1. Introduction. (Please note that you can find a table of contents at the end of this document). This program PP3 ("parvum planetarium") takes the data of various celestial data files and turns them into a LaTeX file that uses PSTricks to draw a nice sky chart containing a certain region of the sky. Current versions are available on its homepage.

You call PP3 with e. g.

```
pp3 mychart.pp3
```

The data files (`stars.dat`, `nebulae.dat`, `boundaries.dat`, `labeldimens.dat`, `lines.dat`, and `milkyway.dat`) must be in the proper directory. The proper directory was compiled into PP3. With Linux, normally it's `/usr/local/share/pp3/`, with Windows the current directory simply. But the environment variable `PP3DATA` can override that.

The resulting chart is by default sent to standard output which you may redirect into a file. But you can define an output filename explicitly in the input script.

If you want to use other data with this program, you may well provide your own catalogue files. Their file formats are very simple, and they are explained in this document together with the respective *read_*. . . ( ) function.

If you give a single dash "`-`" as the only parameter to PP3, the input script is read from standard input. So if you write

```
pp3 - > test.tex && latex test && dvips test
```

and type `^D` (Control-D)[1], a file `test.ps` should be produced that contains a sample chart.

Very important is to know how to write an input script. Please consult the following section "The input script" for this. Here is an example input script:

```
# Chart of the Scorpion, printable on a
# black and white printing device

set constellation SCO           # This is highlighted
set center_rectascension   17
set center_declination    -28
set grad_per_cm             2.5

switch milky_way on
switch eps_output on            # Please call LaTeX and dvips for us
switch colored_stars off        # All the same colour ...
color stars 0 0 0               # ... namely this one
color nebulae 0 0 0
color background 1 1 1
color grid 0.5 0.5 0.5
color ecliptic 0.3 0.3 0.3
color constellation_lines 0.7 0.7 0.7
color labels 0 0 0
color boundaries 0.8 0.8 0.8
color highlighted_boundaries 0 0 0
color milky_way 0.5 0.5 0.5

filename output test.tex        # Here should it go


objects_and_labels              # Now for the second part

delete M 18  NGC 6590  NGC 6634 ;  # Delete superfluous
reposition SCO 20 S ;           # Force sig Sco to be labelled.
text "\\Huge Sco" at 16.2 -41.5 color 0 0 0 towards NW ;
```

[1] it's Control-Z-Return on Windows

2. The resulting LaTeX file doesn't use packages special to PP3. In fact the preamble is rather small. This makes it possible to copy the (maybe huge) `\vbox` with the complete map into an own LaTeX file. However this may be a stupid decision because (especially if the Milky Way is switched on) this consumes much of TeX's pool size.

Some PP3 figures will need a lot of TeX's memory. Normally this is problematic only if the Milky Way is included. If you show a too large portion of the sky, and a lot of Milky Way area is involved, it may even happen that LaTeX simply cannot process it. You should use Milky Way data with lower resolution in this case.

Make sure that the PSTricks package is properly installed.

Please note that you cannot use pdfLaTeX with PP3, because PSTricks can't be used with pdfLaTeX. This is not a problem. First, you can convert all EPS files easily to PDF by hand, and secondly PP3 can even call PS2PDF for you.

3. In order to use the program, you must have a complete and modern LaTeX distribution installed, and a modern Ghostscript. On a Linux system, both programs are possibly already available, and if not you may install them with a package management program.

On Windows, you will probably have to install them by hand. You can download the MikTeX distribution or get the TeXLive CD. If you install Ghostscript, notice that GSView is a very sensible program, too.

4. Some flaws and bugs in this program are already known to its author.

The input script processing is shaky. The comment character '`#`' must be at the beginning of a line or must be preceded by whitespace. The special token "`objects_and_labels`" must not occur within strings. If an error is found in the input script, PP3 doesn't tell the line number. It should be possible to include more than one file, and it should allow for a nesting depth greater than one.

At the moment almost all data structures are kept in memory completely. For the author's needs this is perfectly sufficient, however if you want to use data bases with hundreds of thousands of objects, you will run into trouble. On the other hand it's only necessary to keep all object in memory that are actually drawn. So memory usage could be reduced drastically.

5. Okay, let's start with the header files . . .

```
#include <iostream>
#include <string>
#include <fstream>
#include <sstream>
#include <vector>
#include <map>
#include <list>
#include <iomanip>
#include <cstdlib>
#include <cmath>
#include <cfloat>
  using namespace std;
```

6. Global parameters and default values. I have to break a strict C++ rule here: Never use **#define**! However I really found no alternative to `OUT`. No **const** construct worked, and if it had done, I'd have to use it in every single routine. And ubiquitous $*params.out$'s are ugly.

```
#define OUT (*params.out)
```

7. The following makes it possible to compile a directory prefix into PP3 for the data files. By default, the data files must be in the current directory. You may say e. g.

```
g++ -DPP3DATA=\"/usr/share/pp3/\" -O2 -s pp3.cc -o pp3
```

then they are searched in `/usr/share/pp3/`. If set, an environment variable called `PP3DATA` has highest priority though.

```
  const char *pp3data_env_raw = getenv("PP3DATA");
  const string pp3data_env = (pp3data_env_raw ≡ 0 ? "" : pp3data_env_raw);
#ifdef PP3DATA
  const string pp3data(PP3DATA);
#else
  const string pp3data;
#endif
  const string filename_prefix(¬pp3data_env.empty() ? pp3data_env + '/' : (pp3data.empty() ? "" :
      pp3data + "/"));
```

8. I declare *and* define the structure *parameters* here.

```
⟨ Define color data structure 69 ⟩
struct parameters {
  double center_rectascension, center_declination;
  double view_frame_width, view_frame_height;
  double grad_per_cm;
  double label_skip, minimal_nebula_radius, faintest_cluster_magnitude, faintest_diffuse_nebula_magnitude,
      faintest_star_magnitude, star_scaling, minimal_star_radius, faintest_star_disk_magnitude,
      faintest_star_with_label_magnitude, shortest_constellation_line;
  string constellation;
  int font_size;
  double penalties_label, penalties_star, penalties_nebula, penalties_boundary, penalties_boundary_rim,
      penalties_cline, penalties_cline_rim, penalties_threshold, penalties_rim;
  string filename_stars, filename_nebulae, filename_dimensions, filename_lines, filename_boundaries,
      filename_milkyway, filename_preamble, filename_include;
  string filename_output;
  ostream *out;
  istream *in;
  bool input_file;
  color bgcolor, gridcolor, eclipticcolor, boundarycolor, hlboundarycolor, starcolor, nebulacolor, labelcolor,
      textlabelcolor, clinecolor, milkywaycolor;
  double linewidth_grid, linewidth_ecliptic, linewidth_boundary, linewidth_hlboundary, linewidth_cline,
      linewidth_nebula;
  string linestyle_grid, linestyle_ecliptic, linestyle_boundary, linestyle_hlboundary, linestyle_cline,
      linestyle_nebula;
  bool milkyway_visible, nebulae_visible, colored_stars, show_grid, show_ecliptic, show_boundaries,
      show_lines, show_labels;
  bool create_eps, create_pdf;
  parameters()
  : ⟨ Default values of all global parameters 9 ⟩{ }
  int view_frame_width_in_bp()
  {
    return int(ceil(view_frame_width/2.54 * 72));
  }
  int view_frame_height_in_bp()
  {
    return int(ceil(view_frame_height/2.54 * 72));
  }
} params;
```

9. ⟨ Default values of all global parameters 9 ⟩ ≡

```
center_rectascension(5.8), center_declination(0.0), view_frame_width(15.0), view_frame_height(15.0),
    grad_per_cm(4.0), constellation("ORI"), font_size(10),
label_skip(0.06), minimal_nebula_radius(0.1), faintest_cluster_magnitude(4.0),
    faintest_diffuse_nebula_magnitude(8.0), faintest_star_magnitude(7.0), star_scaling(1.0),
    minimal_star_radius(0.015), faintest_star_disk_magnitude(4.5), faintest_star_with_label_magnitude(3.7),
    shortest_constellation_line(0.1),
penalties_label(1.0), penalties_star(1.0), penalties_nebula(1.0), penalties_boundary(1.0),
    penalties_boundary_rim(1.0), penalties_cline(1.0), penalties_cline_rim(1.0), penalties_threshold(1.0),
    penalties_rim(1.0),
filename_stars(filename_prefix + "stars.dat"), filename_nebulae(filename_prefix + "nebulae.dat"),
    filename_dimensions("labeldimens.dat"), filename_lines(filename_prefix + "lines.dat"),
    filename_boundaries(filename_prefix + "boundaries.dat"),
    filename_milkyway(filename_prefix + "milkyway.dat"),
filename_preamble( ), filename_include( ), filename_output( ), out(&cout), in(0), input_file(false),
bgcolor("bgcolor", 0, 0, 0.4), gridcolor("gridcolor", 0, 0.298, 0.447), eclipticcolor("eclipticcolor", 1, 0, 0),
    boundarycolor("boundarycolor", 0.5, 0.5, 0), hlboundarycolor("hlboundarycolor", 1, 1, 0),
    starcolor("starcolor", 1, 1, 1), nebulacolor("nebulacolor", 1, 1, 1), labelcolor("labelcolor", 0, 1, 1),
    textlabelcolor(1, 1, 0), clinecolor("clinecolor", 0, 1, 0), milkywaycolor(0, 0, 1),
linewidth_grid(0.025), linewidth_ecliptic(0.018), linewidth_boundary(0.035), linewidth_hlboundary(0.035),
    linewidth_cline(0.035), linewidth_nebula(0.018),
linestyle_grid("solid"), linestyle_ecliptic("dashed"), linestyle_boundary("dashed"),
    linestyle_hlboundary("dashed"), linestyle_cline("solid"), linestyle_nebula("solid"),
milkyway_visible(false), nebulae_visible(true), colored_stars(true), show_grid(true), show_ecliptic(true),
    show_boundaries(true), show_lines(true), show_labels(true),
create_eps(false), create_pdf(false)
```

This code is used in section 8.

10. The input script. The input script is a text file that is given as the first and only parameter to PP3. Its format is very simple: First, a '#' introduces a comment and the rest of the line is ignored. Secondly, every command has an opcode–parameter(s) form. Thirdly, opcodes and parameters are separated by whitespace (no matter which type and how much). Forthly, parameters and celestial objects must be separated by "`objects_and_labels`".

11. Part I: Global parameters. Every input script can be divided into two parts, however the second may be absent. They are separated from each other by the token "`objects_and_labels`". Here we process the first part of the input script.

First two small helping routines that just read simple values from the file.

```
⟨ Routines for reading the input script 11 ⟩ ≡
  bool read_boolean(istream &script)
  {
    string boolean;
    script >> boolean;
    if (boolean ≡ "on") return true;
    else if (boolean ≡ "off") return false;
    else
      throw string("Expected␣\"on\"␣or␣\"off\"␣in␣\"switch\"␣construct␣""in␣input␣script");
  }
```

See also sections 12, 13, 21, 23, 24, 25, and 26.

This code is used in section 115.

12. You can give strings in a similar way as on a shell command line: If it doesn't contain spaces, just input it. In the other case you have to enclose it within `"..."`. The same is necessary if it starts with a `"`. Within the double braces, backslashes and double braces have to be escaped with a backslash. This is not necessary if you had a simple string.

So you may write e. g.: `Leo`, `"Leo Minor"`, `\alpha`, and `"{\\sfseries Leo}"`. An empty string can only be written as `""`.

⟨ Routines for reading the input script 11 ⟩ +≡

```
string read_string(istream &script)
{
  const string UnexpectedEOS("Unexpected␣end␣of␣input␣script␣while␣re\
      ading␣a""␣string␣parameter");
  char c;
  string contents;
  while (script.get(c))
    if (¬isspace(c)) break;
  if (¬script) throw UnexpectedEOS;
  if (c ≠ '"') {
    script ≫ contents;
    if (script) contents.insert(contents.begin(), c);
    else contents = c;
  }
  else {
    while (script.get(c)) {
      if (c ≡ '"') break;
      if (c ≡ '\\') {
        if (¬script.get(c)) throw UnexpectedEOS;
        switch (c) {
        case '\\': case '"': contents += c;
          break;
        default: throw string("Unknown␣escape␣sequence␣in␣string");
        }
      }
      else contents += c;
    }
    if (¬script) throw UnexpectedEOS;
  }
  return contents;
}
```

13. Here the actual routine for this first part. The top-level keywords are: "`color`", "`line_width`", "`line_style`", "`switch`", "`penalties`", "`filename`", and "`set`".

```
⟨ Routines for reading the input script 11 ⟩ +≡
  void read_parameters_from_script(istream &script)
  {
    string opcode;
    script ≫ opcode;
    while (opcode ≠ "objects_and_labels" ∧ script) {
      if (opcode[0] ≡ '#') {      /* skip comment line */
        string rest_of_line;
        getline(script, rest_of_line);
      }
      else ⟨ Set color parameters 14 ⟩
      else ⟨ Set line widths 15 ⟩
      else ⟨ Set line styles 16 ⟩
      else ⟨ Set on/off parameters 17 ⟩
      else ⟨ Set penalty parameters 18 ⟩
      else ⟨ Set filename parameters 19 ⟩
      else ⟨ Set single value parameters 20 ⟩
      else throw string("Undefined␣opcode␣in␣input␣script:␣\"") + opcode + '"';
      script ≫ opcode;
    }
  }
```

14. Colours are given as red–green–blue values from 0 to 1. For example,

```
color labels 1 0 0
```

which makes all labels red. The following sub-keywords can be used: "`background`", "`grid`", "`ecliptic`", "`boundaries`", "`highlighted_boundaries`", "`stars`", "`nebulae`", "`labels`", "`text_labels`", "`constellation_lines`", and "`milky_way`". In case of the milky way, the colour denotes the brightest regions. (The darkest have `background` colour.)

⟨ Set color parameters 14 ⟩ ≡

```
if (opcode ≡ "color") {
  string color_name;
  script ≫ color_name;
  if (color_name ≡ "background") script ≫ params.bgcolor;
  else if (color_name ≡ "grid") script ≫ params.gridcolor;
  else if (color_name ≡ "ecliptic") script ≫ params.eclipticcolor;
  else if (color_name ≡ "boundaries") script ≫ params.boundarycolor;
  else if (color_name ≡ "highlighted_boundaries") script ≫ params.hlboundarycolor;
  else if (color_name ≡ "stars") script ≫ params.starcolor;
  else if (color_name ≡ "nebulae") script ≫ params.nebulacolor;
  else if (color_name ≡ "labels") script ≫ params.labelcolor;
  else if (color_name ≡ "text_labels") script ≫ params.textlabelcolor;
  else if (color_name ≡ "constellation_lines") script ≫ params.clinecolor;
  else if (color_name ≡ "milky_way") script ≫ params.milkywaycolor;
  else throw string("Undefined␣\"color\"␣construct""␣in␣input␣script:␣\"") + color_name + '"';
}
```

This code is used in section 13.

