state-proto :do-click (w)
  (let ((nv (send self :no-of-visits))
        (time (send w :time)))
    (if (eql time 0)
        (message-dialog (format nil
                                "State ~s~%No. of visits ~d~%"
                                (send self :title) nv))
      (message-dialog (format nil
                              "State ~s~%No. of visits ~d~%~
                                Proportion ~7,5f~%"
                                (send self :title)
                                nv (/ nv time))))))
```

If the mouse is clicked on a state button, a message dialog pops up informing the user of the number of visits to the state. The user has to dismiss the message dialog before he or she can continue further.

3.2.10 The `:reset` method

```
(defmeth state-proto :reset ()
  (setf (slot-value 'no-of-visits) 0))
```

This method resets the state to its original state at the start of the simulation, which only involves setting the slot value `no-of-visits` to zero.

3.2.11 The `:title` method

```
(defmeth state-proto :title (&optional title)
  (if title
      (setf (slot-value 'title) title)
  (slot-value 'title)))
```

This method sets or retrieves the title of the state.

3.2.12 The `:describe` method

```
(defmeth state-proto :describe (&optional time)
   (let ((nv (send self :no-of-visits)))
    (if (> nv 0)
        (if time
            (format t "~3d         ~7d       ~1,5f~%"
                    (slot-value 'title) nv (/ nv time))
          (format t "State ~a, No of visits: ~g~%"
                  (slot-value 'state-no) nv)))))
```

This method prints the number of visits to the state and the proportion of time the markov chain spent in the state if time is greater than 0. Note that if the state has not been visited yet, this method prints nothing.

3.2.13 The `:next-state` method

```
(defmeth state-proto :next-state ()
  (let* ((u (select (uniform-rand 1) 0))
         (pvec (slot-value 'pvec))
         (state (catch 'index
                  (dotimes (j n)
                           (if (< u (select pvec j))
                               (throw 'index j))))))
    state))
```

This method will return the next state that the Markov chain will go to from this state. A uniform random variable, `u`, is generated. Then each element of `pvec` is checked to see if it is less than `u`. The first item that is corresponds to the next state that the Markov chain will go to. This algorithm is quite naive, but to use more efficient algorithms, one just needs to modify this method suitably.

3.3 Description of `buttons.lsp`

The file `buttons.lsp` contains the code for the design of the control buttons. These buttons are rectangular and sit in the topmost row of the window. The

user may click on any of these buttons to invoke actions associated with the buttons. It is also possible for a button to have no action to perform, as is the case with the `Time` button, which merely indicates the time.

The `button-obj-proto` is a prototype is defined as follows.

```
(defproto button-obj-proto '(boss title loc-and-size
                                  click-action))
```

The slot `boss` is the identity of the window in which the button will sit. The `title` slot holds the title for the button. The `loc-and-size` slot holds a list of four values: the x, y coordinates of the top left corner of the button and the width and height of the button in the *boss* window. The `click-action` slot holds an action to be performed when the mouse is clicked on the button. This value of this slot is actually `funcall`ed to perform the action.

3.3.1 The `:isnew` method

```
(defmeth button-obj-proto :isnew (title loc-and-size action)
  (setf (slot-value 'title) title)
  (setf (slot-value 'loc-and-size) loc-and-size)
  (setf (slot-value 'click-action) action))
```

The `:isnew` method for `button-obj-proto` requires three arguments, a title string, a list of four values containing the x and y coordinates of the top left corner of the button, the width of the button and the height of the button in the window where it is to be installed, and a function that will perform some action when the mouse is clicked on the button. These arguments are stored in the appropriate slots for future use.

3.3.2 The `:title` method

```
(defmeth button-obj-proto :title (&optional title)
  (if title
      (setf (slot-value 'title) title)
    (slot-value 'title)))
```

This method sets and retrieves the title for the button. The title is set if it is supplied.

3.3.3 The `:do-click` method

```
(defmeth button-obj-proto :do-click ()
  (funcall (slot-value 'click-action)))
```

This method performs the action associated with the button. It is called when the user clicks the mouse on the button. Since the action to be performed is stored in the `click-action` slot, this method just `funcall`s the value in that slot.

3.3.4 The `:loc-and-size` method

