surf version 1.0.1

Stephan Endrass <*endrass@mathematik.uni-mainz.de*> May 28, 2000

The aim was to have a tool to visualize some real algebraic geometry: plane algebraic curves given as zero locus of a polynomial in two variables, algebraic surfaces given as zero locus of a polynomial in three variables, hyperplane sections of surfaces: algebraic space curves given as zero locus of two polynomials in three variables: a polynomial of arbitrary degree (the surface) and a linear polynomial (the hyperplane), and lines on surfaces given by two points on a surface. The algorithms should be stable enough not to be confused by curve/surface singularities in codimension greater than one and the degree of the surface or curve. This has been achieved quite a bit. We have drawn curves of degree up to 30 and surfaces of degree up to 20 successfully. However, there are examples of curves/surfaces of lower degree where surf fails to produce perfect images. This happens especially if the equation of the curve/surface is not reduced. Best results are achieved using reduced equations. On the other hand, surf displays the Fermat-curves accurately for degree up to 98.

Contents

1 Overview **3**

1.1 Acknowledgements . . . 3
1.2 Copyright . . . 3
1.3 How to get *surf* . . . 3
1.4 System requirements . . . 3
1.5 Starting *surf* . . . 4
1.6 Scripts versus graphical user interface . . . 4
1.7 Scripts . . . 4
1.8 Output . . . 4
1.9 Sample scripts . . . 5
1.10 *surf* and make . . . 5
1.11 Oddities, bugs and bug reports . . . 6

2 Introduction to *surf*'s command language **6**

2.1 Data types . . . 6
2.2 Operators . . . 6
2.3 Mathematical functions . . . 7
2.4 String functions . . . 7
2.5 Polynomial functions . . . 8
2.6 First examples . . . 8
2.7 Conditional statements . . . 9

3 Features **9**

3.1 Plane curves . . . 9

3.2 Surfaces . . . 10
3.3 Hyperplane sections . . . 11
3.4 Multiple curves/surfaces . . . 11
3.5 Graphs and isolines . . . 12
3.6 Interactive positioning . . . 13
3.7 Preview . . . 13
3.8 Anti aliasing surfaces . . . 13
3.9 Animations . . . 13
3.10 Stereo pictures . . . 14
3.11 Black & white images . . . 15
3.11.1 Dithering with blue noise . . . 15
3.11.2 Dithering with ordered dither . . . 15
3.11.3 Hybrid methods . . . 15
3.11.4 The black & white problem . . . 16
3.12 Algorithms . . . 16
3.13 Output . . . 16

4 List of all reserved words 17

4.1 Reserved words corresponding to the main window . . . 17
4.1.1 Examples . . . 18
4.2 Reserved words corresponding to the position window . . . 18
4.2.1 Examples . . . 19
4.3 Reserved words corresponding to the display window . . . 19
4.3.1 Examples . . . 20
4.4 Reserved words corresponding to the light window . . . 20
4.4.1 Examples . . . 22
4.5 Reserved words corresponding to the clip window . . . 22
4.5.1 Examples . . . 22
4.6 Reserved words corresponding to the dither window . . . 23
4.6.1 Examples . . . 23
4.7 Reserved words corresponding to the save color image window . . . 24
4.7.1 Examples . . . 24
4.8 Reserved words corresponding to the save dithered image window . . . 24
4.8.1 Examples . . . 24
4.9 Reserved words corresponding to the numeric window . . . 25
4.9.1 Examples . . . 25
4.10 Reserved words corresponding to the curve window . . . 25

4.10.1 Examples 25

1 Overview

1.1 Acknowledgements

I thank Prof. W. Barth (University Erlangen) for (en)forcing me to start this project. Hans Hülf, Rüdiger Örtel and Kai Schneider have spent lots of time on coding parts of *surf*. Some of the code has been copied from other places:

- Writing SUN rasterfiles and XWD files has been copied from Michael L. Mauldin's Fuzzy PixMap (fbm) library version 1.2.
- Writing a TIFF file has been copied from John Cristy's Image Magick version 3.0.
- The octree color reduction algorithm is copied from Ian Ashdown's article *Octree Color Quantization* in the C/C++ Users Journal Vol. 13, Number 3, pp. 31-43.

We thank all these people who made their code free so that we could use it.

1.2 Copyright

1.3 How to get *surf*

surf is available via http/ftp at the surf home page *http://surf.sourceforge.net*.

1.4 System requirements

To compile *surf*, the following software is needed:

- **GNU gcc/g++** version 2.7.x or higher (any ISO C++ compiler should work..),
- **make**,
- GNU **flex** version 2.5 or higher (minor versions should work also, but lex does *not* suffice),
- Berkeley **yacc** (GNU bison should work also),
- **GTK+** version 1.2.0 or later (only required if you want to compile **surf** with GUI support),
- **POSIX threads** (If you have Linux make sure you use glibc2),
- **GNU MP (gmp)** version 2 or later,

Warning: Be prepared, the memory consumption of *surf* is about

- 15 MB for a 1000x1000 pixel image,
- 60 MB for a 2000x2000 pixel image and
- 130 MB for a 3000x3000 pixel image (gobble!!).

1.5 Starting *surf*

surf is started by typing `surf` on the command line. Optional arguments are `-no-gui` (or `-n`) for starting *surf* without graphical user interface, `-exec` (or `-x`) to immediately execute the first passed script file and - when using surf *with* GUI - the usual GTK+ options. `-help` prints out the usage information:

```
surf -n|--no-gui file ...
surf [Gtk options] [[-x|--exec] file ...]
surf --help
```

1.6 Scripts versus graphical user interface

surf is designed to visualize algebraic curves and surfaces. This can be done either by *writing scripts* in surf's command language and executing them interactively or from another program (for example make), or by *using surf's graphical user interface*. By using scripts one can draw series of pictures where each picture consists of several surfaces/curves at a high resolution.

1.7 Scripts

Scripts in surf's command language are stored in files with the suffix `.pic`. These files consist of descriptions of curves and/or surfaces and some commands. They can be invoked in two ways:

- by loading a script and pressing the button *execute script* or
- by starting *surf* with the name of a script file from the command line. If you compiled/started *surf* without GUI support, it automatically starts interpreting the script. Otherwise you have to press the *execute script* button.