15. The following lines can be modified: "`grid`", "`ecliptic`", "`boundaries`", "`highlighted_boundaries`", "`nebulae`", and "`constellation_lines`". The linewidth in centimetres must follow.

⟨ Set line widths 15 ⟩ ≡

```
if (opcode ≡ "line_width") {
  string line_name;
  script ≫ line_name;
  if (line_name ≡ "grid") script ≫ params.linewidth_grid;
  else if (line_name ≡ "ecliptic") script ≫ params.linewidth_ecliptic;
  else if (line_name ≡ "boundaries") script ≫ params.linewidth_boundary;
  else if (line_name ≡ "highlighted_boundaries") script ≫ params.linewidth_hlboundary;
  else if (line_name ≡ "nebulae") script ≫ params.linewidth_nebula;
  else if (line_name ≡ "constellation_lines") script ≫ params.linewidth_cline;
  else
    throw string("Undefined␣\"line_width\"␣construct""␣in␣input␣script:␣\"")+line_name+'"';
}
```

This code is used in section 13.

16. The following lines can be modified: "`grid`", "`ecliptic`", "`boundaries`", "`highlighted_boundaries`", "`nebulae`", and "`constellation_lines`". You can set the respective line style to "`solid`", "`dashed`", and "`dotted`".

```
⟨ Set line styles 16 ⟩ ≡
  if (opcode ≡ "line_style") {
    string line_name;

    script ≫ line_name;
    if (line_name ≡ "grid") script ≫ params.linestyle_grid;
    else if (line_name ≡ "ecliptic") script ≫ params.linestyle_ecliptic;
    else if (line_name ≡ "boundaries") script ≫ params.linestyle_boundary;
    else if (line_name ≡ "highlighted_boundaries") script ≫ params.linestyle_hlboundary;
    else if (line_name ≡ "nebulae") script ≫ params.linestyle_nebula;
    else if (line_name ≡ "constellation_lines") script ≫ params.linestyle_cline;
    else
      throw string("Undefined␣\"line_width\"␣construct""␣in␣input␣script:␣\"") + line_name + '"';
  }
```
This code is used in section 13.

17. There are the following boolean values: "`milky_may`", "`nebulae`", "`colored_stars`", "`grid`", "`ecliptic`", "`boundaries`", "`constellation_lines`", "`labels`", "`eps_output`", and "`pdf_output`". You can switch them "`on`" or "`off`".

```
⟨ Set on/off parameters 17 ⟩ ≡
  if (opcode ≡ "switch") {
    string switch_name;

    script ≫ switch_name;
    if (switch_name ≡ "milky_way") params.milkyway_visible = read_boolean(script);
    else if (switch_name ≡ "nebulae") params.nebulae_visible = read_boolean(script);
    else if (switch_name ≡ "colored_stars") params.colored_stars = read_boolean(script);
    else if (switch_name ≡ "grid") params.show_grid = read_boolean(script);
    else if (switch_name ≡ "ecliptic") params.show_ecliptic = read_boolean(script);
    else if (switch_name ≡ "boundaries") params.show_boundaries = read_boolean(script);
    else if (switch_name ≡ "constellation_lines") params.show_lines = read_boolean(script);
    else if (switch_name ≡ "labels") params.show_labels = read_boolean(script);
    else if (switch_name ≡ "eps_output") params.create_eps = read_boolean(script);
    else if (switch_name ≡ "pdf_output") params.create_pdf = read_boolean(script);
    else
      throw string("Undefined␣\"switch\"␣construct""␣in␣input␣script:␣\"") + switch_name + '"';
  }
```
This code is used in section 13.

18. In order to avoid overlaps, PP3 uses a simple penalty algorithm. The standard value for all penalty values is 1000. The meanings of "`stars`", "`labels`", "`nebulae`", "`boundaries`", and "`constellation_lines`" is pretty easy to explain: They come into play when the current label (that is to be positioned) overlaps with the respective object. For example, if you want overlaps with constellation lines to be less probable, you can say

```
penalties constellation_lines 2000
```

There is another concept of importance here: The rim. A rim is a rectangular margin around every label with a width of *skip*. Overlaps in the rim are counted, too, however normally they don't hurt that much. Normally they hurt half as much as the label area (*core*) itself, but this can be changed with "`rim`". With

```
penalties rim 0
```

the rim loses its complete significance. But notice that for each rim penalty a core penalty is added, too, so that the rim can never be more significant than the core.

Within the rim, "`boundaries_rim`" and "`constellation_lines_rim`" are used instead of the normal ones. This is because lines are not so bad in the rim as other stars or nebulae would be, because other stars in the vicinity of a label may cause confusion, lines not.

The third thing about penalties is the maximal penalty of a label. If the penalties of a label exceed this value, the label is supressed. You may overrule this behaviour with an explicit repositioning of the label. You can adjust this maximal badness of the label with "`threshold`". With

```
penalties threshold 10000
```

probably all labels are printed.

```
⟨ Set penalty parameters 18 ⟩ ≡
  if (opcode ≡ "penalties") {
    string penalty_name;
    double value;
    script ≫ penalty_name ≫ value;
    value /= 1000.0;
    if (penalty_name ≡ "stars") params.penalties_star = value;
    else if (penalty_name ≡ "labels") params.penalties_label = value;
    else if (penalty_name ≡ "nebulae") params.penalties_nebula = value;
    else if (penalty_name ≡ "boundaries") params.penalties_boundary = value;
    else if (penalty_name ≡ "boundaries_rim") params.penalties_boundary_rim = value;
    else if (penalty_name ≡ "constellation_lines") params.penalties_cline = value;
    else if (penalty_name ≡ "constellation_lines_rim") params.penalties_cline_rim = value;
    else if (penalty_name ≡ "threshold") params.penalties_threshold = value;
    else if (penalty_name ≡ "rim") params.penalties_rim = value;
    else
      throw string("Undefined␣\"penalties\"␣construct""␣in␣input␣script:␣\"") + penalty_name +
          '"';
  }
```

This code is used in section 13.

19. The most important filename is "`output`". By default it's unset so that the output is sent to standard output. With

```
filename output orion.tex
```

the output is written to `orion.tex`. Most of the other filenames denote the data files. Their file format is described at the functions that read them. Their names are: "`stars`", "`nebulae`", "`label_dimensions`", "`constellation_lines`", "`boundaries`", and "`milky_way`".

"`latex_preamble`" is a file with a LaTeX excerpt with a preamble fragment for the LaTeX output. See ⟨ *create_preamble*( ) for writing the LaTeX preamble 120 ⟩.

"`include`" denotes a file that is included at the current position as an insert script file. This command particularly makes sense at the very beginning of the input script because then you can overwrite the values locally. Note that you can only include zero or one file, and included script files must not contain further `include`s. Apart from that included scripts have the same structure as usual scripts. (This is also true for a possible '`objects_and_labels`' part.)

The meaning of this is of course to write a master script with global settings (e. g. colour, line style, data file names etc.), and to overwrite them for certain regions of the sky, typically stellar constellations.

```
⟨ Set filename parameters 19 ⟩ ≡
  if (opcode ≡ "filename") {
    string object_name;
    script ≫ object_name;
    if (object_name ≡ "output") params.filename_output = read_string(script);
    else if (object_name ≡ "stars") params.filename_stars = read_string(script);
    else if (object_name ≡ "nebulae") params.filename_nebulae = read_string(script);
    else if (object_name ≡ "label_dimensions") params.filename_dimensions = read_string(script);
    else if (object_name ≡ "constellation_lines") params.filename_lines = read_string(script);
    else if (object_name ≡ "boundaries") params.filename_boundaries = read_string(script);
    else if (object_name ≡ "milky_way") params.filename_milkyway = read_string(script);
    else if (object_name ≡ "latex_preamble") params.filename_preamble = read_string(script);
    else if (object_name ≡ "include") {
      if (¬params.filename_include.empty())
        throw string("Nesting␣depth␣of␣include␣files␣greater""␣than␣one");
      params.filename_include = read_string(script);
      ifstream included_script(params.filename_include.c_str());
      read_parameters_from_script(included_script);
    }
    else
      throw string("Undefined␣\"filename\"␣construct""␣in␣input␣script:␣\"")+object_name+'"';
  }
```

This code is used in section 13.

20. Most of these values are numeric, only `constellation` is a string, namely a three-letter all-uppercase astronomic abbreviation of the constellation to be highlighted. It's default is "`ORI`" but you may set it to the empty string with

```
set constellation ""
```

so no constellation gets highlighted. At the moment highlighting means that the boundaries have a brighter colour than normal.

"`center_rectascension`" and "`center_declination`" are the celestial coordinates of the view frame centre. "`box_width`" and "`box_height`" are the dimensions of the view frame in centimetres. "`grad_per_cm`" is the magnification (scale). "`star_scaling`" denotes a radial scaling of the star disks. "`fontsize`" is the global LaTeX font size (in points). It must be 10, 11, or 12. Default is 10.

Many parameters deal with the graphical representation of stars and nebulae: "`shortest_constellation_line`" is the shortest length for a constellation line that is supposed to be drawn. "`label_skip`" is the distance between label and celestial object. "`minimal_nebula_radius`" is the radius under which a nebula is drawn as a mere circle of *that* radius. "`minimal_star_radius`" is the radius of the smallest stellar dots of the graphics. All these parameters are measured in centimetres.

But there are also some magnitudes: The faintest stellar clusters that are drawn by default are of the "`faintest_cluster_magnitude`", all other nebulae drawn by default of the "`faintest_diffuse_nebula_magnitude`". Stars brighter than "`faintest_star_magnitude`" are drawn at all, if they are even brighter than "`faintest_star_with_label_magnitude`" they get a label. Stars brighter than "`faintest_star_disk_magnitude`" are not just mere dots in the background, but get a radius according to their brightness.

Many of these parameters trigger the default behaviour that you can overrule by commands in the second part of the input script.

```
⟨ Set single value parameters 20 ⟩ ≡
  if (opcode ≡ "set") {
    string param_name;
    script ≫ param_name;
    if (param_name ≡ "center_rectascension") script ≫ params.center_rectascension;
    else if (param_name ≡ "center_declination") script ≫ params.center_declination;
    else if (param_name ≡ "box_width") script ≫ params.view_frame_width;
    else if (param_name ≡ "box_height") script ≫ params.view_frame_height;
    else if (param_name ≡ "grad_per_cm") script ≫ params.grad_per_cm;
    else if (param_name ≡ "constellation") params.constellation = read_string(script);
    else if (param_name ≡ "shortest_constellation_line") script ≫ params.shortest_constellation_line;
    else if (param_name ≡ "label_skip") script ≫ params.label_skip;
    else if (param_name ≡ "minimal_nebula_radius") script ≫ params.minimal_nebula_radius;
    else if (param_name ≡ "faintest_cluster_magnitude") script ≫ params.faintest_cluster_magnitude;
    else if (param_name ≡ "faintest_diffuse_nebula_magnitude")
      script ≫ params.faintest_diffuse_nebula_magnitude;
    else if (param_name ≡ "faintest_star_magnitude") script ≫ params.faintest_star_magnitude;
    else if (param_name ≡ "minimal_star_radius") script ≫ params.minimal_star_radius;
    else if (param_name ≡ "faintest_star_disk_magnitude")
      script ≫ params.faintest_star_disk_magnitude;
    else if (param_name ≡ "faintest_star_with_label_magnitude")
      script ≫ params.faintest_star_with_label_magnitude;
    else if (param_name ≡ "star_scaling") script ≫ params.star_scaling;
    else if (param_name ≡ "fontsize") script ≫ params.font_size;
    else throw string("Undefined␣\"set\"␣construct␣in␣input␣script:␣\"") + param_name + '"';
  }
```

This code is used in section 13.

21. Part II: Change printed objects and labels. Here I read and interpret the second part of the input script, *after* the `"objects_and_labels"`. This part doesn't need to be available, and both parts may be empty.

First I define a type that is often used in the following routines for a mapping from a catalogue number on the index in PP3's internal *vectors*. This makes access a lot faster.

⟨ Routines for reading the input script 11 ⟩ +≡

```
  typedef map⟨int, int⟩ index_list;
```

22. Here I create the data structures that make the above mentioned mapping possible. FixMe: They should be defined globally, so that the constellation lines construction can profit by it, too. And then they needn't be given in the parameter lists of the routines.

This mapping is not vital for the program, but the alternative would be to look through the whole of *nebulae* or *stars* to find a star with a certain NGC or HD number. This is probably way to inefficient.

⟨ Create mapping structures for direct catalogue access 22 ⟩ ≡

```
  const int max_ngc = 7840, max_ic = 5386, max_messier = 110;
  index_list ngc, ic, messier;
  for (int i = 0; i < nebulae.size(); i++) {
    if (nebulae[i].ngc > 0 ∧ nebulae[i].ngc ≤ max_ngc) ngc[nebulae[i].ngc] = i;
    if (nebulae[i].ic > 0 ∧ nebulae[i].ic ≤ max_ic) ic[nebulae[i].ic] = i;
    if (nebulae[i].messier > 0 ∧ nebulae[i].messier ≤ max_messier) messier[nebulae[i].messier] = i;
  }
  index_list henry_draper;
  map⟨string, index_list⟩ flamsteed;
  for (int i = 0; i < stars.size(); i++) {
    if (stars[i].hd > 0) henry_draper[stars[i].hd] = i;
    if (stars[i].flamsteed > 0) flamsteed[stars[i].constellation][stars[i].flamsteed] = i;
  }
```

This code is used in section 26.

23. This is a general warning producing routine. Such tests are vital, especially if the nebulae file has been deleted in order to save disk space. I want it to be a warning because it would be very annowing to delete all such opcodes from an input script only because one wants to dispense with e. g. nebulae.

