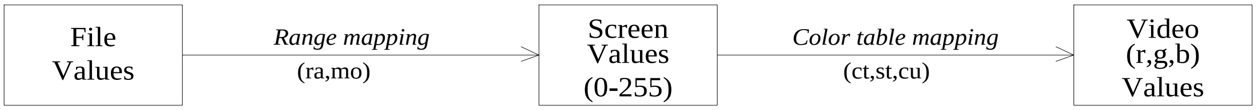
A Image value mapping in Skyview

The color and intensity which appear on the screen are the result of two mappings.



The first mapping, which we call *range mapping* converts the numeric values from the image file to screen values from 0 to 255. These screen values are put into the video display memory. This mapping is performed during the **paint** command. The second mapping is called *color table mapping.* The screen value is used as an index into a color table of red, green and blue values (RGB). The resulting RGB values are sent to the video display. This mapping is performed continuously in real-time by the video hardware. Appendix B describes the default color tables and how to set up a *color table file.*

Skyview provides commands for controlling both of these mappings. As shown in the diagram above, the range mapping is controlled by the **range** and **mode** commands. The color table mapping is controlled by the **color table**, **stretch**, and **cutoff** commands.

The range mapping is applied by *skyview* as the pixels are being read and converted to their screen values. This involves an arithmetic rescaling of each pixel, and is therefore slow. Changes to the range mapping parameters do not affect currently displayed images. The changes are applied only to images painted or repainted subsequently.

Changes to the color table mapping are really changes to the hardware color map itself. Since the color map is used continuously by the video hardware, any such changes result in an immediate change in all of the displayed images.

A.1 Modes

Skyview currently supports four transfer functions for range mapping: linear, logarithmic, loglog, and histogram equalization. In linear mode, the image pixel value (I) is mapped to a screen value (S) according to,

$$S = \begin{cases} 0, & \text{if} I < R_l, \\ 254\frac{I-R_l}{R_h-R_l}, & \text{if} R_l \le I \le R_h, \\ 254, & \text{if} I > R_h. \end{cases}$$

where R_l and R_h are the low and high values set for range mapping (by default these are set to the first 1st and 99th percentile values in the image).

In logarithmic mode the mapping is done according to:

$$S = \begin{cases} 0, & \text{if} I < R_l, \\ 254\ log_{10}\left(9\frac{I-R_l}{R_h-R_l}+1\right), & \text{if} R_l \le I \le R_h, \\ 254, & \text{if} I > R_h. \end{cases}$$

The loglog mode applies the logarithmic transformation twice.

In the histogram equalization mode the image values are mapped into the 0-254 space such that each of the values from 0 to 254 has an equal number of pixels displayed at that value. In this mode, the current range values controlled by the **range** command are ignored.

B Making your own Color Tables

There are 8 different pre-defined color lookup tables available, numbered 0 through 7. To change color tables, (in this case, to color table number 4) use a command like:

```
> ct 4
```

Here is a brief description of the color tables available:

0	grey scale: black and white
1	reverse grey scale: white and black
2	a trip around a color cube
3	eyeballed attempt at matching a spectrum
4	standard for false colors: black, blue, green, red, white
5	reverse color table 4
6	same as 4, but with blue compressed
7	for difference images; white at center (128) with red and blue at the extremities

You could also create your own color table. The name of the file would be used with the color table command, instead of the number for the pre-defined color table. Color table file names you create may be numbers 0 to 7, but they will override the pre-defined **skyview** color tables. Also, if you're going to be using your own color tables often, you should add the directory they are in to your SDATA[24] path.

To do this, simply create a new file within your own directory. Below is an example and description of color table number 4 as described above:

[24]See **Section 2.6.5** for more information on the SDATA environment variable.

0	0	0	0
17	0	0	95
34	0	20	175
51	0	63	250
68	0	112	255
85	0	162	175
102	0	210	90
119	20	255	30
136	135	240	0
153	210	205	0
170	240	135	0
187	255	20	0
204	240	0	0
221	255	85	85
238	255	170	170
255	255	255	255

Column 1 - identifies the screen value to which the RGB values will apply.
Column 2 - contains the red intensity for the screen value in Column 1.
Column 3 - contains the green intensity for the screen value in Column 1.
Column 4 - contains the blue intensity for the screen value in Column 1.