```
(defmeth button-obj-proto :loc-and-size (&optional loc-size)
  (if loc-size
      (setf (slot-value 'loc-and-size) loc-size)
    (slot-value 'loc-and-size)))
```

This method sets and retrieves the value of the slot `loc-size`. Recall that `loc-size` should be a list of four values.

3.3.5 The `:redraw` method

```
(defmeth button-obj-proto :redraw (window)
  (let* ((dc (send window :draw-color))
         (ls (send self :loc-and-size))
         (x (first ls))
         (y (second ls))
         (w (third ls))
         (h (fourth ls))
         (cx (+ x (round (* .5 w))))
         (cy (+ y (round (* .5 h)))))

    (send window :draw-color *button-back-color*)
    (send window :paint-rect x y w h)
    (send window :draw-color *button-text-color*)
    (send window :draw-text
          (slot-value 'title)
          cx (+ cy (round (* .5
                             (send window :text-ascent)))) 1 0)
    (send window :draw-color dc)
    (send window :frame-rect x y w h)
    nil))
```

This method is in charge of redrawing the button in the window where it is installed as appropriate. The colors to be used are available in the global variables `*button-back-color` and `*button-text-color` denoting the background color and the text color respectively. The window is sent the message to draw a framed rectangle whose background color is `*button-back-color*` and then the text is drawn centered horizontally in the button using `*button-text-color*`. The default drawing color is restored at the end.

3.3.6 The `:my-click` method

```
(defmeth button-obj-proto :my-click (x y m1 m2)
  (let* ((ls (send self :loc-and-size))
```

```
        (left-x (first ls))
        (bot-y (second ls))
        (rt-x (+ left-x (third ls)))
        (top-y (+ bot-y (fourth ls))))
    (if (and (< left-x x rt-x) (< bot-y y top-y))
        t
      nil)))
```

This method is used to detect whether the user clicked the mouse in the button. All it does is check whether the coordinates where the button was clicked in the window are inside the rectangle encompassing the button. If so, it returns `t`, otherwise, it returns `nil`.

3.4 Description of `stochast.lsp`

The file `stochast.lsp` consists of a few functions that work on stochastic matrices. Some of the functions are just conveniences and others determine whether a matrix is stochastic or not.

3.4.1 The `:random-stochastic-matrix` function

```
(defun random-stochastic-matrix (n)
  (let* ((tmp (uniform-rand (* n n)))
         (m (row-list (matrix (list n n) tmp))))
    (dotimes (j n)
             (setf (select m j)
                   (/ (select m j)
                      (sum (select m j))))
             (setf (select (select m j) (- n 1))
                   (- 1 (sum
                         (select (select m j)
                                 (iseq 0 (- n 2)))))))
    (matrix (list n n) (combine m))))
```

This function will return a random stochastic matrix of dimension n. First, n rows of n uniform random variables are generated. Then each row is normalized so that the entries are between 0 and 1 and add up to 1. We used this function mainly to debug some routines and therefore, it probably serves not much of a purpose.

3.4.2 The `square-matrixp` function

```
(defun square-matrixp (m)
  (unless (matrixp m)
          (error "not a matrix - ~a" m))
```

```
  (let ((dim (array-dimensions m)))
    (eql (select dim 0) (select dim 1))))
```

This functions returns `t` if the matrix `m` is a square matrix; otherwise, it returns `nil`. It just checks if the dimensions of the matrix are equal.

3.4.3 The `stochastic-matrixp` function

```
(defun stochastic-matrixp (m)
  (unless (square-matrixp m)
          (return))
  (dolist (row (row-list m) t)
          (unless (< (abs (- (sum row) 1))
                     machine-epsilon)
                  (return))))
```

This function returns `t` if `m` is a stochastic matrix; otherwise, it returns `nil`. Note the use of the global variable `machine-epsilon` to check whether each row sums to 1. This is necessary since we are dealing with machine representation of probabilities.

3.4.4 The `doubly-stochastic-matrixp` function

```
(defun doubly-stochastic-matrixp (m)
  (and (stochastic-matrixp m)
       (stochastic-matrixp (transpose m))))
```

This method just returns `t` if the matrix is doubly stochastic, `nil` otherwise.

3.5 Description of `dmc-opts.lsp`

The file `dmc-opts.lsp` contains code that selects options for use by `dmc.lsp`.