1.8 Output

surf calculates both color and black & white images. Color images can currently be stored in the following formats:

- **XWD**,
- **SUN rasterfile**,
- **PPM** and
- **JPEG**

Additionally one can choose a convenient colormap among

- Netscape 216 color cube (8 bit),
- optimized by an octree algorithm (8 bit) and

- true color (24 bit).

Black & white images can be stored in the following formats:

- **Postscript**,
- **Encapsulated Postscript**,
- **TIFF**,
- **XBM**,
- **PGM** and
- **PBM**.

1.9 Sample scripts

You will find some sample scripts together with *surf*'s distribution. They are stored in the `examples` directory.

1.10 *surf* and make

surf can be invoked from make. This comes in quite handy when visualising a series of curves/surfaces. Suppose there are script files s1.pic, s2.pic, ... , sn.pic which create during execution images s1.xwd, s2.xwd, ... , sn.xwd. If for example gif is the desired image file format, an appropriate makefile might look like:

```
#!/bin/bash
#
SURF    = surf
RM      = /bin/rm -f
CONVERT = convert
#
OBJS    = s1.gif s2.gif .... sn.gif
#
.SUFFIXES: .pic .gif
#
.pic.gif:
    ${SURF}    -n $<
    ${CONVERT} $*.xwd $*.gif
    ${RM}      $*.xwd
#
dummy:
    @echo ' '
    @echo 'usage:'
    @echo ' '
    @echo '    print this message:'
    @echo '        make'
    @echo ' '
    @echo '    build images:'
    @echo '        make all'
    @echo ' '
```

```
        @echo '    remove images:'
        @echo '        make clean'
        @echo ' '
#
all: ${OBJS}
#
clean:
        ${RM} *.gif
#
# end of makefile
```

Here convert is the Image Magick image format converter.

1.11 Oddities, bugs and bug reports

In case you find any bug, please use the excellent *Bug Tracking System* on *surf*'s project page at Sourceforge.

2 Introduction to *surf*'s command language

2.1 Data types

The language used in *surf*'s scripts is quite simple. It has got a (very restricted) C-like syntax and provides the four data types

- `int` (integer),
- `double` (double precision float value),
- `string` (any `""`-quoted string) and
- `poly` (any polynomial in x, y and z).

So a valid declaration/initialisation looks like:

- `int a=3;` or `int a; a=3;`
- `double b=3.3;` or `double b; b=3.3;`
- `string c="test.xwd";` or `string c; c="test.xwd";`
- `poly d=(x-3)^3-y^2+z;` or `poly d; d=(x-3)^3-y^2+z;`

There is no comma separator like in C. Declaring a name twice results in an error. The scope of the name begins at the point of its declaration and lasts until the end of the file. There is no method of undeclaring a name.

2.2 Operators

The following arithmetic operators are implemented:

operator	meaning	valid data types
+	binary plus	{int,double,poly}+{int,double,poly}
+	concatenation	{string}+{string}
+	unary plus	+{int,double,poly}
-	binary minus	{int,double,poly}-{int,double,poly}
-	unary minus	-{int,double,poly}
*	multiplication	{int,double,poly}*{int,double,poly}
/	division	{int,double,poly}/{int,double}
%	remainder	{int}%{int}
^	power	{int,double}^{int,double}
		{poly}^{int}
( )	brackets	({int,double,poly})
=	equals	{poly}={int,double,poly}
		{double}={int,double}
		{int}={int}
		{string}={string}
==	equal	{int,double}=={int,double}
!=	not equal	{int,double}!={int,double}
<	smaller than	{int,double}<{int,double}
<=	smaller or equal	{int,double}<={int,double}
>	greater than	{int,double}>{int,double}
>=	greater or equal	{int,double}>={int,double}

The precedence of operators copied from C.

2.3 Mathematical functions

There are some built-in math functions:

function	meaning	valid arguments	returns
sqrt	square root	sqrt({int,double})	double
pow	power	pow({int},{int,double})	double
		pow({double},{int,double})	double
sin	sinus	sin({int,double})	double
cos	cosinus	cos({int,double})	double
arcsin	arcus sinus	arcsin({int,double})	double
arccos	arcus cosinus	arccos({int,double})	double
tan	tangens	tan({int,double})	double
arctan	arcus tangens	arctan({int,double})	double

They take int and double as argument.

2.4 String functions

There are also two functions returning strings:

function	meaning	valid arguments	returns

```
itostr   | int to string | itostr({int})        | string
itostrn  | int to string | itostrn({int},{int}) | string of spec. length
```

itostr converts its argument to a string without blanks. For example `itostr( 31 )` returns "31". itostrn allows to specify the length of the string. For example:

- `itostrn( 3,88 )` returns "088"
- `itostrn( 4,88 )` returns "0088"

2.5 Polynomial functions

Some functions work on polynomials:

```
function | meaning       | valid arguments          | returns
----------------------------------------------------------------
deg      | degree        | deg({poly})             | int
len      | length        | len({poly})             | int
diff     | derivative    | diff({poly},{x,y,z})    | poly
rotate   | rotation      | rotate({poly},{double} |
         |               |   {xAxis,yAxis,zAxis}) | poly
hesse    | hesse surface | hesse({poly})           | poly
```

This enables you to work out arbitrary polynomials.

2.6 First examples

Values can be passed to *surf* by setting global variables. The most important two global variables are `curve` and `surface`, which should be set to the polynomial whose zero set should be visualized. So the shortest effective script contains only three lines, for example:

- 1st example: draw the newton knot

```
clear_screen;
curve=y^2-x^2*(x+1);
draw_curve;
```

- 2nd example: draw a sphere

```
clear_screen;
surface=x^2+y^2+z^2-80;
draw_surface;
```

Both examples can be invoked by pressing the button *execute script*. The command `draw_curve` is somehow equivalent to pressing the button *draw curve*. The command `draw_surface` is somehow equivalent to pressing the button *draw surface*.