All screen values not specified will be interpolated.

C ASCII Table File Format

C.1 Introduction

This appendix describes an ASCII Table File Format [25] (ATFF) for use with tabular data. In this context we define tabular data as any text file that can be represented by clearly defined columns of numbers or text fields with headers and other subsidiary information. Examples of this include slices through images and database search output.

Our goal is to have a table file format that can be easily manipulated using commonly available tools; is easily exportable or importable; and can be readily accessed by application programs. While the FITS (Flexible Image Transport System) standard with table file and binary table file extensions is well suited to the last two requirements, the extensive header information and formatting needed in FITS to make it an acceptable data archive and interchange format also make it cumbersome for our needs to easily create and manipulate table files. ATFF files are formatted ASCII files; making them portable. Access to ATFF files by an application program is somewhat more involved because of the latitude in the format of such files. However, this is more than balanced by the ability to create, manipulate, or access them with nearly any text editor or spreadsheet program.

C.2 Ground Rules

The table file structure we have developed uses simple ASCII files that follow a few simple rules:

1. Tables *without* any header information are acceptable as long as there are no blank fields in the first line, the strings in each field contain no *white space* (i.e. ASCII TAB or SPACE characters), and each data entry in the first line is right justified in its field (the positions of the fields will be based on the positions of the characters in this first line, therefore their placement is critical).

2. Lines at the beginning of the file, that begin with a '\' (in the first column) are either comments or keyword definitions and are not analyzed as part of the table. Comments are introduced by following the backslash with a space character and cause the entire line to be ignored.

3. Keyword lines begin with '\REAL', '\CHAR', or '\INT'. They have the format 'keyword=value units' where keyword is the name of the variable, value is of the correct type for the variable, and units is an informational string. Blanks can be embedded in

[25] The table file format, syntax, and description given here are based on a design and Fortran implementation provided by Dr. John Good at IPAC.

any of the three fields if that field is enclosed in quotes. (Note: Keyword interpretation is not implemented in Skyview with the effect that keyword lines are treated as comments; they are ignored.)

4. The *first* non-comment / non-keyword line defines the field names or column labels (which can be blank) and the field positions. The '|' characters (one of which must be the first non-blank character to begin the first field) on this label line define the fields and any characters between them are taken to be the field name . A '|' indicates that any previous field has ended and a new field begins in the same physical column as the '|'.

5. Fields with blank names may be referred to by a name derived from their position number preceded by the string 'field' (e.g. 'field7' refers to the seventh field in the table). The same naming convention is used for files without headers. Fields that have been named can be reference either by name or position.

6. If the line following the label line also has a '|' as the first character of the first field, it is assumed to define the *type* of the fields. The first non-blank character *in* the field is used to determine the type: 'R', 'r', 'F', 'f', for floating numbers, 'I' or 'i' for integers, 'C' or 'c' for character strings. Since Skyview always interprets fields as character strings, this line and additional header lines are informational only and are not given any special interpretation.

7. If the lines immediately following the *type* line are header lines (i.e. beginning with '|' and preceeded by another header line), they contain, in order: text strings to be used in determining 'blank' fields and the units (e.g. Janskys).

8. Any additional header lines are ignored.

9. The first line that is not a keyword, comment, or header line begins the tabular data. All subsequent lines are treated as tabular data.

C.3 Examples

Tables 1-3 show examples of acceptable table files, starting with the simplest. Since these files are simple ASCII text files, the user is at liberty to use editors or other programs to manipulate them, so long as the structure defined above remains intact.

In the first example, everything defaults: the columns are delimited by where the numbers in the first record end.

In the second example (Table 2), we have added a character field (whose name is the string 'Source Name', spaces included and case sensitive) and integer ('Nsources'). The

Table 1: Simple table

5.245e-05	3.112e-2	345	34.556
4.256e-04	4.242e-3	33	234.453
3.343e-12	4.553e-7	4545	2.334
4.464e-17	4.333e-3	22	3.344
8.453e-04	3.563e-1	3934	343.344
5.445e-05	3.112e-2	845	14.556
4.243e-04	4.242e-3	733	224.459
3.943e-12	9.453e-7	4645	32.338
4.844e-17	8.334e-3	522	43.347
8.753e-04	7.563e-1	3434	353.346
5.667e-05	8.112e-2	445	64.555
4.576e-04	9.242e-3	333	274.454
3.444e-12	6.553e-7	4245	82.333
4.345e-17	5.333e-3	122	99.342
8.365e-04	4.563e-1	3334	303.341