I need three incarnations of this function: One for nebulae, one for Flamsteed number, and one for Henry Draper catalogue numbers.

```
⟨ Routines for reading the input script 11 ⟩ +≡
  bool assure_valid_catalogue_index(const int index, const index_list &catalogue, const string catalogue_name)
  {
    if (catalogue.find(index) ≡ catalogue.end()) {
      cerr ≪ "pp3:␣Warning:␣Invalid␣" ≪ catalogue_name ≪ "␣index:␣" ≪ index ≪ ".\n";
      return false;
    }
    else return true;
  }
  bool assure_valid_catalogue_index(const int index, const string constellation, map⟨string, index_list⟩
          &flamsteed)
  {
    bool found = true;
    if (flamsteed.find(constellation) ≡ flamsteed.end()) found = false;
    else if (flamsteed[constellation].find(index) ≡ flamsteed[constellation].end()) found = false;
    if (¬found) cerr ≪ "pp3:␣Warning:␣Invalid␣Flamsteed␣number:␣" ≪ index ≪ ".\n";
    return found;
  }
  bool assure_valid_catalogue_index(const int index, const index_list &henry_draper)
  {
    if (henry_draper.find(index) ≡ henry_draper.end()) {
      cerr ≪ "pp3:␣Warning:␣Invalid␣HD␣number:␣" ≪ index ≪ ".\n";
      return false;
    }
    else return true;
  }
```

24. In this routine I scan a list of stellar objects, given by a token pair of catalogue name and catalogue index. Such lists are used after some top-level commands below. A mandatory '`;`' ends such a list. Five catalogues are supported: NGC, IC, Messier, Henry-Draper (HD), and Flamsteed numbers (in the form "*Constellation␣Flamsteed number*"). You may use the program 'Celestia' to get the HD numbers for the stars.

```
⟨ Routines for reading the input script 11 ⟩ +≡
  void search_objects(istream &script, index_list &ngc, index_list &ic, index_list &messier, index_list
          &henry_draper, map⟨string, index_list⟩ &flamsteed, vector⟨int⟩ &found_stars, vector⟨int⟩
          &found_nebulae)
  {
    found_stars.clear();
    found_nebulae.clear();
    string catalogue_name;
    int catalogue_index;
    script ≫ catalogue_name;
    while (script ∧ catalogue_name ≠ ";") {
      script ≫ catalogue_index;
      if (catalogue_index ≤ 0) {
        stringstream error_message;
        error_message ≪ "Invalid␣index:␣" ≪ catalogue_index;
        throw error_message.str();
      }
      if (¬script) throw string("Unexpected␣end␣of␣input␣script");
      if (params.nebulae_visible ∨ (catalogue_name ≠ "NGC" ∧ catalogue_name ≠ "IC" ∧ catalogue_name ≠ "M"))
        {
        if (catalogue_name ≡ "NGC") {
          if (assure_valid_catalogue_index(catalogue_index, ngc, "NGC"))
            found_nebulae.push_back(ngc[catalogue_index]);
        }
        else if (catalogue_name ≡ "IC") {
          if (assure_valid_catalogue_index(catalogue_index, ic, "IC"))
            found_nebulae.push_back(ic[catalogue_index]);
        }
        else if (catalogue_name ≡ "M") {
          if (assure_valid_catalogue_index(catalogue_index, messier, "M"))
            found_nebulae.push_back(messier[catalogue_index]);
        }
        else if (catalogue_name ≡ "HD") {
          if (assure_valid_catalogue_index(catalogue_index, henry_draper))
            found_stars.push_back(henry_draper[catalogue_index]);
        }
        else {
          if (assure_valid_catalogue_index(catalogue_index, catalogue_name, flamsteed))
            found_stars.push_back(flamsteed[catalogue_name][catalogue_index]);
        }
      }
      script ≫ catalogue_name;
    }
  }
```

25. This routine essentially does the same as the prevous one, however only for *one* celestial object. This is used for commands that don't take an object list but only one object.

```
⟨ Routines for reading the input script 11 ⟩ +≡
  view_data ∗ identify_object(istream &script, index_list &ngc, index_list &ic, index_list &messier, index_list
        &henry_draper, map⟨string, index_list⟩ &flamsteed, stars_list &stars, nebulae_list & nebulae)
  {
    string catalogue_name;
    int catalogue_index;
    script ≫ catalogue_name ≫ catalogue_index;
    if (¬script) throw string("Unexpected␣end␣of␣input␣script");
    if (catalogue_name ≡ "NGC") {
      if (assure_valid_catalogue_index(catalogue_index, ngc, "NGC")) return &nebulae[ngc[catalogue_index]];
    }
    else if (catalogue_name ≡ "IC") {
      if (assure_valid_catalogue_index(catalogue_index, ic, "IC")) return &nebulae[ic[catalogue_index]];
    }
    else if (catalogue_name ≡ "M") {
      if (assure_valid_catalogue_index(catalogue_index, messier, "M"))
        return &nebulae[messier[catalogue_index]];
    }
    else if (catalogue_name ≡ "HD") {
      if (assure_valid_catalogue_index(catalogue_index, henry_draper))
        return &stars[henry_draper[catalogue_index]];
    }
    else {
      if (assure_valid_catalogue_index(catalogue_index, catalogue_name, flamsteed))
        return &stars[flamsteed[catalogue_name][catalogue_index]];
    }
    return 0;
  }
```

26. Here now the main routine for the second part of the input script. The top-level commands in this section are: "`reposition`", "`delete_labels`", "`add_labels`", "`delete`", "`add`", "`set_text_label`", and "`text`".

```
⟨ Routines for reading the input script 11 ⟩ +≡
  void read_objects_and_labels(istream &script, const dimensions_list &dimensions, objects_list & objects,
          stars_list &stars, nebulae_list & nebulae, texts_list & texts, flexes_list & flexes, const transformation
          &mytransform, bool included = false)
  {
    if (¬params.filename_include.empty( ) ∧ ¬included) {
      ifstream included_file(params.filename_include.c_str( ));
      ⟨ Skip everything till "objects_and_labels" 27 ⟩
      read_objects_and_labels(included_file, dimensions, objects, stars, nebulae, texts, flexes, mytransform, true);
    }
    string opcode;
    script ≫ opcode;
    if (¬script) return;
    ⟨ Create mapping structures for direct catalogue access 22 ⟩
    while (script) {
      if (opcode[0] ≡ '#') {      /* skip comment line */
        string rest_of_line;
        getline(script, rest_of_line);
      }
      else ⟨ Label repositioning 28 ⟩
      else ⟨ Change label text 30 ⟩
      else ⟨ Text labels 35 ⟩
      else {      /* command with objects list */
        vector⟨int⟩ found_stars, found_nebulae;
        ⟨ Label deletion 31 ⟩
        else ⟨ Label activation 32 ⟩
        else ⟨ Celestial object deletion 34 ⟩
        else ⟨ Celestial object activation 33 ⟩
        else throw string("Undefined␣opcode␣in␣input␣script:␣\"") + opcode + '"';
      }
      script ≫ opcode;
    }
  }
```

27. The following algorithm is so bad that it must be considered a bug. It simply searches for the string `"objects_and_labels"` in the *included_file*, but that may occur within a string or wherever. The file should be properly scanned, but I'm too lazy for that. A case where this bug becomes visible should be very rare anyway.

FixMe: Fix this bug, or at least for strings. Maybe the format of input scripts must be changed significantly for this.

```
⟨ Skip everything till "objects_and_labels" 27 ⟩ ≡
  string token;
  included_file ≫ token;
  while (included_file) {
    if (token[0] ≡ '#') getline(included_file, token);
    else if (token ≡ "objects_and_labels") break;
    included_file ≫ token;
  }
```

This code is used in section 26.

28. Sometimes labels have an unfortunate position. But you may say e. g.

```
reposition M 42 E ;
```

to position the label for the Orion Nebula to the right of it. (Abbreviations are taken from the wind rose.) You may use this command to force a certain label to be drawn although PP3 has decided that there is no space for it and didn't print it in the first place.

If you write "`E_`" or "`W_`" the label text is not vertically centred but aligned with the celestrial coordinates at the *baseline* of the label text.

At the moment, this command takes exactly one argument. However the closing semicolon is necessary.

```
⟨ Label repositioning 28 ⟩ ≡
  if (opcode ≡ "reposition") {
    string position, semicolon;
    view_data ∗ current_object = identify_object(script, ngc, ic, messier, henry_draper, flamsteed, stars, nebulae);
    int angle;
    bool on_baseline = false;
    script ≫ position ≫ semicolon;
    if (semicolon ≢ ";") throw string("Expected␣\";\"␣after␣\"reposition\"␣command");
    ⟨ Map a wind rose position to an angle and determine on_baseline 29 ⟩
    if (current_object) {
      current_object→label_angle = angle;
      current_object→with_label = visible;
      current_object→on_baseline = on_baseline;
      current_object→label_arranged = true;
    }
  }
```

This code is used in section 26.

29. ⟨ Map a wind rose *position* to an *angle* and determine *on_baseline* 29 ⟩ ≡

```
if (position ≡ "E" ∨ position ≡ "E_") angle = 0;
else if (position ≡ "NE") angle = 1;
else if (position ≡ "N") angle = 2;
else if (position ≡ "NW") angle = 3;
else if (position ≡ "W" ∨ position ≡ "W_") angle = 4;
else if (position ≡ "SW") angle = 5;
else if (position ≡ "S") angle = 6;
else if (position ≡ "SE") angle = 7;
else throw string("Undefined␣position␣angle:␣") + position;
on_baseline = position[position.length() − 1] ≡ '_';
```

This code is used in sections 28 and 35.

30. PP3 reads default labels from the stars and nebulae data files. But you may say e. g.

```
set_label_text ORI 19 Rigel
```

to rename "α" (Orionis) to "Rigel".

⟨ Change label text 30 ⟩ ≡

```
if (opcode ≡ "set_label_text") {
   view_data ∗ current_object = identify_object(script, ngc, ic, messier, henry_draper, flamsteed, stars, nebulae);
   if (current_object) current_object→label = read_string(script);
}
```

This code is used in section 26.

31. With e. g.

```
delete_labels M 35 M 42 ;
```

you delete the labels (not the nebulae themselves!) of M 35 and M 42.

⟨ Label deletion 31 ⟩ ≡

```
if (opcode ≡ "delete_labels") {
   search_objects(script, ngc, ic, messier, henry_draper, flamsteed, found_stars, found_nebulae);
   for (int i = 0; i < found_stars.size(); i++) {
      stars[found_stars[i]].with_label = hidden;
      stars[found_stars[i]].label_arranged = true;
   }
   for (int i = 0; i < found_nebulae.size(); i++) {
      nebulae[found_nebulae[i]].with_label = hidden;
      nebulae[found_nebulae[i]].label_arranged = false;
   }
}
```

This code is used in section 26.

32. The counterpart of `delete_labels`. It makes sense first and foremost for stars. (Unfortunately this means that you have to use extensively the Henry Draper Catalogue.)

```
⟨ Label activation 32 ⟩ ≡
  if (opcode ≡ "add_labels") {
    search_objects(script, ngc, ic, messier, henry_draper, flamsteed, found_stars, found_nebulae);
    for (int i = 0; i < found_stars.size(); i++) stars[found_stars[i]].with_label = visible;
    for (int i = 0; i < found_nebulae.size(); i++) nebulae[found_nebulae[i]].with_label = visible;
  }
```

This code is used in section 26.

33. This adds objects (mostly nebulae) the the field. Notice that this object is then printed even if it lies outside the view frame (it may be clipped though).

```
⟨ Celestial object activation 33 ⟩ ≡
  if (opcode ≡ "add") {
    search_objects(script, ngc, ic, messier, henry_draper, flamsteed, found_stars, found_nebulae);
    for (int i = 0; i < found_stars.size(); i++) stars[found_stars[i]].in_view = visible;
    for (int i = 0; i < found_nebulae.size(); i++) nebulae[found_nebulae[i]].in_view = visible;
  }
```

This code is cited in section 34.

This code is used in section 26.

34. The opposite of ⟨ Celestial object activation 33 ⟩.

```
⟨ Celestial object deletion 34 ⟩ ≡
  if (opcode ≡ "delete") {
    search_objects(script, ngc, ic, messier, henry_draper, flamsteed, found_stars, found_nebulae);
    for (int i = 0; i < found_stars.size(); i++) stars[found_stars[i]].in_view = hidden;
    for (int i = 0; i < found_nebulae.size(); i++) nebulae[found_nebulae[i]].in_view = hidden;
  }
```

This code is used in section 26.

35. This is the only way to add a text label. The parameters are the text itself, rectascension, declination, RGB colour, and the relative position (uppercase wind rose), followed by a semicolon. For example,

```
text Leo at 11 10 color 1 0 0 towards S ;
```

puts a red "Leo" centered below the point (11 h, +10°) in the Lion. You may leave out the "`color`" and/or the "`towards`" option. The default colour is the last given colour in a previous `text` command, or if this doesn't exist the current text label colour. The default value of "`towards`" is `"NE"`.

The contents of a text label is eventually in an `\hbox`, so you can use that fact. You can also use all PSTricks commands. For example, with

```
text "\\psdots[dotstyle=+,dotangle=45](0,0)\\scriptsize\\ N pole of ecliptic"
     at 18 66.56 towards E_ ;
```

the × marks the northern ecliptical pole, together with a small label text. If such things occur frequently, define it as a LaTeX macro.

If you write "`E_`" or "`W_`" the label text is not vertically centred but aligned with the celestrial coordinates at the *baseline* of the label text.

A "declination flex" is a special text label that stands on a circle of equal declination which can look very nice. You get it with the option "`along declination`" in the parameter list of the text label.

If you use the `tics` option the label is printed along a rectascension or declination circle multiple times, making it possible to print tic marks. For example,

```
text "$#3$" at 0 20 along declination tics rectascension 1 towards N ;
text "$#5$" at 11 0 along declination tics declination 10 towards S ;
```

creates automatically generated labels for the 20° declination circle (every whole hour), and for the 11^h rectascension line (every 10 declination degrees). See the explanation for ⟨ Generate *contents* from current coordinates 37 ⟩ for further details.

⟨ Text labels 35 ⟩ ≡

```
if (opcode ≡ "text") {
  string token, contents, position("NE");
  double rectascension, declination;

  contents = read_string(script);
  script ≫ token;
  if (token ≠ "at") throw string("\"at\"␣in␣\"text\"␣command␣expected");
  script ≫ rectascension ≫ declination;
  script ≫ token;

  int angle = 1;     /* "NE" */
  bool on_baseline = false;
  enum {
    kind_text_label, kind_flex_declination
  } label_kind = kind_text_label;
  double step_rectascension = −1.0;     /* −1 means that no maks are produced. */
  double step_declination = −1.0;

  while (script ∧ token ≠ ";") {
    if (token ≡ "color") script ≫ params.textlabelcolor;
    else if (token ≡ "towards") {
      script ≫ position;
      ⟨ Map a wind rose position to an angle and determine on_baseline 29 ⟩
      }
      else if ($token \equiv$ "along") {
        $script \gg token$;
        if ($token \equiv$ "declination") {
          $label\_kind = kind\_flex\_declination$;
        }
        else throw string("Invalid␣\"along\"␣option");
      }
      else if ($token \equiv$ "tics") {
        $script \gg token$;
        if ($token \equiv$ "rectascension") {
          $script \gg step\_rectascension$;
          if ($step\_rectascension \leq 0.0$) throw string("Invalid␣\"tics\"␣interval");
          $step\_declination = -1.0$;
        }
        else if ($token \equiv$ "declination") {
          $script \gg step\_declination$;
          if ($step\_declination \leq 0.0$) throw string("Invalid␣\"tics\"␣interval");
          $step\_rectascension = -1.0$;
        }
        else throw string("Invalid␣\"tics\"␣option");
      }
      else throw string("Invalid␣\"text\"␣option");
      $script \gg token$;
    }
    if ($\neg script$) throw string("Unexpected␣end␣of␣script␣while␣scanning␣\"text\"");
    if ($step\_rectascension \leq 0.0 \wedge step\_declination \leq 0.0$) {
      ⟨ Insert text label into label container structure 36 ⟩
    }
    if ($step\_rectascension > 0.0$) {
      string $user\_pattern(contents)$;
      double $start = rectascension - floor(rectascension / step\_rectascension) * step\_rectascension$;
      for ($rectascension = start$; $rectascension < 24.0$; $rectascension$ += $step\_rectascension$) {
        $cerr \ll rectascension \ll$ '␣';
        ⟨ Generate contents from current coordinates 37 ⟩
        ⟨ Insert text label into label container structure 36 ⟩
      }
    }
    else if ($step\_declination > 0.0$) {
      string $user\_pattern(contents)$;
      double $start = declination - floor((declination + 90.0)/step\_declination) * step\_declination$;
      for ($declination = start$; $declination \leq 90.0$; $declination$ += $step\_declination$) {
        $cerr \ll declination \ll$ '␣';
        ⟨ Generate contents from current coordinates 37 ⟩
        ⟨ Insert text label into label container structure 36 ⟩
      }
    }
  }
```

This code is used in section 26.