2.7 Conditional statements

CAUTION: There are no `for` and no `while` statements. There is only the crude

```
if( INTEGER-EXPRESSION ) goto LABEL;
```

which you might remember from your early BASIC sessions. Here `INTEGER-EXPRESSION` can be arbitrary complicated as long as it results in an integer. `LABEL` is something like `NAME:` which has occurred before. Consider the example

```
int i=0;
loop:
    surface=x^2+y^2+z^2-(i+1.0)/2.0;
    clear_screen;
    draw_surface;
    filename="sphere"+itostrn( 2,i )+".ras";
    save_color_image;
    i=i+1;
if( i<50 ) goto loop;
```

which obviously draws fifty spheres of increasing radius and saves them into the **SUN rasterfiles**:

sphere00.ras ... sphere49.ras

There exist some more commands explained briefly afterwards. C++ comments are welcome. **Warning:** Check if your loop terminates!

3 Features

In this section most features of *surf* are explained. Many of these features can be invoked from the graphical user interface. All features can be invoked through *surf*'s command language. Command language features are only explained if not accessible through the GUI. For a complete reference to the command language, have a look at the next section.

3.1 Plane curves

To draw a plane curve, enter the equation into *surf*'s text window preceded by `curve=` and followed by a semicolon. Then press the button *draw curve*. Some seconds later the curve will show up in the window titled *color image*. By default the curve is drawn inside the rectangle

-10.0 <= x,y <= 10.0

and is clipped at a circle with radius 10.0. The x-axis is horizontal pointing to the right, the y-axis is vertical and points upwards. By default the image size is 200 x 200 pixels. The image size can be altered by setting *width* and *height* in the main window.

The view can be altered in the *position window*: A different origin can be specified by setting *origin x* and *origin x*. A rotation with center at (0,0) can be specified by setting *rotation about z-axis*. The curve may be scaled by setting *scale factor x* and *scale factor y*. The appearance of the curve can be altered in the *curve window*.

The clipping area can be specified in the *clip window*. For a curve the only reasonable values are *sphere* and *none*.

An arbitrary color can be given to the curve by setting *curve red*,*curve green* and *curve blue* to appropriate values in the *curve window*. The curve width can be set by changing *curve width*. A high value of *curve gamma* sharpens the curve, whereas a low value blurs the curve.

3.2 Surfaces

To draw a surface, enter its equation into *surf*'s text window preceded by `surface=` and followed by a semicolon. Then press the button *draw surface*. Some more seconds later the surface will appear. By default, the surface is calculated inside the cube

-10.0 <= x,y,z <= 10.0

and clipped at a sphere of radius 10.0. The x-axis is horizontal pointing to the right, the y-axis is vertical and points upwards. The z-axis points to you. The spectator is located at (0,0,25) by default.

Changing the view can be done by altering the settings in the *position window*. A different origin may be specified by setting *origin x*, *origin y* and *origin z*. To rotate the surface one can set *rotation about x-axis*, *rotation about y-axis* and *rotation about z-axis* to appropriate values. Rotation is performed on the following order: y-axis, x-axis, z-axis. To scale the surface set *scale factor x*, *scale factor y* and *scale factor z* to desired values. It is also possible to switch from central perspective to parallel perspective.

Illumination and color can be altered in the *light window*. The direction of the normal vector given by the gradient of the surface equation defines one side of the surface which is regarded as outside. You can specify a color for this side by setting *surface red*, *surface green* and *surface blue*. The other side of the surface (inside) can be given a different color by specifying *inside red*, *inside green* and *inside blue*.

Currently only the Phong illumination model is implemented. Therefore the intensity of the surface in one point consists of four components which are calculated separately:

- ambient light,
- diffuse light,
- reflected light and
- transmitted light.

Ambient light is a constant which represents the light a point on the surface receives from the whole environment (the sky, the floor, the lawn ...) but not from the light sources. Diffuse light is the light the point receives from the light sources and which is reflected equally in every possible direction. The amount of diffuse light is independent of the spectator position, it is proportional to the cosine of the angle between the normal vector and the vector from the point to the light source. Reflected light is the light from the light sources which is reflected specular from the surface point. Its amount is proportional to a power of the cosine of the angle between the vector from the point to the spectator and the specular reflection vector from the light source. If a high power of the cosine is taken, the surface will appear shiny, whereas a low power of the cosine lets the surface look rough. Therfore this power is labelled *smoothness*. Transmitted light comes in if a surface is transparent. A constant called *transparence* specifies the percentage of light which passes through the surface. Algebraic surfaces are infinitesimally thin. However our eye is not used to such objects, so we pretend that our surfaces have a constant *thickness*. Specifying a positive thickness for a transparent surface results in a loss of transparency in the places where the surface normal does not point to the spectator.

These four light components are added with weights *ambient*, *diffuse*, *reflected* and *transmitted*.

The number of light sources is limited to nine. For every light source, the position, the color and the intensity can be specified.

The *clip window* allows to specify a different clipping area. Here the center and radius of the clipping area may be specified. Additionally a front and a back clipping plane may be specified.

3.3 Hyperplane sections

To draw one or more hyperplane sections of an algebraic surface, just specify the hyperplane by setting the global variable `plane` to its equation. The section is drawn when the command `cut_with_plane` is interpreted. For example:

```
rot_x=0.3;                         // a nice rotation
rot_y=0.2;
surface=x^2*y^2+y^2*z^2+z^2*x^2-16*x*y*z;
clear_screen;                      // draw the steiner roman surface
draw_surface;
curve_red=0;
curve_green=255;
curve_blue=0;
curve_width=5;
curve_gamma=1.2;
plane=x+y+z;                       // draw a green hyperplane section
cut_with_plane;
plane=x+y+z+4.0;                   // draw another one
cut_with_plane;
```

The color of the hyperplane section can be set by specifying `curve_red`, `curve_green` and `curve_blue`. The width of the section is altered by setting `curve_width` to any suitable value. A high value of `curve_gamma` (eg. 10.0) makes the curve look very pixelized, whereas a small value (eg. 1.0) makes the section look blurred.

3.4 Multiple curves/surfaces

Multiple curves can be drawn in script files just by *NOT* clearing the screen. This works fine for plane curves. Just consider the following example:

```
do_background=yes;
clear_screen;
curve=y^2-x^2*(x-1);
draw_curve;             // draw a cubic
do_background=no;
curve=x;
draw_curve;             // draw y-axis
curve=y;
draw_curve;             // draw y-axis
```

Not that every curve will be drawn just over all curves that have been draw so far.