Table 2: A More Complex Table

X	Y	Source Name	Nsources	ZZ	lon
real	real	char	int	real	real
5.245e-05	3.112e-2	AB #123a	12	345	34.556
4.253e-04	4.242e-3	DF #126a	13	33	234.453
3.343e-12	4.553e-7	DD #455q	222	4545	2.334
4.464e-17	4.333e-3	GJ #129w	88	22	3.344
8.453e-04	3.563e-1	DD #385o	121222	3934	343.344
5.445e-05	3.112e-2	TU #401u	834	845	14.556
4.243e-04	4.242e-3	DD #494w	33	733	224.459
3.943e-12	9.453e-8	YT #934m	986	5423	32.338
4.844e-17	8.334e-3	JJ #292j	2321	522	49.193
8.753e-04	7.563e-1	II #391d	1093	3434	782.310
5.667e-05	8.112e-2	QO #887x	1107	445	65.224

Table 3: An Even More Complex table

```
\ Because of the space following the '\', this line is not a keyword
\ definition line, but is instead considered to be a comment.
\ The next few lines define keywords of types 'real', 'char', and
\ 'int'.
\
\REAL time = 27.2345 hours
\CHAR user = 'John Good'
\INT 'run number ' = 57
\REAL Cost = 5674 dollars
\
\ Note that keywords can have units ('hours', for example), or names
\ with embedded spaces (if you enclose it in quotes: 'run number').
\
|    X        |     Y       | Source Name | Nsources |  ZZ  |  lon    |
|  real       |   real      |    char     |   int    | real |  real   |
|   --        |             |    none     |          |      |  000    |
|   Jy        |   flux      |             |  stars   |      | degrees |
  5.245e-05     3.112e-2       AB #123a        12      345    34.556
  4.253e-04     4.242e-3       DF #126a        13       33   234.453
  3.343e-12     4.553e-7         none         222     4545     2.334
    --                           none          88       22     000
    --                           none                 3934     000
  5.445e-05     3.112e-2       TU #401u                845     000
  4.243e-04     4.242e-3       DD #494w        33      733   224.459
    --                           none         986     5423    32.338
    --          8.334e-3       JJ #292j      2321      522     000
  8.753e-04     7.563e-1       II #391d      1093     3434     000
```

columns are delimited by the vertical bars. In this case, the header is needed since without it the Source Names values would be treated as two distinct fields.

In Table 3, we have added the rest of the header lines. They must appear in the order given: Field labels, data types, Blank symbol, Units. In order to include a Units header line, the labels, types, and blank header lines must appear before it with empty fields if necessary.

Note that by treating all fields as character strings the third value in the Y column is '4.553e-7', the fourth is '' (empty string), and the fourth value in the X column is '`--`'.

D The field_of_view Command

The **field_of_view** command, abbreviated **fov,** is used to create a diagram on the screen, using a set of graphics commands. This diagram can then be manipulated on the image.

The **fov** command requires that you specify the name of a description *file* on the command line. This file contains the graphics commands which define the actual diagram.

D.1 Interactive Placement of the Field of View

When only the **file** parameter is specified, the diagram is drawn on the image, and mouse is enabled for moving and rotating it.

Following each movement of the cursor, the diagram is re-drawn at the new cursor location. You can create a **skeleton** outline of the drawing to be used in the interactive placement. If you specify **skeleton** at the beginning of the list of commands in the **file** that define the skeleton, and then specify **end skeleton** at the end of the list, only the commands in between will be re-drawn at each new cursor location. This way, you can omit some of the detail of the diagram, decreasing the jerky motions caused by trying to continually re-draw a detailed graphic.

Clicking the left button places the cursor into position mode which drags the graphic around the image. Clicking the right button invokes rotation mode which changes the orientation of the graphic. Clicking the center button ends interactive mode, drawing the graphic in its full detail (everything that is not defined as the skeleton) and in the desired colors. This is called the *permanent graphic* below. In this example,

```
> fov iso
```

instructions for drawing the field of view for the iso telescope are taken from a file called **iso**. The mouse is then used to position and rotate the field of view as desired.