```
(case (screen-has-color)
      (nil
       (defvar *normal-state-color* 'white)
       (defvar *current-state-color* 'black)
       (defvar *normal-state-text-color* 'black)
       (defvar *current-state-text-color* 'white))
      (t
       (defvar *normal-state-color* 'blue)
       (defvar *current-state-color* 'red)
       (defvar *normal-state-text-color* 'white)
       (defvar *current-state-text-color* 'black)))
(defvar *history-file* "history.mc")
(defvar *time-window* 50)
(defvar *run-step-delay* 30)
```

Depending on whether the user has a color monitor or not, some global variables are given appropriate values. The variable `*normal-state-color*` refers to the background color of a normal state, `*current-state-color*` refers to the background color of the current state, `normal-state-text-color*` refers to the color used for text in a normal state, and `*current-state-text-color*` refers to the color used for text in the current state. The user can change these to reflect his or her preference.

For history recording, a default file name of "history.mc" is used. Note that this does not imply history will be recorded; the user has to enable history recording from the menu, and when that is done, the user is given a choice to change the file name. The variable `*time-window*` refers to the window width used for sample path display—the sample path for only the last `*time-window*` time ticks is displayed in the sample paths window. If the user sets this variable to zero, then the user gets a scroll-bar under the window which can be scrolled for the complete sample path plot. It must be noted that even when the value of `*time-window*` is non-zero, the complete history can be displayed in the plot by rescaling the plot.

Finally, since the transition takes place extremely quickly, at least on a workstation, a delay is employed to slow things down. A value of 30 means a delay of half a second.

3.6 Description of `but-opts.lsp`

The file `but-opts.lsp` contains the options for the control buttons.

```
(provide "but-opts")
(case (screen-has-color)
      (nil
       (defvar *button-text-color* 'white)
       (defvar *button-back-color* 'black))
      (t
       (defvar *button-text-color* 'black)
       (defvar *button-back-color* 'cyan)))
```

The variables `*button-text-color*` and `*button-back-color*`, refer to the text color and the background color of the control buttons respectively.

4 Some Applications

We present two applications in this section. Although, we have experimented with quite a few applications described in Kemeny and Snell[4], it would be cumbersome to reproduce them all here.

The first application is a realization of the gambler's ruin problem. The user can visually play along and estimate the probability of the gambler going broke. The second is an illustration of the Ergodic theorem for Markov Chains.

4.1 Gambler's Ruin

This scenario deals with a gambler who has a probability p of winning one more dollar with each play of the game, and a probability q of losing a dollar. If successive plays of the game are independent, then, the collection of random variables $\{X_n,\ n > 0\}$, where X_n is the gambler's fortune at time n, is a Markov Chain. Given a starting amount of i dollars, we can calculate the probability that the gambler will win N dollars before going broke. The following code will give a visual representation of the gambler's fortune. Starting with i dollars, the probability P_i that the gambler will eventually have N dollars is given by:

$$P_i = \begin{cases} \frac{1-(q/p)^i}{1-(q/p)^N}, & \text{if } p \neq \frac{1}{2} \\ \frac{i}{N}, & \text{if } p = \frac{1}{2} \end{cases}$$

For a detailed explanation of the derivation of this formula, see Ross[5].

This can be illustrated by means of our code. The file `gambler.lsp` contains the following.

```
(defvar n 26)
(defvar *prob-win* .5)
```

These two lines define the gambler's maximum fortune, $26 - 1 = \$25$, and the probability of winning each game, p, is 0.45.

Next, since we are going to be playing a lot of games, we would like to speed up things as much as possible. One easy candidate for the speedup is the `:next-state` method for the `state-proto` object. We do that as follows.

```
(defmeth state-proto :next-state ()
  (let ((cs (slot-value 'state-no)))
    (if (or (eql cs 0) (eql cs (- n 1)))
cs
      (let ((u (select (uniform-rand 1) 0)))
(if (< u *prob-win*)
    (+ cs 1)
  (- cs 1))))))
```

This method takes into account the fact that the transition matrix is sparse. Note that the value of `n` is used in the code, which means that `n` should not be changed in the course of the run.

Now, we are ready to create the Markov Chain.