Multiple surfaces can be drawn by specifying up to 9 surfaces in the variables **surface**, **surface2** ... **surface9**. Additionally it is possible to draw on every surface any number of hyperplane sections.

```
rot_x=0.69;                        // a nice rotation
rot_y=0.35;
illumination=ambient_light +       // specify illumination
             diffuse_light +       // model
             reflected_light +
             transmitted_light;
transparence=35;                   // set transparence for surface no 1
transparence2=35;                  // set transparence for surface no 2
surface=x^2+y^2+z^2-30;            // first surface: a sphere
surface2_red=255;                  // second surface: a red steiner surface
surface2_green=0;
surface2_blue=0;
surface2=x^2*y^2+x^2*z^2+y^2*z^2-16*x*y*z;
clear_screen;
draw_surface;                      // draw the surface
curve_width=5;
curve_red=0;
curve_green=255;
curve_blue=0;
plane=x+y+z-6.0;                   // draw a green hyperplane section
surf_nr=1;                         // on the sphere
cut_with_plane;
curve_red=0;
curve_green=255;
curve_blue=255;
plane=x+y+z+4.0;                   // draw a turquoise hyperplane section
surf_nr=2;                         // on the steiner surface
cut_with_plane;
```

3.5 Graphs and isolines

Given a polynomial function `f(x,y)` and a set of levels `z1`, ... ,`zn`, **surf** can visualize the graph `z=f(x,y)` and all isoline for the levels `z1`, ... ,`zn` as follows:

```
rot_x=-0.8;
clear_screen;
poly f=x^2+y^2; // graph of (x,y)->x^2+y^2
surface=z-f;
draw_surface;      // draw the graph
curve_width=3;     // width of isoline
plane=z-1;
cut_with_plane;    // draw isoline f(x,y)=1
plane=z-2;
cut_with_plane;    // draw isoline f(x,y)=2
plane=z-3;
cut_with_plane;    // draw isoline f(x,y)=3
plane=z-4;
```

```
cut_with_plane;    // draw isoline f(x,y)=4
plane=z-5;
cut_with_plane;    // draw isoline f(x,y)=5
plane=z-6;
cut_with_plane;    // draw isoline f(x,y)=6
plane=z-7;
cut_with_plane;    // draw isoline f(x,y)=7
plane=z-8;
cut_with_plane;    // draw isoline f(x,y)=8
plane=z-9;
cut_with_plane;    // draw isoline f(x,y)=9
```

If however your function `f` is not polynomial, try to expand calculate its Taylor series. Since the new root algorithms work fine with polynomials of degree up to 30, you might approximate `f` by its Taylor series. If your function is piecewise defined, better use another program.

3.6 Interactive positioning

The *position* window provides an interface to adjust the curve/surface position. You can set the 9 buttons into the three modes *translate*, *rotate* and *scale*.

3.7 Preview

If you try to draw a surface and give the equation to *surf*, the resulting image normally does not look nice at all. You have to find the right scaling, rotation and so on. Often you want to see immediately what happens if you change some value. But it simply takes *surf* too long to calculate one image. Here comes the preview in. Setting the preview buttons in the *main window* to *3x3* has the effect that only every 9th pixel is calculated, setting it to *9x9* only every 81st pixel is calculated. But one can still get an impression of what the image looks like, AND computation is speeded up by the factor 9 resp. 81.

Up to two preview buttons can be pressed at one time. If for example *9x9* and *1x1* are pressed, then the image will be calculated in three steps. First, every 81st pixel, after that every 9th pixel and finally every pixel will be calculated.

3.8 Anti aliasing surfaces

Especially in animations aliasing is very disturbing. Therefore if in the *display window*, *antialiasing level* is set to a value n > 1, then in a second pass all pixels differing by a value of at least *antialiasing threshold* from one of their neighbours are refined. Exactly n^2+1 intensity values are calculated. In most cases an antialiasing level of 4 will remove aliasing.

3.9 Animations

On a nifty machine *surf* is fast enough to provide a real time animation of an algebraic curve of degree < 5. For example

```
// -------------------------
// animation of a cubic curve
// -------------------------
```

```
clear_screen;
double a=-10.0;
loop:
    curve=y^2-(x^2-1)*(x-a);
    clear_pixmap;
    draw_curve;
    a=a+0.1;
if( a <= 10.0 ) goto loop;
```

calculates some 200 curves. In a 200x200 window, *surf* shows me about five frames per second on a sparc 20. However, real time animations of algebraic surfaces are still beyond computation power (or do you call a 200-processor-machine your own?). But you can calculate a series of images with *surf* and convert this series of images to the movie format of your choice.

```
// --------------------------
// the 4-nodal cubic rotating
// --------------------------
width=200;
height=200;                  // set image size
double sf=0.3;
scale_x=sf;
scale_y=sf;
scale_z=sf;                  // set scaling
double Pi=2*arccos(0);
double w2=sqrt(2);           // define some constants
poly p=1-z-w2*x;
poly q=1-z+w2*x;
poly r=1+z+w2*y;
poly s=1+z-w2*y;             // define tetrahedral coordinates
poly cubic=4*(p^3+q^3+r^3+s^3)-(p+q+r+s)^3; // the cubic
int i=0;
loop:
    surface=rotate(cubic,2*Pi/100*i,zAxis); // rotate the cubic
    clear_screen;
    draw_surface;                           // draw the cubic
    filename="cubic"+itostrn(3,i)+".ras";
    save_color_image;                       // save the image
    i=i+1;
if( i < 100 ) goto loop;                    // repeat 100 times
```

Here some 100 SUN rasterfiles are created. Afterwards you could use some tool to convert these single images to a movie.

3.10 Stereo pictures

Have you ever watched one of those films with that red and green glasses? *surf* tries to accomplish exactly this effect when you set *eye distance* in the *display window* to a value greater than zero. The following situation is simulated: The spectator is located at (0,0,*spectator z*) and the distance between his eyes is *eye distance*. The surface will appear at the z-coordinate *distance from screen*. Furthermore it is possible to adjust to specific red-green or red-blue glasses by setting *left eye red value*, *right eye green value* end *right eye blue value*. In particular it is assumed that the right eye wears the red glass.