D.2 Command Line Placement of the Field of View

The **fov** command also accepts a position and orientation from the command line. Centered on the specified coordinate and rotated counter-clockwise the given number of degrees, **fov** draws a graphic whose details are described in the file specified. The command:

```
> fov iras 17h42m21s -28d55m41.1s 45
```

would place the diagram described in the file **iras** at 17h42m21s -28d55m41.1s, rotated 45 degrees.

D.3 The Description File

The description **file** is the most important part of the **fov** command. It contains the graphics commands for the diagram.

D.3.1 Syntax

The graphics commands in the input **file** that describe the diagram are placed one command per line. The commands available for use in the description file are:

#	Comment line
origin x y	Location of the "hot spot", the spot to which the coordinate in the **fov** command refers.
move x y	Set the starting position for the next graphic. This is like moving a pen to x y without touching the paper. This point becomes the **current position**.
draw x y	Draw a straight line from the current position to this new x y. This x y becomes the **current position**.
circle n x y	Draw a circle with a **radius** (in pixels) centered at x y. This x y becomes the **current position**.
scale s	Scale all following x y values by scale **s**.
offset x y	Offset all following x y values by **x** and **y**, that is move the origin to (**x y**).
skeleton	Start the portion of the graphic which is to be repeatedly drawn in interactive mode. If no skeleton is specified, the entire graphic is drawn repeatedly. Note: The skeleton is not drawn as part of the permanent graphic. Typically, some or all of the skeleton is duplicated in the permanent graphic description. If a skeleton is used, it must be placed at the start of the description file.
end skeleton	End the skeleton portion of the graphic. Description statements following this line constitute the permanent portion of the graphic. (See skeleton)
set color cname	Draw all future marks in color **cname** (any Xcolor–the color names known to the X-Windows system–can be used).

The origin of the graphics is placed at the cursor position if in the interactive mode. The origin is reset and graphics commands are reprocessed from the beginning every time the graphic is drawn.

D.3.2 Formulas

Positions on the graphic are computed as follows:

first apply offset, scale, and origin

$$
\begin{aligned}
x_a &= (x + x_{offset}) * s + x_{origin} \\
y_a &= (y + y_{offset}) * s + y_{origin}
\end{aligned}
$$

next apply rotation

$$
\begin{aligned}
x_b &= x_a * cos(\theta) + y_a * sin(\theta) \\
y_b &= -x_a * sin(\theta) + y_a * cos(\theta)
\end{aligned}
$$

next apply cdelt1

$$
\begin{aligned}
x_c &= x_{center} + x_b/d_1 \\
y_c &= y_{center} + y_b/d_2
\end{aligned}
$$

where
x = value from the graphic description line (like **move x y**).
y = value from the graphic description line (like **move x y**).
x_{offset} = value from offset statement
y_{offset} = value from offset statement
x_{origin} = value from origin statement
y_{origin} = value from origin statement
s = scale value
x_{center} = current cursor position or x value specified in fov command
y_{center} = current cursor position or y value specified in fov command
θ = current rotation (interactive) or value specified in fov command.
x_c = computed graphic position.
y_c = computed graphic position.
d_1 = CDELT1 value of the current image
d_2 = CDELT2 value of the current image

Notes:

All **x y**, **radius**, **x_offset**, and **y_offset** values are specified in degrees. To specify arcminutes or other unit, merely include the appropriate **scale** statement (*e.g.***scale .01666666667** for arcminutes).

E Skyview Treatment of BLANK Pixel Values

The **skyview** treatment of BLANK pixel values is designed to be compatible with both the FITS standard and the current IRAF conventions. The FITS standard states that BLANK pixels in floating point images shall be given the value NaN (from the IEEE floating point standard). IRAF, on the other hand, does not allow pixel values of NaN to appear in its image files. BLANK values in IRAF are represented by a user selected value which usually defaults to zero. In order to accomodate these two conventions, **skyview** uses the following rules.

Definition: The **current blank value** is the value derived using the following rules. It is derived when the image is painted, and cannot be changed without repainting the image. This **current blank value** is reported by the **ci** (current_image) command.

E.1 When Painting an Image

E.1.1 Integer images

With images where the FITS keyword BITPIX is equal to 8, 16, or 32, and all IRAF files:

The value given by the header keyword BLANK is observed.