```
(defvar m (identity-matrix n))
(dotimes (i (- n 1))
         (when (> i 0)
               (setf (aref m i i) 0)
               (setf (aref m i (+ i 1)) *prob-win*)
               (setf (aref m i (- i 1)) (- 1 *prob-win*))))
(def initial-state 12)
(setf z (send dmc-proto :new m initial-state))
```

The transition matrix for our Markov Chain is:

$$P_{00} = P_{NN} = 1$$
$$P_{i,i+1} = p = 1 - P_{i,i-1} = 1 - q, \quad i = 1, 2, \ldots, N-1.$$

In other words, the probability of the Markov chain moving to the next state, or, of the gambler winning this play of the game, is p, as was mentioned earlier. Similarly, the probability of the Markov chain moving back one state, or, of the gambler losing a dollar, is $q = 1 - p$. By defining the initial state of the Markov chain to be 12, we are giving the gambler twelve dollars to start with.

The last line above creates a new instance of the `dmc-proto` object using the transition matrix that was just defined, beginning in our chosen initial state. Note that although the transition matrix is used in creating the Markov Chain, it is not used afterwards, since we have overridden the `:next-state` method for `state-proto`.

Upon executing the code, the user gets a window like the one shown in figure 1. The current State is highlighted with a different color. In the figure, we have also taken the liberty to show the sample paths window and the pop-up message dialog box when the mouse is clicked on one of the state buttons. Note that the sample path window is not enabled automatically, but has to be requested from the menu in the Markov Chain window.

Now, how do we run the Markov Chain a large number of times? The following code will help us do that.

```
(defun test-gambler (number)
  (let ((w 0)
(l 0)
(tmp nil))
    (dotimes (j number)
             (format t "Game: ~d~%" j)
     (send z :reset)
     (send z :run-dmc-silently 500)
     (setf tmp (send z :current-state))
     (if (eql tmp (- n 1))
 (setf w (+ w 1))
       (if (eql tmp 0)
```

```
(setf l (+ l 1)))))
 (format t "Wins: ~d, Losses: ~d, ~
            Decided games: ~d, ~
            Proportion wins: ~d~%"
 w l (+ w l) (/ w (+ w l)))))
```

This function will play `number` games, where `number` is given as an argument to this function. Note that after each game, we determine the current state, so that we may detect whether the gambler won or lost, or whether the game did not end. When the games are all over, the number of wins and losses is printed along with the proportion of wins.

Figure 1: The Markov Chain window created by `gambler.lsp`

In a test run, we called the function `test-gambler` with an argument 1000. After 1000 games, the gambler's fortune reached \$25 474 times in 976 decided games, yielding a winning percentage of 48.57%. According to the formula, since the probability of winning each turn was 0.5, the gambler's winning percentage was expected to be 12/25, or 48%. The computer-generated Markov chain has yielded results very close to the ones predicted theoretically. It must be remarked that this simulation took quite a while even on a loaded workstation.

We can think of many things that can be optimized, but that is a subject that doesn't concern us now.

Now suppose, the probability of winning is not .5, but say, 0.45, and the gambler's initial fortune is \$15. All one needs to do is to change the values of `*prob-win*` and repeat the whole process using 15 as the initial state. A run of 1000 games yielded 126 wins in 997 decided games, a winning percentage of 0.126379. This is close to the the theoretical value of

$$\frac{1-(0.55/0.45)^{15}}{1-(0.55/0.45)^{25}} = 0.128657.$$

4.2 The Ergodic Theorem

Consider the following transition matrix:

$$\mathbf{P} = \begin{bmatrix} .5 & .3 & .2 \\ .1 & .8 & .1 \\ .7 & .2 & .1 \end{bmatrix}$$

If this matrix is raised to successive powers, an interesting thing happens:

$$\mathbf{P}^{(5)} = \begin{bmatrix} .313 & .554 & .133 \\ .283 & .590 & .127 \\ .318 & .549 & .134 \end{bmatrix}$$

$$\mathbf{P}^{(10)} = \begin{bmatrix} .297 & .573 & .130 \\ .296 & .575 & .130 \\ .297 & .573 & .130 \end{bmatrix}$$

It seems that the entries in each column are approaching the same value.

To better explain what this result means, a few more terms must be explained. A state has *period* d if $P_{ii}^n = 0$ if n is not divisible by d. For example, if the process starts in state i and it is possible for the process to enter that state only at times 3, 6, 9,..., then state i has period 3. Any state with period 1 is *aperiodic*. A state i is *positive recurrent* if, starting in i, the process is expected to return to i in a finite amount of time. A positive recurrent, aperiodic state is called an *ergodic* state. A Markov chain is said to be *irreducible* if all its states communicate, or, given any state i, it is possible to reach state j within a finite time, and vice versa.

We can now state what is known as The Ergodic Theorem:

For an irreducible ergodic Markov chain $\lim_{n\to\infty} P_{ij}^n$ exists and is independent of i, and is denoted by

$$\pi_j = \lim_{n\to\infty} P_{ij}^n$$

Furthermore, π_j is the unique nonnegative solution to

$$\pi_j = \sum_{i=0}^{\infty} \pi_i P_{ij}, j \geq 0, \text{ with } \sum_{j=0}^{\infty} \pi_j = 1$$

The value π_j is called the *limiting probability* that the process will be in state j at time n. It can also be shown that this value is also the proportion of time that the process will be in state j in the long run.

Because `dmc.lsp` is designed to calculate the proportion of time that the Markov chain is in each state for the time that the process has run, we can graphically illustrate the validity of this theorem.

The file `thm.lsp` contains the following code:

```
(def n 15)
(def m (random-stochastic-matrix n))
(def initial-state 7)
(setf z (send dmc-proto :new m initial-state))
```

In a sample run, the following transition matrix was created for a 5-state Markov chain by the `random-stochastic-matrix` function:

$$\begin{bmatrix} 0.209948 & 0.197941 & 0.0526538 & 0.465235 & 0.0742217 \\ 0.217497 & 0.234188 & 0.311191 & 0.222549 & 0.0145759 \\ 0.029961 & 0.0702461 & 0.0834497 & 0.440806 & 0.375538 \\ 0.206709 & 0.172418 & 0.258068 & 0.169594 & 0.193211 \\ 0.0142226 & 0.178653 & 0.117397 & 0.366932 & 0.322796 \end{bmatrix}$$

Using `dmc.lsp`, the Markov chain was run for one million steps. The following results were obtained:

```
> (send z :run-dmc-silently 1000000)
NIL
> (send z :describe)
-----------------------------------------

State    No of visits  Proportion
-----------------------------------------
0           137858       0.13786
1           169561       0.16956
2           178713       0.17871
3           308272       0.30827
4           205597       0.20560
-----------------------------------------
```

A calculation of the limiting probabilities for the transition matrix above yielded:

$$\pi_0 = 0.13780$$
$$\pi_1 = 0.16943$$
$$\pi_2 = 0.17862$$
$$\pi_3 = 0.30836$$
$$\pi_4 = 0.20578$$

The proportion of time that the Markov chain spent in each state is remarkably close to the limiting probability for each state, as predicted by the theorem.

5 Final Remarks

We hope that users of our software will report improvements or suggestions to us. The principal contact for the software and the paper is the first author at `naras@cda.mrs.umn.edu`. We also hope to collect more examples of applications to use them in teaching introductory probability concepts.

References

[1] Feller, William, *An Introduction to Probability Theory and its Application*, Third edition, John Wiley, (1968).

[2] Gomatam, Shanti, *Answers to Frequently Asked Questions about S*, `statlib` archive, (1992).

[3] Kantrowitz, Mark, *Answers to Frequently Asked Questions about Lisp*, `rtfm.mit.edu` anonymous `ftp` archive, (1993).

[4] Kemeny, John G., and Snell, J. Laurie, *Finite Markov Chains*, Springer Verlag, (1976).

[5] Ross, Sheldon M., *Introduction to Probability Models*, Fourth edition, Academic Press, (1989).

[6] Sewell, Wayne, *Weaving a program: Literate programming in WEB*, Van Nostrand Reingold, (1989).

[7] Sitaram, Dorai, *How to use SLaTeX*, `cs.rice.edu` anonymous `ftp` archive, (1991).

[8] Tierney, Luke, *Lisp-Stat, An Object-Oriented Environment for Statistical Computing and Dynamic Graphics*, Fourth edition, Academic Press, (1990).