3.11 Black & white images

If a color image of a surface/curve has been calculated, this image can be mapped to a black and white image by pressing the button *dither surface* or *dither curve*. The second one is just designed for dithering curves. The appearance of the black and white image can be altered/adjusted in several ways in the *dither window*. Since the mapping itself is done by dithering, the dithering algorithm can be specified. Currently available are seven algorithms coming in three groups:

3.11.1 Dithering with blue noise

- Floyd-Steinberg filter
- Jarvis, Judis and Ninke filter
- Stucki filter

All three filters are based on the same idea of error distribution. Floyd Steinberg is the simplest one, whereas Stucki differs from Jarvis only by its weights. They tend to produce disturbing patterns if they process large areas of intensity near 0.5. Therefore one can let them proceed in a serpentine fashion, which reduces the patterns. Nearly all patterns disappear if the weights are disturbed randomly. The algorithms are best for use with low resolution printers, typically 300 dpi. Some (most?) 600 dpi laser printers do not like these algorithms, since they do not like isolated pixels.

3.11.2 Dithering with ordered dither

- Clustered dot ordered dither
- Dispersed dot ordered dither

The clustered dot ordered dither is a fast method and produces satisfying results in combination with high resolution printers (600 dpi and more). The second algorithm is for use with low resolution printers. Both perform no error distribution. Depending on the printer resolution and the number of emulated gray levels, one can choose the pattern size:

- 4 x 4 pixels: 16 gray levels,
- 8 x 8 pixels: 64 gray levels or
- 16 x 16 pixels: 256 gray levels.

3.11.3 Hybrid methods

- Knuth's dot diffusion
- Knuth's smooth dot diffusion

Both algorithms combine clustered dot ordered dither and error distribution. Depending on the printer resolution on can choose the number of barons in a 8x8 matrix to be

- 1 for resolutions of 1200 dpi or above or
- 2 for resolutions of 600 dpi or above.

The barons are the bad guys in a matrix which get all the error left over from the good guys.

3.11.4 The black & white problem

The surfaces on black and white images often don't look very impressive; often it is hard to detect the edges of a surface. An algorithm called enhancing the edges avoids this drawback. This algorithm takes a value *alpha* in [0,1] as input. Best results are achieved with alpha around 0.9.

The intensity of the background on the black and white image can be specified by altering the value *background* to any value in [0.1]. Here 0 is black whereas 1 means white.

The *tone scale adjustment* maps intensity values between 0 and 0.1 to 0, values between 0.1 and 0.9 linear to [0,1] and values between 0.9 and 1 to 1. This is used to enhance the contrast of an image. An additional gamma correction can be also performed to correct the linearity of an output device.

By specifying *pixel size* one can correct the printer pixel size: A value of 50 means that the radius of a pixel is exactly half the distance between two neighbouring pixels. A value of 100 says that the radius of a pixel is exactly the distance between two neighbouring pixels.

3.12 Algorithms

The heart of *surf* is an algorithm which determines all roots of a polynomial in one variable. Currently you can choose between seven methods in the *numeric window*. The first six methods use a chain of derivatives to determine intervals where the polynomial has exactly one root. They differ by the iteration method which is used to find the roots in these intervals. Some of the iteration methods were just implemented out of academic interest. However, they all work. The last method uses Rockwoods all roots algorithm: the polynomial is converted into a bezier function and the roots of the bezier function are approximated by the roots of the control polygon.

For curves/surfaces of degree less than ten, all methods work. When the degree gets higher, best results are achieved by the bisection, the Newton and the bezier all roots method. At last, for a degree higher than 30 only the bisection methods seems to work (up to degree 50). If a curve has multiple components, the bisection and the Newton method tend to produce the best results.

Moreover it is possible to specify a numerical precision *epsilon* which is used in all root finders. Additionally the maximal number of *iterations* of the iteration methods can be specified.

3.13 Output

surf can store color images in one of several file formats. In the *save color image window* you can choose between

- **XWD** (X Window Dump),
- **SUN rasterfile**,
- **PPM** (Portable PixMap) and
- **JPEG**.

Additionally the color space can be chosen among

- Netscape 216 color cube (8 bit),
- optimized by an octree algorithm (8 bit) and
- True color (24 bit).

The first colormap is just the 6x6x6 colormap Netscape uses. The second one results from an octree algorithm which chooses the most used 216 colors among all colors of the image. Storing an image in True color results in better quality, but bigger file size.

surf can store black and white images in different file formats. We have implemented

- **Postscript**,
- **EPS** (Encapsulated Postscript),
- **TIFF**,
- **XBM**,
- **PGM** and
- **PBM**.

For postscript and encapsulated postscript also the resolution may be specified among

- 75 dpi,
- 100 dpi,
- 150 dpi,
- 300 dpi,
- 600 dpi and
- 1200 dpi.

These settings may be chosen in the *save dithered image window*. When using postscript, the image will (regardless its size) appear centred on the side (which is assumed to be a4).

4 List of all reserved words

A reserved word in *surf*'s language is either a command or a global variable. A command is invoked mostly without parameters. Global variables are either constant or may be altered. The commands correspond to pushbuttons of *surf*'s GUI, global variables correspond to other panel items.

4.1 Reserved words corresponding to the main window

```
res. word          | type    | description
--------------------------------------------------------------------------
clear_screen       | command | erase the image
clear_pixmap       | command | erase the image in memory (useful for
                   |         | real-time-animations of algebraic curves)
draw_curve         | command | draw the curve defined by the global
                   |         | polynomial curve
draw_surface       | command | draw the surfaces defined by the global
                   |         | polynomials surface, surface2, ...
cut_with_plane     | command | draw the hyperplane section defined
                   |         | defined by the linear polynomial plane
```

```
dither_surface      | command | convert color image to a dithered
                    |         | black and white image
dither_curve        | command | convert color image to a dithered
                    |         | black and white image (for curves only)
save_color_image    | command | save color image in file defined by the
                    |         | global string filename
save_dithered_image | command | save dithered black and white image in
                    |         | file defined by the global string filename
set_size            | command | not needed any more (still there for
                    |         | compatibility issues)
```

```
res. word | type    | range         | default | description
-------------------------------------------------------------------------
curve     | poly    |           any |       0 | polynomial of curve
surface   | poly    |           any |       0 | polynomial of surface
surface2  | poly    |           any |       0 | polynomial of surface2
...       | ...     |           ... |     ... | ...
surface9  | poly    |           any |       0 | polynomial of surface9
plane     | poly    |        linear |       0 | equation of hyperplane
width     | int     | {64,...,3000} |     200 | width of surface image
height    | int     | {64,...,3000} |     200 | height of surface image
filename  | string  |           any |      "" | filename used in
          |         |               |         | save_color_image,
          |         |               |         | save_dithered_image
surf_nr   | int     |     {1,...,9} |       1 | surface which is used
          |         |               |         | for cut_with_plane
```