But, if no BLANK header keyword exists, **skyview** uses the value assigned to **blank** by the last **set blank** command and issues a warning (because the value **set** may be leftover from earlier actions).

Also, if the blank value that has been **set** is out-of-range for the number of bits per pixel (defined by the BITPIX keyword), **skyview** discards the blank value and issue a warning.

If the user has not **set** a blank value (**set blank undefined**), all pixels in the image are considered non-blank.

E.1.2 Floating-point images

With images where the FITS keyword BITPIX is equal to -64, or -32:

Skyview uses IEEE NaN (Not a Number) as the blank pixel value.

However, if there is a BLANK keyword in the FITS header, the user is warned that this constitutes a FITS format error and the value is being ignored.

E.2 When 'Zapping' Pixels in an Image

When **zap**ping pixels, if **zap_value** has been **set** to **blank**, the **current blank value** (based on the above input image rules) is used. If the **current blank value** is undefined, the zap command will not work and an error message is given, so the user may define it.

If the command

```
> set zap_value n
```

is entered (and **n** is a numeric value), this **zap_value** is used.

In any case, if the **zap_value** is out-of-range for the number of bits per pixel (defined by the FITS keyword BITPIX), the zap command will not work and an error message is issued.

E.3 When Writing an Image

A new parameter has been added to the **write** command. It is called the **blank output value**. This parameter can be used to specify the value of blank pixels in the output image. It is **required** if the **current blank value** is IEEE NaN and you want your output file to be in IRAF format.

E.3.1 Integer Images

With images where the FITS keyword BITPIX is equal to 8, 16, 32, and all IRAF files:

If **blank output value** is specified, the FITS keyword BLANK is written in the header of the new image and the value is used. All blank pixels in the image are given this value.

But, if no **blank output value** is specified in the write command, the **current blank value** is used.

If the **current blank value** is undefined, no changes are made to either the BLANK header keyword or to any pixels in the new image.

E.3.2 Floating-point Images

With images where the FITS keyword BITPIX is equal to -64, -32:

If the **current blank value** is not IEEE NaN (i.e. image came from IRAF), blank pixels are changed to IEEE NaN and the BLANK keyword is not written in the header.

F Server Mode Details

F.1 Server Mode Library Man Pages

F.1.1 Function Summary

The calls in the svclib library are as follows:

```
int   handle    = svc_init ( char *filepath );
int   status    = svc_send ( int handle, char *cmdstring );
char *retstr    = svc_receive ( int handle );
int   status    = svc_close ( int handle );

SVC  *svcstruct = svc_struct ( char *retstr );
void  status    = svc_free ( SVC *svcstruct );
char *val       = svc_val ( char *retstr, char *key, char *val )
```

where SVC is defined by

```
typedef struct
{
  int  nalloc;
  int  count;   /* number of key-val pairs */
  char **key;
  char **val;
}  SVC;
```

F.1.2 svc_init

```
NAME
    svc_init - open a connection to a server program

SYNOPSIS
    #include "svclib.h"

    int  svc_init ( char *filepath );

DESCRIPTION
    svc_init() forks a separate process, starts up the server program
    specified by "filepath" in that subprocess, and sets up two-way
    communication with that program.

RETURN VALUES
    svc_init() returns an integer handle to be used in subsequent calls,
    or SVC_ERROR for failure.
```

F.1.3 svc_send

```
NAME
    svc_send  - send a command to a server program

SYNOPSIS
    #include "svclib.h"

    int  svc_send ( int handle, char *cmdstring );

DESCRIPTION
    svc_send() sends a command to the server program.  The server program
    will receive the command via its stdin stream as if it had been
    entered by an interactive user.

RETURN VALUES
    svc_send() returns SVC_OK or SVC_ERROR.
```

F.1.4 svc_receive

```
NAME
    svc_receive - fetch a line of response from a server program

SYNOPSIS
    #include "svclib.h"

    char svc_receive ( int handle );

DESCRIPTION
    svc_receive() fetches a line of output from a server program's
    stdout stream.

RETURN VALUES
    svc_receive() returns a pointer to the server's response string.
    This string is stored in a static area within svclib and will be
    overwritten by subsequent calls to svc_receive().  If svc_receive()
    fails, a NULL pointer is returned.
```