36. FixMe: Eventually this section must be a clean function call. I insert a text label or a flex into the correct data structure.

```
⟨ Insert text label into label container structure 36 ⟩ ≡
  switch (label_kind) {
  case kind_text_label:
    texts.push_back(text(contents, rectascension, declination, params.textlabelcolor, angle, on_baseline));
    break;
  case kind_flex_declination: flexes.push_back(new declination_flex(contents, rectascension, declination,
          params.textlabelcolor, angle, on_baseline));
    break;
  }
```

This code is used in section 35.

37. FixMe: Eventually this section must be a clean function call. Here I create the LaTeX macro for the tics marks. It is called `\coordinates` and has nine parameters, however not all are used so far:

#1 Rectascension in hours.
#2 Declination in degrees.
#3 Rectascension in integer hours (truncated, not rounded).
#4 Rectascension fraction of hour in minutes (truncated, not rounded).
#5 Declination in rounded integer degrees.

All decimal points are replaced by "`{\DP}`" commands.

```
⟨ Generate contents from current coordinates 37 ⟩ ≡
  {
    stringstream coordinates;
    coordinates.setf(ios::fixed);
    coordinates ≪ "\\def\\coordinates#1#2#3#4#5#6#7#8#9{\\TicMark{" ≪ user_pattern ≪ "}}";
    coordinates ≪ "\\coordinates{" ≪ rectascension ≪ "}{" ≪ declination ≪ "}{";
    coordinates ≪ int(floor(rectascension)) ≪ "}{" ≪ int(floor((rectascension − floor(rectascension)) ∗ 60.0)) ≪
        "}{";
    if (int(declination) > 0.0) coordinates ≪ '+';
    coordinates ≪ int(declination) ≪ "}{}{}{}{}";
    contents = coordinates.str();
    while (contents.find(".") ≠ string::npos) contents.replace(contents.find("."), 1, "{\\DP}");
  }
```

This code is cited in section 35.

This code is used in section 35.

38. Data structures and file formats. First I describe the data structures that directly contain celestial objects such as stars and nebulae. This is a little bit unfortunate for the program itself, because actually other things should be defined first (e. g. **dimension**). However I think that the program can be digested more easily this way.

All data structures are **struct**s and no **class**es. The reason is that they are all very simple and all object oriented security measures are only hindering.

For each data type called **classname** I define a container called **classnames_list** that is a **vector**. For almost each data type I define a routine that can read it from a file in the vector. There I also decribe the respective file format.

39. View data: Positioning and labels. This first **struct** can be used as an ancestor in the various celestial objects data structurs (e. g. stars) for containing all view frame dependent information. Most importantly it contains the label of a celestial object, its position relatively to the object, and its size.

in_view is *visible* if the object is actually displayed on screen. x and y contain the screen coordinates in centimetres. *radius* is the radial size of the object in centimetres. *skip* is given in centimetres, too. It denotes the space between the outer boundary of the object (enclosed by *radius*) and the label.

with_label is *visible* if the object has a label, with *label_width*, *label_height*, *label_depth* (estimated in centimetres) and *label_angle*. *Please note the all heights in* PP3 *are* total *heights, thus the same as TEX's height + depth!*

on_baseline is true, if the *baseline* of the label text is supposed to be vertically aligned with the celestial position. (This works not for all *label_angle*s.)

label_arranged is *true* if the best place for the label has been found already. Only then the label should be really printed, but the real use of *label_arranged* is that it avoids the label to be arranged twice. *lable_angle* can only have eight possible values: 0: 0°, 1: 45°, 2: 90°, 3: 135°, 4: 180°, 5: 225°, 6: 270°, and 7: 315°.

scope( ) returns the maximal scope of the object. It is used in ⟨ Find objects in vicinity of *objects*[*i*] 83 ⟩ to find all objects in the vicinity of a given on-object.

```
typedef enum {
  hidden, visible, undecided
} visibility;
struct view_data {
  visibility in_view;
  double x, y;
  double radius, skip;
  visibility with_label;
  bool label_arranged;
  string label;
  double label_width, label_height, label_depth;
  bool on_baseline;
  color label_color;
  int label_angle;
  view_data( )
  : in_view(undecided), x(DBL_MAX), y(DBL_MAX), radius(0.0), skip(params.label_skip), with_label(undecided),
      label_arranged(false), label( ), label_width(0.0), label_height(0.0), label_depth(0.0), on_baseline(false),
      label_color(params.labelcolor), label_angle(0) { }
  void get_label_boundaries(double &left, double &right, double &top, double &bottom) const;
  double scope( ) const
  {
    return radius + skip + fmax(label_width, label_height) + 2.0 * skip;
  }
  bool has_valid_coordinates( ) const
  {
    return x ≠ DBL_MAX ∧ y ≠ DBL_MAX;
  }
  virtual double penalties_with(const double left, const double right, const double top, const double
      bottom, bool core = true) const;
};
```

40. This is the only structure that is not put into a container directly, but via references. The reason is the virtual routine *penalties_with*( ); I want to use polymorphy.

```
typedef vector⟨view_data *⟩ objects_list;
```

41. Each label is stored in **view_data** by its dimensions, its *skip*, the *radius* of the object itself, and the *label_angle*. While these quantities are very convenient to use for the positioning of the label, they are pretty unfortunate if you want to know which coordinates are occupied by the label in order to find out possible overlaps.

Therefore I calculate here the rectangular boundaries of a label. They are stored in *left*, *right*, *top*, and *bottom* in screen centimetres.

```
void view_data::get_label_boundaries(double &left, double &right, double &top, double &bottom) const
{
  const double origin_x = x + (radius + skip) * cos(M_PI_4 * double(label_angle));
  const double origin_y = y + (radius + skip) * sin(M_PI_4 * double(label_angle));
  switch (label_angle) {
  case 0: case 1: case 7: left = origin_x;
    break;
  case 2: case 6: left = origin_x − label_width/2.0;
    break;
  case 3: case 4: case 5: left = origin_x − label_width;
    break;
  }
  right = left + label_width;
  switch (label_angle) {
  case 0: case 4: bottom = origin_y − (on_baseline ? 0.0 : label_height/2.0);
    break;
  case 1: case 2: case 3: bottom = origin_y;
    break;
  case 5: case 6: case 7: bottom = origin_y − label_height;
    break;
  }
  if (on_baseline) bottom −= label_depth;
  top = bottom + label_height;
}
```

42. Every object of type **view_data** (or a descendant of it) must be able to calculate the overlap of itself with a rectangle given by *left*, *right*, *top*, and *bottom*. It must then create a penalty value out of it. Normally this is just the overlap itself in square centimetres, like here.

```
double view_data::penalties_with(const double left, const double right, const double top, const double
        bottom, bool core) const
{
  if (with_label ≡ visible ∧ label_arranged) {
    double left2, right2, top2, bottom2;
    get_label_boundaries(left2, right2, top2, bottom2);
    const double overlap_left = fmax(left, left2);
    const double overlap_right = fmin(right, right2);
    const double overlap_top = fmin(top, top2);
    const double overlap_bottom = fmax(bottom, bottom2);
    const double overlap_x = fdim(overlap_right, overlap_left);
    const double overlap_y = fdim(overlap_top, overlap_bottom);
    return overlap_x * overlap_y * params.penalties_label;
  }
  else return 0.0;
}
```

43. Stars. The actual star data is – like all other user defined data structure in this program – organised as a **struct** because it's too simple for a **class**. *hd* is the Henry Draper Catalog number, *bs* is the Bright Star Catalog number. *rectascension* is given in hours, *declination* in degrees. *b_v* is the B−V brightness in magnitudes (equals $m_{\text{phot}} - m_{\text{vis}}$) and thus contains colour. *name* either contains the Bayer name (only the Greek letter and maybe an exponent number), or, if this doesn't exist, the Flamsteed number as a string. *spectral_class* is the complete spectral class, starting with "F6" or whatever.

```
struct star : public view_data {
  int hd, bs;
  double rectascension, declination;
  double magnitude;
  double b_v;
  int flamsteed;
  string name;
  string constellation;
  string spectral_class;
  star()
  : hd(0), bs(0), rectascension(0.0), declination(0.0), magnitude(0.0), b_v(0.0), flamsteed(0), name(""),
      spectral_class(""), view_data() { }
  virtual double penalties_with(const double left, const double right, const double top, const double
      bottom, bool core = true) const;
};
```

44. In order to find the penalties with a (labelled) star, I first calculate them for the label itself, which may be 0.0, in particular if *label_arranged* is still *false*. Then I determine the rectangular overlap, just like in **view_data**::*penalties_with*( ). This is unfortunate, because stars are circles and not rectangles. FixMe: This should be done justice to.

```
double star::penalties_with(const double left, const double right, const double top, const double
        bottom, bool core) const
{
  double penalties = view_data::penalties_with(left, right, top, bottom, core);
  const double left2 = x − radius, right2 = x + radius, top2 = y + radius, bottom2 = y − radius;
  const double overlap_left = fmax(left, left2);
  const double overlap_right = fmin(right, right2);
  const double overlap_top = fmin(top, top2);
  const double overlap_bottom = fmax(bottom, bottom2);
  const double overlap_x = fdim(overlap_right, overlap_left);
  const double overlap_y = fdim(overlap_top, overlap_bottom);
  penalties += overlap_x ∗ overlap_y ∗ params.penalties_star;
  return penalties;
}
typedef vector⟨star⟩ stars_list;
```

45. I want to be able to read the star data records from a text format file. The format of the input is a text stream with the following format. Each star entry consists of four lines:

1. A row with seven fields separated by whitespace:
   – Henry Draper Catalogue number (**int**, '0' if unknown),
   – BSC number (**int**, '0' if unknown),
   – rectascension in hours (**double**),
   – declination in degrees (**double**),
   – visual brightness in magnitudes (**double**),
   – B−V brightness in magnitudes (**double**, '99.0' if unknown),
   – Flamsteed number (**int**, '0' if unknown).
2. The label (astronomical name) for the star, as a LaTeX-ready string, e. g. "`$\alpha$`", "`$\phi^{2}$`", or simply "`$23$`". May be the empty string.
3. The astronomical abbreviation of the constellation. It must be all uppercase.
4. The spectral class. It must start with the spectral class letter, followed by the fraction digit, followed by the luminosity class as a Roman number, e. g. "`F5III`". Anything may follow as in "`K2-IIICa-1`", however the mandatory parts must not contain any whitespace.

```
istream &operator≫(istream &in, star &s)
{
  in ≫ s.hd ≫ s.bs ≫ s.rectascension ≫ s.declination ≫ s.magnitude ≫ s.b_v ≫ s.flamsteed;
  char ch;
  do in.get(ch); while (in ∧ ch ≠ '\n');
  getline(in, s.name);
  getline(in, s.constellation);
  getline(in, s.spectral_class);
  return in;
}
```

46. Here I read a set of stars from a file with the name *filename*. The file *stars_file* is a text file that consists solely of a list of **star**s.

```
void read_stars(stars_list &stars)
{
  ifstream stars_file(params.filename_stars.c_str());
  if (¬stars_file) throw string("No␣stars␣file␣found:␣" + params.filename_stars);
  star current_star;
  stars_file ≫ current_star;
  while (stars_file) {
    current_star.label = string("\\Starname{") + current_star.name + '}';
    stars.push_back(current_star);
    stars_file ≫ current_star;
  }
}
```

47. Nebulae. This is also used for stellar clusters of course. In the whole program "nebula" denotes nebulae, galaxies, clusters, and other non-stellar objects.

```
typedef enum {
  unknown, galaxy, emission, reflection, open_cluster, globular_cluster
} nebula_kind;
```

48. Since nebulae appear on the screen next to stars, and because they can have labels, they are descendants of **view_data**, too.

ngc, *ic*, and *messier* are of course the respective catalogue number. Since it's silly that *ngc* and *ic* are defined, a value of '0' means that the nebula is not part of the respective catalogue. *constellation* is a three-character string with the name of the respective constellation in all uppercase. *rectascension* is given in hours, *declination* in degrees. *diameter_x* and *diameter_y* are the extent of the nebula in the horizonal and the vertical direction and are given in degrees, *horizontal_angle* (in degrees) is the angle between the horizontal axis of *diameter_x* and the intersecting celestial rectascension circle. *magnitude* (in magnitudes) is the *total* brightness (not the brightness density).

```
struct nebula : public view_data {
  int ngc, ic, messier;    /* 0 if not defined. */
  string constellation;
  double rectascension, declination;
  double magnitude;
  double diameter_x, diameter_y;
  double horizontal_angle;
  nebula_kind kind;
  nebula()
  : ngc(0), ic(0), messier(0), constellation(), rectascension(0.0), declination(0.0), magnitude(0.0),
      diameter_x(0.0), diameter_y(0.0), horizontal_angle(0.0), kind(unknown) { }
  virtual double penalties_with(const double left, const double right, const double top, const double
      bottom, bool core = true) const;
};
typedef vector⟨nebula⟩ nebulae_list;
```

49. In order to find the penalties with a (labelled) nebula, I first calculate them for the label itself, which may be 0.0, in particular if *label_arranged* is still *false*. Then I determine the rectangular overlap, just like in **view_data**::*penalties_with*( ). This is unfortunate, because nebulae are circles and not rectangles. FixMe: This should be done justice to.

```
double nebula::penalties_with(const double left, const double right, const double top, const double
        bottom, bool core) const
{
  double penalties = view_data::penalties_with(left, right, top, bottom, core);
  const double left2 = x − radius, right2 = x + radius, top2 = y + radius, bottom2 = y − radius;
  const double overlap_left = fmax(left, left2);
  const double overlap_right = fmin(right, right2);
  const double overlap_top = fmin(top, top2);
  const double overlap_bottom = fmax(bottom, bottom2);
  const double overlap_x = fdim(overlap_right, overlap_left);
  const double overlap_y = fdim(overlap_top, overlap_bottom);
  penalties += overlap_x ∗ overlap_y ∗ params.penalties_nebula;
  return penalties;
}
```

50. As you can see, the file format for a nebulae file is very simple, because there are no string fields with possible whitespace within them. It's just a stream of fields separated by whitespace. *kind* is an **int**, and it's the canonical translation of the **nebula_kind** to **int**.