4.1.1 Examples

```
width=400;                // Set image width
height=300;               //  and height
surface=x^2+y^2+z^2-81;   // Set global variable surface to a sphere
draw_surface;             // Draw the sphere onto the screen
plane=x+y+z;              // Choose a hyperplane
cut_with_plane;           // Draw the hyperplane section
filename="sphere.ras";
save_color_image;         // Save the color image in file sphere.xwd
dither_surface;           // Perform dithering on the color image
filename="sphere.ps";
save_dithered_image;      // Save the dithered image in sphere.ps
```

4.2 Reserved words corresponding to the position window

```
res. word    | type   | range        | def. | description
-------------------------------------------------------------------------
origin_x     | double | ]-9999,9999[ |    0 |  \
origin_y     | double | ]-9999,9999[ |    0 |   > position of origin
origin_z     | double | ]-9999,9999[ |    0 |  /
spec_z       | double |     ]0,9999[ |  100 | spectator dist. from origin
rot_x        | double | ]-9999,9999[ |    0 | rotation angle of surface
```

```
              |        |               |     | about the x-axis
rot_y         | double | ]-9999,9999[ |    0 | rotation of surface
              |        |               |     | about the y-axis
rot_z         | double | ]-9999,9999[ |    0 | rotation of surface
              |        |               |     | about the z-axis
scale_x       | double | ]-9999,9999[ |    1 | ratio surface is scaled in
              |        |               |     | direction of the x-axis
scale_y       | double | ]-9999,9999[ |    1 | ratio surface is scaled in
              |        |               |     | direction of the y-axis
scale_z       | double | ]-9999,9999[ |    1 | ratio surface is scaled in
              |        |               |     | direction of the z-axis
perspective   |    int |         {0,1} |    0 | perspective to use
parallel      |    int |             0 |    0 | constant
central       |    int |             1 |    1 | constant
first         |    int |       {0,1,2} |    0 | first performed  \
second        |    int |       {0,1,2} |    1 | second performed  > action
third         |    int |       {0,1,2} |    2 | third performed  /
translate     |    int |             0 |    0 | constant
rotate        |    int |             1 |    1 | constant
scale         |    int |             2 |    2 | constant
```

4.2.1 Examples

```
double Pi=2*arccos(0);
origin_x = -3;
origin_y = -4;               // Set origin to point (-3,-4,2)
origin_z =  2;
spec_z = 25;                 // Spectator is now at (-3,-4,27)
rot_x = Pi/2;                // Rotate 90 degrees about x-axis
rot_y = Pi/4;                // Rotate 45 degrees about y-axis
rot_z = Pi;                  // Rotate 180 degrees about z-axis
scale_x = 1.0;               // Don't scale in x-direction
scale_y = 1.5;               // Shrink surface in y-direction
scale_z = 1/2;               // Oversize surface in z-direction
first   = rotate;            // rotate first
second  = scale;             // then scale
third   = translate;         // then translate
```

4.3 Reserved words corresponding to the display window

```
res. word               | type   | range        | def.  | description
--------------------------------------------------------------------------
dither_colors           | int    |     {yes,no} |   yes | color dithering
dither_steps            | double | [5,...,255] |  20.0 | steps of dithering
normalize               | int    |     {yes,no} |    no | normalize image
normalize_factor        | double |    ]0,...,5] |   1.0 | multiply with
antialiasing            | int    |     {1,..,8} |     1 | level of
                        |        |              |       | antialiasing
antialiasing_threshold  | double |        ]0,1[ |   0.1 | threshold
antialiasing_radius     | double | [0.5,...,2] |   2.0 | radius
```

```
depth_cueing            | int    |    {yes,no} |    no | use depth cueing
depth_value             | double |  [-1000,10[ | -14.0 | depth of mist
stereo_eye              | double |  [-100,100] |   0.0 | eye distance
stereo_z                | double |    [-30,30] |   5.0 | dist. from screen
stereo_red              | double |       [0,1] |   1.0 | left eye red
stereo_green            | double |       [0,1] |   0.7 | right eye green
stereo_blue             | double |       [0,1] |   0.0 | right eye blue
```

4.3.1 Examples

```
dither_colors          = yes;  // perform color dithering
dither_steps           = 60.0; // use soft dithering
normalize              = yes;
normalize_factor       = 1.5;  // light up image
antialiasing           = 4;    // do 4 fold antialiasing
antialiasing_threshold = 0.05; // with a low threshold
antialiasing_radius    = 1.5;  // and a small radius
depth_cueing           = yes;  // perform depth cueing
depth_value            = -11.0;// from -11 on everythin is dark
stereo_eye             = 5.0;  // make a red-blue image
stereo_z               = 2.0;  // object 2 units before screen
stereo_red             = 1.0;
stereo_green           = 0.0;
stereo_blue            = 1.0;
```