F.1.5 svc_close

```
NAME
    svc_close - shut down a server program and close the connection

SYNOPSIS
    #include "svclib.h"

    int svc_close ( int handle );

DESCRIPTION
    svc_close() flushes and then closes the pipes to the server program.
    It then kills the server program.

RETURN VALUES
    If "handle" is invalid svc_close() returns SVC_ERROR, otherwise it
    returns SVC_OK.
```

F.1.6 svc_struct

```
NAME
    svc_struct - parse a server program response string

SYNOPSIS
#include "svclib.h"

        SVC  *svcstruct = svc_struct ( char *retstr );

DESCRIPTION
    svc_struct() takes a response string from the server program and
    parses it into a structure defined as follows:

         typedef struct
         {
            int  nalloc;
            int  count;   /* number of key-val pairs */
            char **key;
            char **val;
         }  SVC;

    The response string is assumed to follow the "grammar" as defined
    in the grammar specification.

RETURN VALUES
    svc_struct() returns a pointer to the SVC struct.  Space for this
    struct is allocated by svc_struct(), and the space must subsequently
    be freed by the user program (see svc_free()).  If string "retstr"
    cannot be parsed, svc_struct() returns a NULL pointer.
```

F.1.7 svc_free

```
NAME
    svc_free - free the memory areas allocated by svc_struct

SYNOPSIS
#include "svclib.h"

        void svc_free ( SVC *svcstruct );

DESCRIPTION
    svc_free() frees the memory areas allocated by svc_struct().
```

F.1.8 svc_val

```
NAME
    svc_val - obtain keyword value from a server program response string

SYNOPSIS
#include "svclib.h"

        char svc_val (char *retstr, char *keystr, char *valptr)

DESCRIPTION
    svc_val() looks through the suplied server program response string
    for the specified keyword and returns the associated value.
    Internally, svc_val() calls svc_struct(), then searches through the
    key-val pairs for the desired key, and returns the associated val
    using the user specified valptr.  svc_val() then calls svc_free() to
    release the space allocated within svc_struct().

RETURN VALUES
    If successful, svc_val() places the resulting value into the location
    pointed to by valptr, and returns a pointer to that result.  If
    svc_val() fails, it returns a NULL pointer.
```

F.2 Server Mode Grammar

The complete grammar of the server mode responses is as follows:

```
   NOTES:  ::=  means "is defined as"
    |    means OR
   <x>   is a "category" name
  (x y)  means x and y taken together
    x*   means repeat zero or more times
    x+   means repeat one or more times
   "x"   means verbatim
   ..    means all the characters between (ASCII code order)


<isis>             ::= <data>
<data>             ::= <data_head> <content> <data_tail>
<data_head>        ::= "["
<data_tail>        ::= "]"
<content>          ::= <structure> | <array>

<structure>        ::= "struct " <struct_values>
<array>            ::= "array " <array_values>

<struct_values>    ::= <assignment> ("," <assignment>)*
<assignment>       ::= <w> <identifier> <w> "=" <w> <value> <w>
<value>            ::= <string> | <data>

<array_values>     ::= <element> ("," <element>)*
<element>          ::= <w> <value> <w>

<string>           ::= <unquoted_string> | <quoted_string>
<unquoted_string>  ::= <char_subset>+
<quoted_string>    ::= """ <char>* """

<identifier>       ::= <letter> (<letter> | <digit> | "_")*

<char>             ::= <char_subset> | "\"" | "\\"
                       | "," | " " | "=" | "[" | "]"

<char_subset>      ::= <letter> | <digit>
                       | "`" | "." | "/" | ";" | "'"
                       | "-" | "~" | "!" | "@" | "#" | "$" | "%"
```

```
                      | "^" | "&" | "*" | "(" | ")" | "_" | "+"
                      | "{" | "}" | "|" | ":" | "<" | ">" | "?"

<w>                 ::= (" ")*
<letter>            ::= "a" .. "z" | "A" .. "Z"
<digit>             ::= "0" .. "9"
```

For example,

```
[struct
  x = 5.e-6,
  x_coord="2h 23m 11.5s",
  list =
    [array 3,
      [struct
        x="another one",
        b=9df
      ],
      11,
      "test 1"
    ]
]
```

is a valid structure (line breaks here have been added for readability; they are not allowed in the structure itself).

Index