If the *horizontal_angle* is unknown, is must be 720.0. *diameter_y* must always have a valid value, even if it's actually unknown; in this case it must be equal to *diameter_y*.

```
istream &operator≫(istream &in, nebula &n)
{
  int kind;
  in ≫ n.ngc ≫ n.ic ≫ n.messier ≫ n.constellation ≫ n.rectascension ≫ n.declination ≫ n.magnitude ≫
      n.diameter_x ≫ n.diameter_y ≫ n.horizontal_angle ≫ kind;
  n.kind = nebula_kind(kind);
  return in;
}
```

51. Here I read a set of nebulae from a file with the name *filename*. The format is just a stream of **nebula**e.

```
void read_nebulae(nebulae_list &nebulae)
{
  ifstream nebulae_file(params.filename_nebulae.c_str( ));
  nebula current_nebula;
  nebulae_file ≫ current_nebula;
  while (nebulae_file) {
    ⟨ Create nebula label 52 ⟩
    nebulae.push_back(current_nebula);
    nebulae_file ≫ current_nebula;
  }
}
```

52. In order to create a LaTeX-ready label, I first choose which catalog to use. It is from the Messier, the NGC, and the IC catalogue the first in which the nebula appears, i. e. the first non-zero catalogue number. The catalogue abbreviation itself is stored in *catalog*, whereas the **stringstream** *number* contains the number within that catalogue. I just concatenate both to the *label*.

```
⟨ Create nebula label 52 ⟩ ≡
  string catalogue;
  stringstream number;
  if (current_nebula.messier > 0) {
    catalogue = "\\Messier{";
    number ≪ current_nebula.messier;
  }
  else if (current_nebula.ngc > 0) {
    catalogue = "\\NGC{";
    number ≪ current_nebula.ngc;
  }
  else if (current_nebula.ic > 0) {
    catalogue = "\\IC{";
    number ≪ current_nebula.ic;
  }
  else throw string("Invalid␣catalogue:␣\"") + catalogue + '"';
  current_nebula.label = catalogue + number.str() + '}';
```

This code is used in section 51.

53. Constellation boundaries. They are stored in an external file called `constborders.dat`.

Is is very convenient to have a special data type for a point in the program that has created `constborders.dat`. In this program the advantage is not so big but it's sensible to use the same data structures here.

```
  struct point {
    double x, y;
    point(const double x, const double y)
    : x(x), y(y) { }
    point()
    : x(0.0), y(0.0) { }
  };
```

54. An object of type **boundary** contains one stright line of a constellation boundary. This consists of the two endpoints which come from the original boundary catalog by Delporte of 1930, and zero or more interpolated points calculated by Davenhall (1990). The interpolated points help to draw a curved line where this is necessary.

Every line usually belongs to exactly two constellations (of course). They are stored in the field *constellations*. FixMe: There are boundaries with only one owner. I don't know how this can happen.

```
  struct boundary {
    vector⟨point⟩ points;
    vector⟨string⟩ constellations;
    bool belongs_to_constellation(const string constellation) const;
  };
  typedef vector⟨boundary⟩ boundaries_list;
```

55. In this routine I use only the first three letters of the constellation abbreviation. The reason is that the original catalog uses `"SER1"` and `"SER2"` for "Serpent Caput" and "Serpent Cauda". However, I see them as one constellation. Be aware that this routine expects the constellation abbreviation in uppercase.

```
bool boundary::belongs_to_constellation(const string constellation) const
{
  for (int i = 0; i < constellations.size(); i++)
    if (constellations[i].substr(0, 3) ≡ constellation.substr(0, 3)) return true;
  return false;
}
```

56. Of course I need to be able to read the boundaries from `constborders.dat`. The input stream is a text stream of the following format. The set of celestial boundaries is stored as a set of elementary lines without kinks.

1. Number of points ($size_1$) in the line (**int**).
2. Repeated $size_1$ times:
   – rectascension of point in hours (**double**),
   – declination of point in degrees (**double**).
3. Number of constellations ($size_2$) touching this border line (**int**, should be be always 2, however it isn't with current data).
4. Repeated $size_2$ times:
   – All uppercase astronomical abbreviation of the constellation. It may distinguish between "`SER1`" and "`SER2`" for Serpens Caput and Serpens Cauda.

All fields are separated by whitespace.

```
istream &operator≫(istream &in, boundary &p)
{
  int size;

  in ≫ size;
  p.points.resize(size);
  for (int i = 0; i < size; i++) in ≫ p.points[i].x ≫ p.points[i].y;
  in ≫ size;
  p.constellations.resize(size);
  for (int i = 0; i < size; i++) in ≫ p.constellations[i];
  return in;
}
```

57. Here I read a set of boundaries from a file. It's simply a list of **boundary**'s.

```
void read_constellation_boundaries(boundaries_list &boundaries)
{
  ifstream boundaryfile(params.filename_boundaries.c_str());
  boundary current_boundary;

  boundaryfile ≫ current_boundary;
  while (boundaryfile) {
    boundaries.push_back(current_boundary);
    boundaryfile ≫ current_boundary;
  }
}
```

58. As a big exception to the other classes, I don't derive **boundary** itself from **view_data**, but its smaller brother, **boundary_atom**. In contrast to **boundary**, **boundary_atom** only contains two points of the view frame, between which a boundary line will be drawn. This is not totally accurate since boundary lines are not totally straight which may cause problems at the poles. However, it should be good enough.

```
struct boundary_atom : public view_data {
  point start, end;
  boundary_atom(point start, point end);
  virtual double penalties_with(const double left, const double right, const double top, const double
      bottom, bool core = true) const;
};
```

59. The nice thing about **boundary_atom** is that after it has been constructed, it's finished. Nothing has to be changed any more because all is known in the moment of construction.

Since boundary atoms are only created when they are visible, I can set *in_view* = *visible* etc.

```
boundary_atom::boundary_atom(point start, point end)
: start(start), end(end) {
  x = (start.x + end.x)/2.0;
  y = (start.y + end.y)/2.0;
  radius = hypot(end.x − start.x, end.y − start.y)/2.0;
  radius *= M_PI_2;
  in_view = visible;
  with_label = hidden;
  skip = 0.0;
}
```

60. The full algorithm that is used here is described in the ⟨ Definition of *line_intersection*( ) for intersection of two lines 62 ⟩. Except for peculiar cases there should be exact two intersections altogether, which means that it's the same as with constellation lines.

Objects of this type are created in ⟨ Create a **boundary_atom** for the *objects* 111 ⟩.

⟨ Definition of *line_intersection*( ) for intersection of two lines 62 ⟩

```
double boundary_atom::penalties_with(const double left, const double right, const double top, const
        double bottom, bool core) const
{
  double intersection;
  point r(end.x − start.x, end.y − start.y);
  vector⟨point⟩ intersection_points;
  if (line_intersection(left − start.x, r.x, start.y, r.y, bottom, top, intersection))
    intersection_points.push_back(point(left, intersection));
  if (line_intersection(right − start.x, r.x, start.y, r.y, bottom, top, intersection))
    intersection_points.push_back(point(right, intersection));
  if (line_intersection(top − start.y, r.y, start.x, r.x, left, right, intersection))
    intersection_points.push_back(point(intersection, top));
  if (line_intersection(bottom − start.y, r.y, start.x, r.x, left, right, intersection))
    intersection_points.push_back(point(intersection, bottom));
  if (start.x > left ∧ start.x < right ∧ start.y > bottom ∧ start.y < top) intersection_points.push_back(start);
  if (end.x > left ∧ end.x < right ∧ end.y > bottom ∧ end.y < top) intersection_points.push_back(end);
  if (intersection_points.empty()) return 0.0;
  if (intersection_points.size() ≠ 2) {
    cerr ≪ "pp3:␣Funny␣" ≪ intersection_points.size() ≪
        "-fold␣constellation␣boundary␣intersecrtion." ≪ endl;
    return 0.0;
  }
  const double length = hypot(intersection_points[1].x − intersection_points[0].x,
      intersection_points[1].y − intersection_points[0].y);
  return (core ? 8.0 ∗ params.penalties_boundary : 0.5 ∗ params.penalties_boundary_rim)/72.27 ∗ 2.54 ∗ length;
}
```

61. Constellation lines. This is not about the *boundaries*, but about the connection lines between the main stars of a given constellation. They are mere eye candy. I call their **struct** type "connections" to keep the name unique and concise.

A **connection** consists of *lines*. The point coordinates are screen coordinates in centimetres. *from* and *to* are the star star and the end star, given by their index in *stars*.

Notice that *start* and *end* aren't defined before *draw_constellation_lines*( ) has been called, *and* the constellation line is actually visible. Then they contain the screen coordinates of the start and the end point of the line.

```
struct connection : public view_data {
  point start, end;
  int from, to;
  connection(const int from, const int to)
  : from(from), to(to), start(), end() {
    in_view = visible;
    with_label = hidden;
    skip = 0.0;
  }
  virtual double penalties_with(const double left, const double right, const double top, const double
      bottom, bool core = true) const;
};
typedef vector⟨connection⟩ connections_list;
```

62. In order to calculate penalties, we need some sort of overlap. Since there is no area overlap between a line and a rectangle, I need the overlapping line length.

This is the first part of the algorithm. It is actually pretty simple, however hard to explain. We have two lines to intersect: One edge of the label rectangle and the constellation line. The rectangle is given by *left*, *right*, *top*, and *bottom* – as usual. The constellation line is given by their start point *start* and end point *end*. Or, in parameterised form:

$$\vec{x} = \begin{pmatrix} start.x \\ start.y \end{pmatrix} + \lambda \begin{pmatrix} end.x - start.x \\ end.y - start.y \end{pmatrix}, \qquad \lambda \in [0;1].$$

The edge of the rectangle is given by (left edge as example)

$$\left(\vec{x} - \begin{pmatrix} left \\ 0 \end{pmatrix} \right) \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix} = 0$$

$$\Rightarrow \quad start.x + \lambda(end.x - start.x) = left$$

$$\overset{end.x - start.x \neq 0}{\Leftrightarrow} \lambda = \frac{left - start.x}{end.x - start.x}$$

with the boundary condition $bottom < start.y + \lambda(end.y - start.y) =: intersection < top$. With the abbreviations/assignments (also exemplarily for the 'left' case)

$$\begin{aligned} numerator &= left - start.x, \\ denominator &= end.x - start.x, \\ zero\_point &= start\_y, \\ slope &= end.y - start.y, \\ min &= bottom, \quad max = top \end{aligned}$$

we get the following routine for finding out whether a certain label rectangle edge is intersected by the constellation line or not:

⟨ Definition of *line_intersection*( ) for intersection of two lines 62 ⟩ ≡

```
bool line_intersection(double numerator, double denominator, double zero_point, double slope, double
        min, double max, double &intersection)
{
  if (denominator ≡ 0) return false;
  const double lambda = numerator/denominator;
  intersection = zero_point + lambda ∗ slope;
  return lambda > 0.0 ∧ lambda < 1.0 ∧ intersection > min ∧ intersection < max;
}
```

This code is cited in section 60.

This code is used in section 60.

63. Now for the second part of the intersection algorithm. Here I apply the preceding routing on all four label edges. I return an overlap as the penalty value, however I pretend that the line has a width of 8 pt. For an overlap with the rim, it's only 0.5 pt.

```
double connection::penalties_with(const double left, const double right, const double top, const double
        bottom, bool core) const
{
  double intersection;
  point r(end.x − start.x, end.y − start.y);
  vector⟨point⟩ intersection_points;
  if (line_intersection(left − start.x, r.x, start.y, r.y, bottom, top, intersection))
    intersection_points.push_back(point(left, intersection));
  if (line_intersection(right − start.x, r.x, start.y, r.y, bottom, top, intersection))
    intersection_points.push_back(point(right, intersection));
  if (line_intersection(top − start.y, r.y, start.x, r.x, left, right, intersection))
    intersection_points.push_back(point(intersection, top));
  if (line_intersection(bottom − start.y, r.y, start.x, r.x, left, right, intersection))
    intersection_points.push_back(point(intersection, bottom));
  if (start.x > left ∧ start.x < right ∧ start.y > bottom ∧ start.y < top) intersection_points.push_back(start);
  if (end.x > left ∧ end.x < right ∧ end.y > bottom ∧ end.y < top) intersection_points.push_back(end);
  if (intersection_points.empty()) return 0.0;
  if (intersection_points.size() ≠ 2) {
    cerr ≪ "pp3:␣Funny␣" ≪ intersection_points.size() ≪
        "-fold␣constellation␣line␣intersecrtion." ≪ endl;
    return 0.0;
  }
  const double length = hypot(intersection_points[1].x − intersection_points[0].x,
      intersection_points[1].y − intersection_points[0].y);
  return (core ? 8.0 ∗ params.penalties_cline : 0.5 ∗ params.penalties_cline_rim)/72.27 ∗ 2.54 ∗ length;
}
```

64. I must be able to read a file which contains such data. Here, too, the text file format is very simple: It's a list of constellation line paths separated by '`;`'. A star name must be of the form

Constellation␣*Flamsteed number*

or

`HD`␣*Henry-Draper Catalogue number*

where *Constellation* must be given as an all uppercase three letter abbreviation. For example, `ORI␣19` is α Ori (Rigel).

The hash sign '`#`' introduces comments that are ignored till end of line, however they mustn't occur within a star.

```
void read_constellation_lines(connections_list &connections, const stars_list &stars)
{
  ifstream file(params.filename_lines.c_str());
  string from_catalogue_name, to_catalogue_name;
  int from_catalogue_index = 0, to_catalogue_index = 0;
  file >> to_catalogue_name;
  while (file) {
    if (to_catalogue_name[0] ≡ '#') {      /* skip comment */
      string dummy;
      getline(file, dummy);
    }
    else if (to_catalogue_name ≡ ";")      /* start a new path */
      from_catalogue_index = 0;
    else {
      file >> to_catalogue_index;
      if (from_catalogue_index > 0 ∧ to_catalogue_index > 0) {
        ⟨ Create one connection 65 ⟩
      }
      from_catalogue_name = to_catalogue_name;
      from_catalogue_index = to_catalogue_index;
    }
    file >> to_catalogue_name;
  }
}
```

65. In the loop I try to find the index within *stars* of the 'from' star and the 'to' star.

```
⟨ Create one connection 65 ⟩ ≡
  int from_index = −1, to_index = −1;
  for (int i = 0; i < stars.size(); i++) {
    ⟨ Test whether stars[i] is the 'from' star 66 ⟩
    ⟨ Test whether stars[i] is the 'to' star 67 ⟩
  }
  if (from_index ≡ −1 ∨ to_index ≡ −1) {
    stringstream error_message;
    error_message ≪ "Unrecognised␣star␣in␣constellation␣lines:␣";
    if (from_index ≡ 0) error_message ≪ from_catalogue_name ≪ '␣' ≪ from_catalogue_index;
    else error_message ≪ to_catalogue_name ≪ '␣' ≪ to_catalogue_index;
    throw error_message.str();
  }
  connections.push_back(connection(from_index, to_index));
```

This code is used in section 64.