4.4 Reserved words corresponding to the light window

```
res. word          | cat.   | range         | def. | description
-----------------------------------------------------------------------
illumination       | int    |      {0,..15} |    7 | illumination model
ambient_light      | int    |             1 |    1 | constant
diffuse_light      | int    |             2 |    2 | constant
reflected_light    | int    |             4 |    4 | constant
transmitted_light  | int    |             8 |    8 | constant
surface_red        | int    |  {0,...,255}  |  123 | \  outside
surface_green      | int    |  {0,...,255}  |  104 |  > color of surface
surface_blue       | int    |  {0,...,255}  |  238 | /  (medium slate blue)
inside_red         | int    |  {0,...,255}  |  230 | \  inside
inside_green       | int    |  {0,...,255}  |  180 |  > color of surface
inside_blue        | int    |  {0,...,255}  |   30 | /  (golden)
surface2_red       | int    |  {0,...,255}  |  123 | \  outside
surface2_green     | int    |  {0,...,255}  |  104 |  > color of surface2
surface2_blue      | int    |  {0,...,255}  |  238 | /  (medium slate blue)
inside2_red        | int    |  {0,...,255}  |  230 | \  inside
inside2_green      | int    |  {0,...,255}  |  180 |  > color of surface2
inside2_blue       | int    |  {0,...,255}  |   30 | /  (golden)
...                | ...    | ...           |  ... | ...
...                | ...    | ...           |  ... | ...
...                | ...    | ...           |  ... | ...
surface9_red       | int    |  {0,...,255}  |  123 | \  outside
```

```
surface9_green    | int    |  {0,...,255} |  104 |  > color of surface9
surface9_blue     | int    |  {0,...,255} |  238 | /  (medium slate blue)
inside9_red       | int    |  {0,...,255} |  230 | \  inside
inside9_green     | int    |  {0,...,255} |  180 |  > color of surface9
inside9_blue      | int    |  {0,...,255} |   30 | /  (golden)
ambient           | int    |  {0,...,100} |   35 | amount of ambient light
diffuse           | int    |  {0,...,100} |   60 | diffuse reflected light
reflected         | int    |  {0,...,100} |   60 | specular reflected light
transmitted       | int    |  {0,...,100} |   60 | spec. transmitted light
smoothness        | int    |  {0,...,100} |   13 | roughness of surface
transparency      | int    |  {0,...,100} |   80 | transparency of surface
ambient2          | int    |  {0,...,100} |   35 | amount of ambient light
diffuse2          | int    |  {0,...,100} |   60 | diffuse reflected light
reflected2        | int    |  {0,...,100} |   60 | specular reflected light
transmitted2      | int    |  {0,...,100} |   60 | spec. transmitted light
smoothness2       | int    |  {0,...,100} |   13 | roughness of surface2
transparency2     | int    |  {0,...,100} |   80 | transparency of surface2
...               | ...    | ...          |  ... | ...
...               | ...    | ...          |  ... | ...
...               | ...    | ...          |  ... | ...
ambient9          | int    |  {0,...,100} |   35 | amount of ambient light
diffuse9          | int    |  {0,...,100} |   60 | diffuse reflected light
reflected9        | int    |  {0,...,100} |   60 | specular reflected light
transmitted9      | int    |  {0,...,100} |   60 | spec. transmitted light
smoothness9       | int    |  {0,...,100} |   13 | roughness of surface9
transparency9     | int    |  {0,...,100} |   80 | transparency of surface9
light1_x          | double | [-9999,9999] | -100 | \
light1_y          | double | [-9999,9999] |  100 |  \ position and volume
light1_z          | double | [-9999,9999] |  100 |  / of the first light
light1_vol        | int    | {0,...,100}  |   50 | /  source
light1_red        | int    |   {0,..,255} |  255 | \
light1_green      | int    |   {0,..,255} |  255 |  > color of first
light1_blue       | int    |   {0,..,255} |  255 | /  light source
light2_x          | double | [-9999,9999] |    0 | \
light2_y          | double | [-9999,9999] |  100 |  \ position and volume
light2_z          | double | [-9999,9999] |  100 |  / of the second light
light2_vol        | int    | {0,...,100}  |    0 | /  source
light2_red        | int    |   {0,..,255} |  255 | \
light2_green      | int    |   {0,..,255} |  255 |  > color of second
light2_blue       | int    |   {0,..,255} |  255 | /  light source
...               | ...    | ...          |  ... | ...
...               | ...    | ...          |  ... | ...
...               | ...    | ...          |  ... | ...
light9_x          | double | [-9999,9999] |  100 | \
light9_y          | double | [-9999,9999] | -100 |  \ position and volume
light9_z          | double | [-9999,9999] |  100 |  / of the ninteh light
light9_vol        | int    | {0,...,100}  |    0 | /  source
light9_red        | int    |   {0,..,255} |  255 | \
light9_green      | int    |   {0,..,255} |  255 |  > color of nineth
light9_blue       | int    |   {0,..,255} |  255 | /  light source
```

4.4.1 Examples

```
illumination = ambient_light
             + diffuse_light
             + reflected_light
             + transmitted_light; // Select illumination
surface_red   = 205;
surface_green = 92;
surface_blue  = 92;               // Select indian red for surface outside
inside_red    = surface_red;
inside_green  = surface_green;
inside_blue   = surface_blue;     // Select indian red for surface inside
ambient      = 10;                // 40% ambient light
diffuse      = 60;                // 60% diffuse light
reflected    = 60;                // 60% reflected light
transmitted  = 70;                // 60% reflected light
smoothness   = 50;                // make surface shiny
transparence = 90;                // very transparent
thickness    = 20;                // but also very thick
light2_x = 100;
light2_y = 0;
light2_z = 200;
light2_volume = 100;              // turn on light no. 2 red at (100,0,200)
light2_red   = 255;
light2_green = 0;
light2_blue  = 0;
```

4.5 Reserved words corresponding to the clip window

```
reserved word  | cat.   | range        | def. | description
---------------------------------------------------------------------------
clip           | int    |    {0,...,5} |    0 | clipping area
ball           | int    |            0 |    0 | constant
cylinder_xaxis | int    |            1 |    1 | constant
cylinder_yaxis | int    |            2 |    2 | constant
cylinder_zaxis | int    |            3 |    3 | constant
cube           | int    |            4 |    4 | constant
none           | int    |            5 |    5 | constant
clip_front     | double | [-9999,9999] |   10 | \ additional clip region
clip_back      | double | [-9999,9999] |  -10 | /
radius         | double |     ]0,9999] |   10 | radius of clip region
center_x       | double | [-9999,9999] |    0 | \
center_x       | double | [-9999,9999] |    0 |  > center of clip region
center_x       | double | [-9999,9999] |    0 | /
```

4.5.1 Examples

```
clip = cube;
radius = 7;
center_x = -3;          // Set clipping area to cube with center at
```

```
center_y = 2;           // (-3,2,1) and edge length 14
center_z = 1;
clip_front = 4;         // Clip off points with z > 4
clip_back = -10;        // Clip off points with z > -10
```