66. Here I test whether the current star in the loop is the 'from' star. Of course only if I haven't found it already (*from_index* ≡ 0). If apparently both stars have been found already, I leave the loop immediately.

```
⟨ Test whether stars[i] is the 'from' star 66 ⟩ ≡
  if (from_index ≡ −1)
    if (from_catalogue_name ≡ "HD") {
      if (stars[i].hd ≡ from_catalogue_index) from_index = i;
    }
    else {
      if (from_catalogue_name ≡ stars[i].constellation)
        if (stars[i].flamsteed ≡ from_catalogue_index) from_index = i;
    }
  else if (to_index ≠ −1) break;
```

This code is cited in section 67.

This code is used in section 65.

67. Perfectly analogous to ⟨ Test whether *stars*[*i*] is the 'from' star 66 ⟩.

```
⟨ Test whether stars[i] is the 'to' star 67 ⟩ ≡
  if (to_index ≡ −1)
    if (to_catalogue_name ≡ "HD") {
      if (stars[i].hd ≡ to_catalogue_index) to_index = i;
    }
    else {
      if (to_catalogue_name ≡ stars[i].constellation)
        if (stars[i].flamsteed ≡ to_catalogue_index) to_index = i;
    }
  else if (from_index ≠ −1) break;
```

This code is used in section 65.

68. The Milky Way. The file is a text file as usual with the following structure, everything separated by whitespace:

1. The maximal (= equatorial) diagonal half distance of two pixels in degrees (**double**). This value is used as the *radius* for the milky way 'pixels'. Of course it must be the minimal radius for which there are no gaps between the pixels.
2. The pixels themselves with two **double**s and one **int** each:
   - The rectascension in hours.
   - The declination in degrees.
   - The grey value of the pixel from 1 to 255. Zero is not used because zero-value pixels are not included into the data file anyway.

In PP3's standard distribution, this file was produced with the helper program `milkydigest.cc`, and the original Milky Way bitmap was photographed and compiled by Axel Mellinger.

pixels is a **vector**⟨**vector**⟨**point**⟩⟩. The outer (first) index is the grey value, and for every grey value there is an inner vector (second index) with a list of all points (rectascension, declination) that have this value. This construction makes the drawing in Ghostview looking nice, but more importantly it reduces the number of colour change commands in the LaTeX file. Unfortunately this doesn't reduce pool size usage.

```
⟨ Reading the milkyway into pixels 68 ⟩ ≡
  ifstream file(params.filename_milkyway.c_str());
  double radius;

  file ≫ radius;

  double rectascension, declination, x, y;
  int pixel;
  const double cm_per_grad = 1.0/(mytransform.get_rad_per_cm() * 180.0/M_PI);

  radius *= cm_per_grad/2.54 * 72.27;
  file ≫ rectascension ≫ declination ≫ pixel;
  while (file) {
    if (mytransform.polar_projection(rectascension, declination, x, y)) pixels[pixel].push_back(point(x, y));
    file ≫ rectascension ≫ declination ≫ pixel;
  }
```

This code is cited in section 114.

This code is used in section 114.

69. Colour. It's very convenient to have a unified data structure for all colours that appear in this program. Its internal structure is trivial, and I only support the RGB colour model. The only complicated thing is *name* here. I need it because of PSTricks' way to activate colours: They must get names first. I could get rid of it if I called all colours e. g. "`tempcolor`" or "`dummycolor`" and activated them at once. But this is not necessary.

```
⟨ Define color data structure 69 ⟩ ≡
  struct color {
    double red, green, blue;
    string name;
    color(string name, double red, double green, double blue)
    : red(red), green(green), blue(blue), name(name) { }
    color(double red, double green, double blue)
    : red(red), green(green), blue(blue), name() { }
    void set(ostream &out) const;
  };
```

This code is used in section 8.

70. This routine is used when the name of the colour is not necessary, because it's only needed locally. This is the case for text labels and the Milky Way dots.

```
  void color::set(ostream &out) const
  {
    out ≪ "\\newrgbcolor{dummycolor}{" ≪ red ≪ '␣' ≪ green ≪ '␣' ≪ blue ≪
        "}\\dummycolor\n␣␣\\psset{linecolor=dummycolor}%\n";
  }
```

71. Both output and input of **color**s is asymmetric: When I *read* them I assume that I do it from an input script. Then it's a mere sequence of the three colour values.

```
  istream &operator≫(istream &in, color &c)
  {
    in ≫ c.red ≫ c.green ≫ c.blue;
    if (¬in
     ∨ c.red < 0.0 ∨ c.red > 1.0
     ∨ c.green < 0.0 ∨ c.green > 1.0
     ∨ c.blue < 0.0 ∨ c.blue > 1.0)  throw string("Invalid␣RGB␣values␣in␣input␣script");
    return in;
  }
```

72. But when I *write* them, I assume that I do it into a LaTeX file with PSTricks package activated. Then I deploy a complete `\newrgbcolor` command.

```
  ostream &operator≪(ostream &out, const color &c)
  {
    if (c.name.empty())  throw string("Cannot␣write␣unnamed␣color␣to␣stream");
    out ≪ "\\newrgbcolor{" ≪ c.name ≪ "}{" ≪ c.red ≪ '␣' ≪ c.green ≪ '␣' ≪ c.blue ≪ "}%\n";
    return out;
  }
```

73. This routine is hitherto only used when drawing the milky way. It helps to find a colour between the two extremes *c1* and *c2*. The value of *x* is always between 0 and 1 and denotes the point on the way between *c1* and *c1* in the RGB colour space where the new colour should be. I interpolate linearly. In order to create a new colour object, I need a *new_name* for it, too.

```
color interpolate_colors(const double x, const color c1, const color c2, const string new_name = "")
{
  if (x < 0.0 ∨ x > 1.0) throw string("Invalid␣x␣for␣color␣interpolation");
  const double y = 1.0 − x;
  return color(new_name, y ∗ c1.red + x ∗ c2.red, y ∗ c1.green + x ∗ c2.green, y ∗ c1.blue + x ∗ c2.blue);
}
```

74. Drawing the chart. This is done in two steps. First the lines and the celestial objects are printed. During this phase a lot of labels may accumulate. They cannot be drawn at that phase, because the arrangement with minimal overlaps can only be calculated when all are known.

Therefore I have to fill a container called *objects* which contains elements of type **view_data**. The part of each **star** or **nebula** or whatever that is **view_data** is appended to *objects*, if and only if the object is visible in the view frame. The typical command for that is

```
objects.push_back(&stars[i]);
```

(here for stars). Please notice that it is *not* important whether or not the respective object bears a label. Its data is needed in any case.

In the second phase I arrange and print the labels.

75. Coordinate transformations. They are done by the class **transformation**. *width* and *height* contain the view frame dimensions in centimetres. *rad_per_cm* is the resolution.

The 3×3 matrices a and *a_unscaled* are rotation matrices for the transformation in cartesian space from the equatorial system into the azimuthal system, where the center of the view frame is the z axis. While *a_unscaled* refers to a celestial sphere with radius 1, a contains the additional scaling for the actual centimetres on the paper.

```
class transformation {
  double width, height, rad_per_cm;
  double a[3][3], a_unscaled[3][3];
  inline double stretch_factor(double z) const;
public: transformation(const double rectascension, const double declination, const double width, const
      double height, const double grad_per_cm);
  bool polar_projection(const double rectascension, const double declination, double &x, double &y)
      const;
  double get_rad_per_cm() const { return rad_per_cm; }
};
```

76. Starting point is the parallel projection of a celestial unit sphere on the x-y plane. This projection looks like a sky globus seen from far away with a strong zoom objective.

It is the starting point because it simply is a necessary intermediate step: All celestial positions given in polar coordinates (*rectascension*, *declination*) must be transformed to cartesian coordinates in order to apply a rotation on them for having the center of the view frame in the center of the coordinate system. From there it's a trivial step to the parallel projection.

But it squeezes the rim areas too much, which I don't like.

By *stretch* I mean the radial stretch factor that should relieve this distortion. If *stretch* was 1.0, we would get the mentioned parallel projection.

Therefore I stretch the plot here so that this effect is minimised. The result is the equidistant azimuthal projection that can be described best if you imagine a plot with its center exactly on the north pole: All circles of equal declination have then the same distance from each other. So the plot keeps its circular form. The *radial* scale is constant everywhere and equal to *grad_per_cm*. Perpendicular to that, the scale is decreasing from *grad_per_cm* (center) to $2/\pi \cdot \mathit{grad\_per\_cm}$ (border). The exact relation is

$$\begin{aligned}
\mathit{scale}_{\parallel} &= \mathit{grad\_per\_cm}, \\
\mathit{scale}_{\perp} &= \mathit{grad\_per\_cm} \cdot \frac{r}{\arccos\sqrt{1-r^2}}, \quad \text{and thus} \\
\mathit{stretch} &= \frac{\arccos\sqrt{1-r^2}}{r}. \qquad (1)
\end{aligned}$$

r is simply the distance from the center of the plot and it is 1 on the border. But I don't have r, I have z. I could to the substitution $r = \sqrt{1-z^2}$ leading to a very badly converging Taylor series, but much better is $\tilde{z} = 1 - z$ and to use a Taylor series of

$$\mathit{stretch} = \frac{\arccos(1-\tilde{z})}{\sqrt{1-(1-\tilde{z})^2}}.$$

Maple V says

$$\mathit{stretch} = 1 + 1/3\,\tilde{z} + 2/15\,\tilde{z}^2 + \frac{2}{35}\,\tilde{z}^3 + \frac{8}{315}\,\tilde{z}^4 + \frac{8}{693}\,\tilde{z}^5 + \mathcal{O}(\tilde{z}^{11/2}).$$

This is good enough (error less than 1 %). There are two alternatives:

1. Use the Tayor expansion of (1) directly, because this expansion converges very quickly, especially for the center-near regions. This would mean to calculate $r(z)$ for every point, but this shouldn't be too costly.
2. Transform back in spherical coordinates, re-interpret them as planar polar coordinates and transform them to planar cartesian. Probably too difficult.

Why not calculating (1) (maybe with a subsitution) directly without series expansion? First, it may be too costly. But secondly, I don't like that in the particularly interesting region close to the center of the view frame both numerator and denominator get close to 0, and eventually they do reach it. Maybe I'm paranoid, but I don't like that.

```
inline double transformation::stretch_factor(const double z) const
{
  const double z̃ = 1.0 − z;
  return 1.0 + z̃ ∗ (1.0/3.0 + z̃ ∗ (2.0/15.0 + z̃ ∗ (2.0/35.0 + z̃ ∗ (8.0/315.0 + z̃ ∗ (8.0/693.0)))));
}
```

77. This is not only the constructor for **transformation**, it is also the initialiser for the whole transformation. As much work as possible is this routine, in order to keep calculations easy in the function *polar_projection*( ) which will be called hundreds if not thousands of times.

rectascension is given in hours, together with *declination* it's the center of the desired view frame. *width* and *height* give its dimensions in centimetres, *grad_per_cm* is the resulting resolution in the center.

$$a\_unscaled = \begin{pmatrix} \sin\varphi & \cos\varphi & 0 \\ \cos\varphi\cos\alpha & -\sin\varphi\cos\alpha & \sin\alpha \\ -\cos\varphi\sin\alpha & \sin\varphi\sin\alpha & \cos\alpha \end{pmatrix}.$$

```
transformation::transformation(const double rectascension, const double declination, const double
        width, const double height, const double grad_per_cm)
{
  const double phi = −(rectascension + 12) ∗ 15.0 ∗ M_PI/180.0;
  const double delta = declination ∗ M_PI/180.0;
  const double alpha = −delta + M_PI_2;
  rad_per_cm = grad_per_cm ∗ M_PI/180.0;
  a_unscaled[0][0] = sin(phi);
  a_unscaled[0][1] = cos(phi);
  a_unscaled[0][2] = 0.0;
  a_unscaled[1][0] = cos(phi) ∗ cos(alpha);
  a_unscaled[1][1] = −sin(phi) ∗ cos(alpha);
  a_unscaled[1][2] = sin(alpha);
  a_unscaled[2][0] = −cos(phi) ∗ sin(alpha);
  a_unscaled[2][1] = sin(phi) ∗ sin(alpha);
  a_unscaled[2][2] = cos(alpha);
  for (int i = 0; i < 3; i++)
    for (int j = 0; j < 3; j++)  a[i][j] = a_unscaled[i][j]/rad_per_cm;
  transformation::width = width;
  transformation::height = height;
}
```

78. Here I transform the equatorial position (*rectascension*, *declination*) to the cartesian position (x, y). The resulting cartesian positions represents an azimuthal equidistant projection with the center of the view frame (see *rectascension* and *declination* in the constructor of **transformation**).

It returns *true* if the resulting point is within the view frame, otherwise *false*.

```
bool transformation::polar_projection(const double rectascension, const double declination, double
        &x, double &y) const
{
  const double phi = rectascension ∗ 15.0 ∗ M_PI/180.0;
  const double delta = declination ∗ M_PI/180.0;
  const double cos_delta = cos(delta);
  const double x0 = cos_delta ∗ cos(phi);
  const double y0 = cos_delta ∗ sin(phi);
  const double z0 = sin(delta);
  const double z1 = a_unscaled[2][0] ∗ x0 + a_unscaled[2][1] ∗ y0 + a_unscaled[2][2] ∗ z0;
  if (z1 < −DBL_EPSILON) return false;
  const double stretch = stretch_factor(z1);
  const double x1 = a[0][0] ∗ x0 + a[0][1] ∗ y0;
  const double y1 = a[1][0] ∗ x0 + a[1][1] ∗ y0 + a[1][2] ∗ z0;
  x = x1 ∗ stretch + width/2.0;
  y = y1 ∗ stretch + height/2.0;
  if (x < 0.0 ∨ x > width ∨ y < 0.0 ∨ y > height) return false;
  return true;
}
```

79. Label organisation. Without labels, star charts are not very useful. But labels mustn't overlap, and they should not overlap with other chart elements such as star circles or nebula circles. Here I try to develop a simple algorithm that is able to avoid most of these problems. There are two main routines here: *arrange_labels*( ) and *print_labels*( ).

arrange_labels( ) modifies the *label_angle* field in each celestial object in order to avoid any overlap with other objects, namely other labels, stars, or nebulae. It does so by testing all eight values for *label_angle* and calculating a *penalty* value for each of them. This *penalty* is

$$penalty = overlap_{\text{labels}} + overlap_{\text{objects}} + penalty_{\text{lines}}.$$

print_labels( ) actually generates the LaTeX code for printing them.

80. First the routine that actually calculates the overlap. It simply finds the common rectangular area in squared centimetres. Both rectangles are given by their boundaries, *left1*, *right1*, *top1*, *bottom1* enclose the first rectangle, *left2*, *right2*, *top2*, *bottom2* the second.

```
double calculate_overlap(double left1, double right1, double top1, double bottom1, double left2, double
        right2, double top2, double bottom2)
{
  const double overlap_left = fmax(left1, left2);
  const double overlap_right = fmin(right1, right2);
  const double overlap_top = fmin(top1, top2);
  const double overlap_bottom = fmax(bottom1, bottom2);
  const double overlap_x = fdim(overlap_right, overlap_left);
  const double overlap_y = fdim(overlap_top, overlap_bottom);
  return overlap_x * overlap_y;
}
```

81. Nor for a structure with the real dimensions in centimetres of all possible labels. They are read from the file `labeldims.dat` and stored in a **dimensions_list**.

```
struct dimension {
  double width, height, depth;
  dimension()
  : width(0.0), height(0.0), depth(0.0) { }
  dimension(const dimension &d)
  : width(d.width), height(d.height), depth(d.depth) { }
};
typedef map⟨string, dimension⟩ dimensions_list;
```

82. Now we re-arrange the labels, namely *objects*[*i*].*with_label* for all *i*. For efficiency, I first find all neighbours of the on-object and do all the following work only with them. In the inner *k*-loop I test all possible *label_angle*s and calculate their *penalty*.

If a label leaps out of the view frame, this adds to *penalty* the gigantic value of 10000.0.

```
⟨ create_preamble( ) for writing the LaTeX preamble 120 ⟩
⟨ Helping routines for nebulae labels 88 ⟩
void arrange_labels(objects_list &objects, dimensions_list &dimensions)
{
  objects_list vicinity;
  ⟨ Set label dimensions 87 ⟩
  for (int i = 0; i < objects.size( ); i++) {
    vicinity.clear( );
    if (objects[i]→in_view ≡ visible ∧ objects[i]→with_label ≡ visible ∧ ¬objects[i]→label_arranged) {
      ⟨ Find objects in vicinity of objects[i] 83 ⟩
      double best_penalty = DBL_MAX;
      int best_angle = 0;
      for (int k = 0; k < 8; k++) {
        objects[i]→label_angle = k;
        double on_object_left, on_object_right, on_object_top, on_object_bottom;
        objects[i]→get_label_boundaries(on_object_left, on_object_right, on_object_top, on_object_bottom);
        double penalty = 0.0;
        double rim_width = 2.0 ∗ objects[i]→skip;
        for (int j = 0; j < vicinity.size( ); j++) {
          penalty += vicinity[j]→penalties_with(on_object_left, on_object_right, on_object_top, on_object_bottom) +
              vicinity[j]→penalties_with(on_object_left − rim_width, on_object_right + rim_width,
              on_object_top + rim_width, on_object_bottom − rim_width, false) ∗ params.penalties_rim;
        }
        if (on_object_left < 0.0 ∨ on_object_bottom < 0.0 ∨ on_object_right >
              params.view_frame_width ∨ on_object_top > params.view_frame_height)
          penalty += 10000.0;
        if (penalty < best_penalty) {
          best_penalty = penalty;
          best_angle = k;
        }
      }
      if (¬objects[i]→label.empty( ))
        if (best_penalty < 0.4 ∗ params.penalties_threshold ∗ objects[i]→label_height ∗ objects[i]→label_width) {
          objects[i]→label_angle = best_angle;
          objects[i]→label_arranged = true;
#ifdef DEBUG
          stringstream penalty;
          penalty.precision(2);
          penalty ≪ "␣" ≪ best_penalty ≪ "␣(" ≪ best_penalty/(objects[i]→label_height ∗
              objects[i]→label_width) ∗ 100 ≪ "%)";      /∗ objects[i]→label += penalty.str( ); ∗/
          cerr ≪ "pp3DEBUG:␣Object␣" ≪ objects[i]→label ≪ '␣';
          const star ∗s = dynamic_cast⟨star ∗⟩(objects[i]);
```

```
            if (s) cerr ≪ s→constellation;
            cerr ≪ "␣has␣penalty␣of" ≪ penalty.str() ≪ endl;
#endif
          }
          else {
            objects[i]→with_label = hidden;
            objects[i]→label_arranged = true;
          }
        }
      }
    }
```

83. All objects in the vicinity of *objects*[*i*] eventually end up in the **vector** *vicinity*. Here I fill this structure. I use a very rough guess for finding the neighbours, so there will probably be too many of them, but it makes calculation much easier.

The practical thing is that neighbouring objects are ordered in increasing brightness in star data file, which means that lables of bright stars are arranged first, and labels of fainter stars must cope with these positions.

Of course, it's guaranteed that *objects*[*i*] is not part of its vicinity.

⟨ Find objects in vicinity of *objects*[*i*] 83 ⟩ ≡

```
  const double on_object_scope = objects[i]→scope();
  for (int j = 0; j < objects.size(); j++) {
    if (i ≠ j ∧ objects[j]→in_view ≡ visible) {
      const double distance = hypot(objects[i]→x − objects[j]→x, objects[i]→y − objects[j]→y);
      if (distance < on_object_scope + objects[j]→scope() ∧ ⟨ Condition to exclude double stars and such 84 ⟩)
        vicinity.push_back(objects[j]);
    }
  }
```

This code is cited in section 39.

This code is used in section 82.

84. ⟨ Condition to exclude double stars and such 84 ⟩ ≡

```
  (distance > objects[j]→skip ∨ (objects[j]→with_label ≡ visible ∧ ¬objects[j]→label.empty()))
```

This code is used in section 83.

85. Finally I print out all labels by generation LaTeX code from any of them. I do that by calculating the coordinates in centimetres of the *bottom left* corner of the label box, and placing the TeX box there. This TeX box lies within another one with zero dimensions in order to keep the point of origin (bottom left of the view frame) intact.

```
  void print_labels(const objects_list &objects)
  {
    for (int i = 0; i < objects.size(); i++)
      if (objects[i]→in_view ≡ visible ∧ objects[i]→with_label ≡ visible ∧ objects[i]→label_arranged) {
        double left, right, top, bottom;

        objects[i]→get_label_boundaries(left, right, top, bottom);
        if (left < 0.0 ∨ bottom < 0.0 ∨ right > params.view_frame_width ∨ top > params.view_frame_height)
          continue;
        objects[i]→label_color.set(OUT);
        OUT ≪ "\\hbox␣to␣0pt{";
        OUT ≪ "\\hskip" ≪ left ≪ "cm";
        OUT ≪ "\\vbox␣to␣0pt{\\vss\n␣␣\\hbox{\\Label{";
        OUT ≪ objects[i]→label;
        OUT ≪ "}}\\vskip" ≪ bottom ≪ "cm";
        OUT ≪ "\\hrule␣height␣0pt}\\hss}%\n";
#ifdef DEBUG
        OUT ≪ "\\psframe[linewidth=0.1pt](" ≪ left ≪ ',' ≪ bottom ≪ ")(" ≪ right ≪ ',' ≪
            bottom + objects[i]→label_depth ≪ ")%\n";
        OUT ≪ "\\psframe[linewidth=0.3pt](" ≪ left ≪ ',' ≪ bottom ≪ ")(" ≪ right ≪ ',' ≪
            top ≪ ")%\n";
#endif
      }
  }
```

86. Here I read label dimensions from a text file. The format is very simple:

1. The LaTeX representation of the label on a line of its own.
2. In the following line, width, height, and depth of the label in centimetres (both **double**), separated by whitespace.

This is repeated for every data record.

```
  void read_label_dimensions(dimensions_list &dimensions)
  {
    ifstream file(params.filename_dimensions.c_str());
    string name, dummy;

    getline(file, name);
    while (file) {
      file ≫ dimensions[name].width ≫ dimensions[name].height ≫ dimensions[name].depth;
      getline(file, dummy);      /* Read the '\n' */
      getline(file, name);
    }
  }
```

87. Determining label dimensions. Here I go through all *objects* and set the *label_width*, *label_height*, and *label_depth* which have been zero so far. It may happen that a label is not found (possibly because *dimensions* is totally empty because no label dimensions file could be found). In this case I call *recalculate_dimensions*( ) to get all labels recalculated via an extra LaTeX run.

dimensions_recalculated is *true* if *recalculate_dimensions*( ) has been called and thus one can assume that all needed labels are now available. It is merely to remove unnecessary tests and make the procedure faster.

The **throw** command should never happen. It means an internal error.

```
⟨ Set label dimensions 87 ⟩ ≡
  bool dimensions_recalculated = false;
  for (int i = 0; i < objects.size( ); i++) {
    view_data *current_object = objects[i];
    if (current_object→with_label ≡ visible) {
      if (¬dimensions_recalculated ∧ dimensions.find(current_object→label) ≡ dimensions.end( )) {
        recalculate_dimensions(dimensions, objects);
        dimensions_recalculated = true;
        if (dimensions.find(current_object→label) ≡ dimensions.end( ))
          throw string("Update␣of␣label␣dimensions␣file␣failed:␣\"") + current_object→label +
              "\"␣not␣found";
      }
      current_object→label_width = dimensions[current_object→label].width;
      current_object→label_height = dimensions[current_object→label].height;
      current_object→label_depth = dimensions[current_object→label].depth;
    }
  }
```

This code is used in section 82.

88. When PP3 is started it reads a file usually called `labeldimens.dat` in order to know width, height, and depth (in centimetres) of all labels. This is vital for the penalty (i. e. overlap) calculations. But it may be that a label that you want to include can't be found in this file, or you have deleted it because you've changed the LaTeX preamble of the output (i. e. the fonts). In these cases PP3 automatically creates a new one. It is then used in the following runs to save time.

Here I do this. First I store all label names (not only the missing!) in *required_names*. I assure that every label occurs only once.

Then I create a *temp_file* which is a LaTeX file that – if sent through LaTeX – is able to create another temporary file called *raw_labels_file*. This is read and stored directly in *dimensions* (where it belongs to naturally). At the same time a new, updated *cooked_labels_file* (aka `labeldimens.dat`) is created.

By the way, I am forced to use *two* temporary files. It is impossible to let the LaTeX file create directly the file `labeldimens.dat`. The reason are the label string: They may contain characters that have a special meaning in LaTeX, and I'm unable to avoid any tampering.

⟨ Helping routines for nebulae labels 88 ⟩ ≡

```
void recalculate_dimensions(dimensions_list &dimensions, const objects_list &objects)
{
  list⟨string⟩ required_names;
  for (int i = 0; i < objects.size(); i++) {
    const string current_name = objects[i]→label;
    if (¬current_name.empty()) required_names.push_back(current_name);
  }
  required_names.unique();
  ofstream temp_file("pp3temp.tex");
  create_preamble(temp_file);
  temp_file ≪ "\n\\begin{document}\n" ≪ "\\newwrite\\labelsfile\n" ≪
      "\\catcode`\\_=11␣␣%␣for␣underscores␣in␣the␣filename\n" ≪
      "\\immediate\\openout\\labelsfile=pp3temp.dat\n" ≪ "\\catcode`\\_=8\n";
  list⟨string⟩::const_iterator p = required_names.begin();
  while (p ≠ required_names.end()) temp_file ≪ "\\setbox0␣=␣\\hbox{" ≪ *(p++) ≪
        "}\n␣␣\\immediate\\write\\labelsfile{""\\the\\wd0s␣\\the\\ht0s␣\\the\\dp0s}\n";
  temp_file ≪ "\\immediate\\closeout\\labelsfile\n\\end{document}\n";
  temp_file.close();
  string commandline("latex␣pp3temp");
  if (params.filename_output.empty()) commandline += "␣>␣pp3dump.log";
  if (system(commandline.c_str()) ≠ 0)
    throw string("Label␣dimensions␣calculations:␣LaTeX␣call␣failed:␣") + commandline;
  ifstream raw_labels_file("pp3temp.dat");
  ofstream cooked_labels_file("labeldimens.dat");
  cooked_labels_file.setf(ios::fixed);
  cooked_labels_file.precision(5);
  p = required_names.begin();
  while (p ≠ required_names.end()) {
    string current_width, current_height, current_depth;
    string current_name;
    current_name = *(p++);
    raw_labels_file ≫ current_width ≫ current_height ≫ current_depth;
    current_width.substr(0, current_width.length() − 3);
```

```
      current_height.substr(0, current_height.length() − 3);
      current_depth.substr(0, current_depth.length() − 3);
      dimensions[current_name].width = strtod(current_width.c_str(), 0)/72.27 ∗ 2.54;
      dimensions[current_name].height = strtod(current_height.c_str(), 0)/72.27 ∗ 2.54;
      dimensions[current_name].depth = strtod(current_depth.c_str(), 0)/72.27 ∗ 2.54;
      dimensions[current_name].height += dimensions[current_name].depth;
      cooked_labels_file ≪ current_name ≪ '\n' ≪ dimensions[current_name].width ≪ '␣' ≪
          dimensions[current_name].height ≪ '␣' ≪ dimensions[current_name].depth ≪ '\n';
    }
  }
```

This code is used in section 82.

89. User labels. The user should be able to insert arbitaray text into the chart. The code for this is provided here. The data type of them, **text**, is of course a classical derivative of **view_data**. It only holds the celestial coordinates and the text (LaTeX) string of the label.

```
  struct text : public view_data {
    string contents;
    double rectascension, declination;
    text(string t, double r, double d, color c, int angle, bool on_baseline);
  };
  typedef list⟨text⟩ texts_list;
```

90. In the constructor I modify the values of some **view_data** elements, because a **text** is a special label. For example it mustn't be arranged, because it is already. And it gets another colour (if wanted).

```
  text::text(string t, double r, double d, color c, int angle, bool on_baseline)
  : contents(t), rectascension(r), declination(d) {
    label = string("\\TextLabel{") + t + '}';
    with_label = visible;
    label_angle = angle;
    view_data::on_baseline = on_baseline;
    label_arranged = true;
    label_color = c;
    radius = skip = 0.0;
  }
```

91. This routine is called *draw_...* due to its analogy to the other drawing function. But curiously enough, I don't draw anything here, because labels are drawn in *print_labels*( ) and **text**s consists only of labels. I only have to assure that I don't include invisible labels.

```
void draw_text_labels(transformation &mytransform, texts_list &texts, objects_list &objects)
{
  texts_list::iterator p = texts.begin();
  while (p ≠ texts.end()) {
    double x, y;
    if (mytransform.polar_projection(p→rectascension, p→declination, x, y)) {
      p→in_view = visible;
      p→x = x;
      p→y = y;
      objects.push_back(&(*p));
    }
    p++;
  }
}
```

92. Flex labels. "Flex labels" are drawn along a path that needn't (and usually isn't) be a straight line. Unfortunately, due to this, their dimensions are very difficult to predict and therefore they are *not* part of the penalty algorithm – at least not in the current version.