4.6 Reserved words corresponding to the dither window

```
reserved word                | cat.   | range     | def. | description
-------------------------------------------------------------------------
dithering_method             | int    | {0,...,6} |    1 | dithering method
floyd_steinberg_filter       | int    |         0 |    0 | constant
jarvis_judis_ninke_filter    | int    |         1 |    1 | constant
stucki_filter                | int    |         2 |    2 | constant
clustered_dot_ordered_dither | int    |         3 |    3 | constant
dispersed_dot_ordered_dither | int    |         4 |    4 | constant
dot_diffusion                | int    |         5 |    5 | constant
smooth_dot_diffusion         | int    |         6 |    6 | constant
```

```
reserved word     | cat.   | range     | def. | description
-------------------------------------------------------------------------
serpentine_raster | int    |  {yes,no} |  yes | use of serpentine raster
random_weights    | int    |  {yes,no} |  yes | use of random weights
weight            | double |     [0,1] |  0.5 | amount of random weights
barons            | int    |     {0,1} |    1 | number of barons
one_baron         | int    |         0 |    0 | constant
two_baron         | int    |         1 |    1 | constant
pattern_size      | int    |   {0,1,2} |    1 | size of dithering tile
pattern_4x4       | int    |         0 |    0 | constant
pattern_8x8       | int    |         1 |    1 | constant
pattern_16x16     | int    |         2 |    2 | constant
enhance_edges     | int    |  {yes,no} |  yes | enhance edges of b w image
alpha             | double |     [0,1] |  0.9 | filter coefficient used in
                  |        |           |      | for enhancing the edges
background        | double |     [0,1] |  1.0 | background intensity of
                  |        |           |      | b w image
tone_scale_adjustment | int    |  {yes,no} |  yes | perform tone scale adjust.
gamma             | double |    ]0,oo[ |  1.3 | gamma-correction
pixel_size        | int    |  ]50,100] |   73 | correction for printers
                  |        |           |      | that produce too fat pixels
```

4.6.1 Examples

```
dithering_method = stucki_filter;    // select stucki filter
serpentine_raster = yes;             // turn on serpentine raster
random_weights = yes;                // turn on random weights
weight = 0.5;                        // select 50% weights
enhance_edges = yes;                 // turn on enhancing edges
alpha = 0.8;                         // edges less visible than default
background = 0.5;                    // gray background for b w image
tone_scale_adjustment = yes;         // perform tone scale adjustment
```

```
gamma = 1.5;                         // more gamma-correction than default

dithering_method = dispersed_dot;    // select dispersed dot ordered dither
pattern_size = pattern_16x16;        // select a 16x16-tile

dithering_method = dot_diffusion;    // select dot-diffusion
barons = two_barons;                 // select a 2-barons tile
```

4.7 Reserved words corresponding to the save color image window

```
reserved word       | type | range   | def. | description
-----------------------------------------------------------------------
color_file_format   | int  |   {0,1} |    1 | file format
xwd                 | int  |       0 |    0 | constant
sun                 | int  |       1 |    1 | constant
color_file_colormap | int  | {0,1,2} |    0 | colormap type
netscape            | int  |       0 |    0 | constant
optimized           | int  |       1 |    1 | constant
truecolor           | int  |       2 |    2 | constant
```

4.7.1 Examples

```
color_file_format   = xwd;
color_file_colormap = truecolor; // format is 24 bit XWD
```

4.8 Reserved words corresponding to the save dithered image window

```
reserved word         | type | range     | def. | description
-----------------------------------------------------------------------
resolution            | int  | {0,...,5} |    3 | (printer) resolution
res_75dpi             | int  |         0 |    0 | constant
res_100dpi            | int  |         1 |    1 | constant
res_150dpi            | int  |         2 |    2 | constant
res_300dpi            | int  |         3 |    3 | constant
res_600dpi            | int  |         4 |    4 | constant
res_1200dpi           | int  |         5 |    5 | constant
dithered_file_format  | int  | {0,...,4} |    2 | file format
postscript            | int  |         0 |    0 | constant
encapsulated          | int  |         1 |    1 | constant
xbm                   | int  |         2 |    2 | constant
tiff                  | int  |         3 |    3 | constant
bm2font               | int  |         4 |    4 | constant
```

4.8.1 Examples

```
resolution           = res_300dpi;     // select 300 dpi
dithered_file_format = bm2font;        // TeX pk
```

4.9 Reserved words corresponding to the numeric window

```
reserved word              | cat.   | range      | def. | description
---------------------------------------------------------------------------
root_finder                | int    | {0,...,6} |    6 | used root finder
d_chain_bisection          | int    |          0 |    0 | constant
d_chain_regula_falsi       | int    |          1 |    1 | constant
d_chain_pegasus            | int    |          2 |    2 | constant
d_chain_illinois           | int    |          3 |    3 | constant
d_chain_anderson_bjoerck   | int    |          4 |    4 | constant
d_chain_newton             | int    |          5 |    5 | constant
bezier_all_roots           | int    |          6 |    6 | constant
epsilon                    | double |      ]0,1[ | 1e-4 | precision of
                           |        |            |      | root finder
iterations                 | int    |   [1,2000] |  200 | max. number of
                           |        |            |      | iterations
```

4.9.1 Examples

```
root_finder = d_chain_bisection;      // Slow, but safe
epsilon = 1.0e-7;                     // Work very precise
iterations = 80;                      // max. 80 iterations on each root
```

4.10 Reserved words corresponding to the curve window

```
reserved word | type   | range       | def. | description
---------------------------------------------------------------------------
curve_red     | int    | {0,...,255} |  255 | \
curve_green   | int    | {0,...,255} |  255 |  > curve color
curve_blue    | int    | {0,...,255} |  255 | /
curve_width   | double | {1,2,...}   |    1 | width of curve
curve_gamma   | double | ]0,oo[      |  4.0 |
```

4.10.1 Examples

```
curve_red  =0;
curve_green=255;
curve_blue =0;                       // make the curve look green
curve_width=6.0;                     // thick curve
curve_gamma=2.0;                     // intensity increases slower
```