Eventually PP3 could have many kinds of flexes, therefore there is a common (abstract) ancestor for all of them called **flex_label**. But at the moment I only implement the by far most important one, the **declination_flex**, see below.

Flexes are special also in another respect: They are read from the input script. This means that the *flexes_list* that contains all flexes (of all kinds) is filled during reading the input script. Therefore there is no *read_flexes*( ), and no file format associated with it.

```
struct flex_label : public view_data {
  double rectascension, declination;
  flex_label(string s, double r, double d, color c, int a, bool b);
  virtual bool draw(const transformation &mytransform, dimensions_list &dimensions, objects_list
      &objects) const = 0;
};
```

93. Of course I need no extra label because a flex is a label of its own. So a flex only uses *label* and *label_angle*. The rest is unimportant because the label isn't printed anyway because the coodrinates x and y are invalid anyway and therefore printing will be rejected in *print_labels*( ).
Actually *on_baseline* is insignificant for flexes.

```
flex_label :: flex_label(string s, double r, double d, color c, int a, bool b)
: rectascension(r), declination(d) {
  label_color = c;
  label = string("\\FlexLabel{") + s + '}';
  label_angle = a;
  on_baseline = b;
  in_view = visible;
  with_label = hidden;
  label_arranged = true;
}
typedef list⟨flex_label *⟩ flexes_list;
```

94. **declination_flex** enables us to write a text along a path of constant declination. This looks very nice on the chart.

```
struct declination_flex : public flex_label {
  declination_flex(string s, double r, double d, color c, int a, bool b)
  : flex_label(s, r, d, c, a, b) { }
  virtual bool draw(const transformation &mytransform, dimensions_list &dimensions, objects_list
      &objects) const;
  virtual double penalties_with(const double left, const double right, const double top, const double
      bottom, bool core = true) const;
};
```

95. This is the main declination flex routine. I calculate the dimensions of the label in oder to find out how long the path must be and how much it must be shifted vertically in order to get the desired angular positioning. Both values can only be estimates. (For example, the scale is not constant on the map, so it cannot work perfectly. However I try to assure that the path will be too long rather than too short.)

Then I calculate the coordinates of three points that form the path. Start point, end point, and one exactly in between. Of course all have the same declination. *lower* is measured in em and denotes the amount by which the whole text is shifted downwards. This is sub-optimal because it may affect letterspacing disadvantageously.

Finally the three coordinate pairs and the label text are sent to the output in form of a PSTricks text path command.

```
bool declination_flex::draw(const transformation &mytransform, dimensions_list
        &dimensions, objects_list &objects) const
{
  if (dimensions.find(label) ≡ dimensions.end()) recalculate_dimensions(dimensions, objects);
  if (dimensions.find(label) ≡ dimensions.end())
    throw string("Update␣of␣label␣dimensions␣file␣failed:␣\"") + label + "\"␣not␣found";
  const double label_width = dimensions[label].width;
  const double label_height = dimensions[label].height;
  const double rectascension_halfwidth = label_width ∗ mytransform.get_rad_per_cm() ∗
      180.0/M_PI/15.0/cos(declination ∗ M_PI/180.0)/2.0;
  char justification;
  double path_point_rectascension[3];
  path_point_rectascension[0] = rectascension;
  path_point_rectascension[1] = rectascension − rectascension_halfwidth;
  path_point_rectascension[2] = rectascension − 2.0 ∗ rectascension_halfwidth;
  switch (label_angle) {
  case 0: case 1: case 7: justification = 'l';
    break;
  case 2: case 6: justification = 'c';
    for (int i = 0; i < 3; i++) path_point_rectascension[i] += rectascension_halfwidth;
    break;
  case 3: case 4: case 5: justification = 'r';
    for (int i = 0; i < 3; i++) path_point_rectascension[i] += 2.0 ∗ rectascension_halfwidth;
    break;
  }
  double lower;
  switch (label_angle) {
  case 0: case 4: lower = −0.4;
    break;
  case 1: case 2: case 3: lower = 0.0;
    break;
  case 5: case 6: case 7: lower = −0.8;
    break;
  }
  double x[3], y[3];
  for (int i = 0; i < 3; i++)
    if (¬mytransform.polar_projection(path_point_rectascension[i], declination, x[i], y[i])) return false;
  label_color.set(OUT);
```

```
    OUT ≪ "\\FlexLabel{\\pstextpath[" ≪ justification ≪ "](0," ≪ lower ≪
        "em){\\pscurve[linestyle=none]%\n␣␣";
    for (int i = 0; i < 3; i++) OUT ≪ '(' ≪ x[i] ≪ "cm," ≪ y[i] ≪ "cm)";
    OUT ≪ "}{\\dummycolor" ≪ label ≪ "}}%\n";
    return true;
  }
```

96. As I've already said, a flex is (at the moment) excluded from the penalty algorithm. This is realised here: This routine always returns zero.

```
  double declination_flex::penalties_with(const double left, const double right, const double top, const
        double bottom, bool core) const
  {
    return 0.0;
  }
```

97. This is the drawing routine called from the main program. It goes through all flexes twice: First it only fills *objects*. Then it actually draws the flexes. The reason is efficiency: This assures that all needed labels are available before the first flex is about to be draw which may mean a reclaculation of the label dimensions. If I realised this in one pass, the needed labels would occure over and over again, and every time it would be necessary to recalculate the dimensions.

```
  void draw_flex_labels(const transformation &mytransform, const flexes_list &flexes, objects_list
        &objects, dimensions_list &dimensions)
  {
    flexes_list::const_iterator p = flexes.begin();
    while (p ≠ flexes.end()) objects.push_back(*p++);
    p = flexes.begin();
    while (p ≠ flexes.end()) (*p++)→draw(mytransform, dimensions, objects);
  }
```

98. Drawing stars. Stars are a little bit simpler than nebulae because they are mere disks. They are only included if they have a certain minimal magnitude. The disk radius is calculated according to

$$radius = \sqrt{m_{\min} - m + radius_{\min}^2}\,, \quad \text{if } m < m_{\min}\,,$$
$$radius = m_{\min}\,, \quad \text{otherwise.}$$

The star gets a label by default only if it has a certain magnitude. This is even a little bit stricter than the related condition above.

Then only the stellar colour has yet to be calculated, and it can be printed.

```
⟨ create_hs_colour( ) for star colour determination 99 ⟩
void draw_stars(const transformation &mytransform, stars_list &stars, objects_list &objects)
{
  for (int i = 0; i < stars.size( ); i++)
    if (stars[i].in_view ≠ hidden ∧ stars[i].magnitude < params.faintest_star_magnitude) {
        /* Effectively all stars of the BSC */
      if (mytransform.polar_projection(stars[i].rectascension, stars[i].declination, stars[i].x, stars[i].y)) {
        stars[i].in_view = visible;
        const double m_dot = params.faintest_star_disk_magnitude;
        const double r_min = params.minimal_star_radius/params.star_scaling;
        stars[i].radius = params.star_scaling * (stars[i].magnitude < m_dot ?
            sqrt((m_dot − stars[i].magnitude)/300.0 + r_min * r_min) : r_min);
        if (stars[i].with_label ≡ undecided) stars[i].with_label = (stars[i].magnitude <
              params.faintest_star_with_label_magnitude ∧ ¬stars[i].name.empty( )) ? visible : hidden;
        if (params.colored_stars) {
          OUT ≪ "\\newhsbcolor{starcolor}{";
          create_hs_colour(stars[i].b_v, stars[i].spectral_class);
          OUT ≪ "␣1}%\n";
        }
        else OUT ≪ params.starcolor;
        OUT ≪ "\\pscircle*[linecolor=starcolor](" ≪ stars[i].x ≪ "," ≪ stars[i].y ≪ "){" ≪
            stars[i].radius/2.54 * 72.27 ≪ "pt}%\n";
        objects.push_back(&stars[i]);
      }
      else stars[i].in_view = hidden;
    }
    else stars[i].in_view = stars[i].with_label = hidden;
}
```

99. I want to use the B−V magnitude for the colour of the star disks on the maps. Here I map the value of the B−V magnitude to a colour in the HSB space. 'HSB' – 'Hue, Saturation, Brightness' (all three fom 0 to 1). Brightness is always 1, so only hue and saturation have to be calculated.

There are three intervals for B−V with the boundaries *bv0*, *bv1*, *bv2*, and *bv3*. *bv0*–*bv1* is blue, *bv1*–*bv2* is white, and *bv2*–*bv3* is red. On each boundary, the hue values *hue0*–*hue3* respectively are valid. Inbetween I interpolate linearly (rule of three).

```
⟨ create_hs_colour( ) for star colour determination 99 ⟩ ≡
  void create_hs_colour(double b_v, string spectral_class)
  {
    double hue, saturation;
    const double bv0 = −0.1, bv1 = 0.001, bv2 = 0.62, bv3 = 1.7;
    const double hue0 = 0.6, hue1 = 0.47, hue2 = 0.17, hue3 = 0.0;
    const double min_saturation = 0.0, max_saturation = 0.2;
    ⟨ Handle missing B−V value 100 ⟩
    if (b_v < bv0) b_v = bv0;    /* cut off extreme values */
    if (b_v > bv3) b_v = bv3;
    if (b_v < bv1) {    /* blue star */
      hue = (b_v − bv0)/(bv1 − bv0) * (hue1 − hue0) + hue0;
      saturation = (b_v − bv0)/(bv1 − bv0) * (min_saturation − max_saturation) + max_saturation;
    }
    else if (b_v < bv2) {    /* white star: constantly white. */
      hue = 0.3;    /* could be anything */
      saturation = 0;
    }
    else {    /* red star */
      hue = (b_v − bv2)/(bv3 − bv2) * (hue3 − hue2) + hue2;
      saturation = (b_v − bv2)/(bv3 − bv2) * (max_saturation − min_saturation) + min_saturation;
    }
    OUT ≪ hue ≪ '␣' ≪ saturation;
  }
```

This code is used in section 98.

100. Since there are some stars in the stellar catalogue without a B−V brightness, I need a fallback on the spectral class. For such stars is $b\_v = 99$. In this routine I use the very first character in the string with the spectral class for determining an estimated value for B−V. The values are averages of all stars in the BSC with the respective spectral class.

```
⟨ Handle missing B−V value 100 ⟩ ≡
  if (b_v > 90.0) {
    switch (spectral_class[0]) {
    case 'O': b_v = 0.0; break;
    case 'B': b_v = −0.07; break;
    case 'A': b_v = 0.11; break;
    case 'F': b_v = 0.43; break;
    case 'G': b_v = 0.89; break;
    case 'K': b_v = 1.24; break;
    case 'M': b_v = 1.62; break;
    case 'N': b_v = 2.88; break;
    case 'S': b_v = 1.84; break;
    case 'C': b_v = 3.02; break;
    default: b_v = 0.0; break;
    }
  }
```

This code is used in section 99.

101. Drawing nebulae. Only nebulae with a certain minimal brightness are included, and all Messier objects, but all of these get a label by default.

The first decision I have to make here is whether the nebula has all necessary data for drawing a neat ellipsis that has the correct diameters and the correct angle. If this is not available (*horizontal_angle* = 720°), the nebula ellipsis is re-calculated so that

$$\mathit{diameter\_x}_{\text{new}} = \mathit{diameter\_y}_{\text{new}} \quad \text{and}$$
$$\mathit{diameter\_x}_{\text{new}} \cdot \mathit{diameter\_y}_{\text{new}} = \mathit{diameter\_x}_{\text{old}} \cdot \mathit{diameter\_y}_{\text{old}}$$

(make the ellipsis a circle of the same area) which means

$$\mathit{diameter\_x}_{\text{new}} := \mathit{diameter\_y}_{\text{new}} := \sqrt{\mathit{diameter\_x}_{\text{old}} \cdot \mathit{diameter\_y}_{\text{old}}}.$$

The *radius* of the nebula is a rough estimate: It is simply the half of *diameter_x*. If the nebula is too small, it is printed as a minimal circle. If it's large enough, it is printed in its (almost) full beauty, see ⟨ Draw nebula shape 102 ⟩.

```
void draw_nebulae(const transformation &mytransform, nebulae_list &nebulae, objects_list &objects)
{
  OUT << "\\psset{linecolor=nebulacolor,linewidth=" << params.linewidth_nebula <<
      "cm,linestyle=" << params.linestyle_nebula << ",curvature=1␣.5␣-1}%\n";
  for (int i = 0; i < nebulae.size(); i++)
    if (nebulae[i].in_view ≡ visible ∨ (nebulae[i].in_view ≡ undecided ∧ (((nebulae[i].kind ≡ open_cluster ∨
          nebulae[i].kind ≡ globular_cluster) ∧ nebulae[i].magnitude < params.faintest_cluster_magnitude) ∨
    ((nebulae[i].kind ≡ galaxy ∨ nebulae[i].kind ≡ reflection ∨ nebulae[i].kind ≡ emission) ∧
          nebulae[i].magnitude < params.faintest_diffuse_nebula_magnitude) ∨ nebulae[i].messier > 0))) {
      if (mytransform.polar_projection(nebulae[i].rectascension, nebulae[i].declination, nebulae[i].x,
            nebulae[i].y)) {
        nebulae[i].in_view = visible;
        if (nebulae[i].horizontal_angle > 360.0)
          nebulae[i].diameter_x = nebulae[i].diameter_y = sqrt(nebulae[i].diameter_x * nebulae[i].diameter_y);
        nebulae[i].radius = nebulae[i].diameter_x/2.0/mytransform.get_rad_per_cm() * M_PI/180.0;
        if (nebulae[i].with_label ≠ hidden) nebulae[i].with_label = visible;
        if (nebulae[i].radius > params.minimal_nebula_radius) {
          ⟨ Draw nebula shape 102 ⟩
        }
        else {
          nebulae[i].radius = params.minimal_nebula_radius;
          OUT << "\\pscircle(" << nebulae[i].x << "," << nebulae[i].y << "){" <<
              nebulae[i].radius/2.54 * 72.27 << "pt}%\n";
        }
        objects.push_back(&nebulae[i]);
      }
      else nebulae[i].in_view = hidden;
    }
    else nebulae[i].in_view = nebulae[i].with_label = hidden;
}
```

102. This is the core of *draw_nebula*( ). In order to draw the (almost) ellipsis, I define four reference points at the vertexes of the ellipsis. In the loop they are then transformed to screen coordinates and printed.

Mathematically, the algorithm used here works only for infitesimally small nebulae on the equator. The problem of "finding a point that is x degrees left from the current point with an angle of α degrees" is actually much more difficult. This is also the reason for this special case *nebulae*[*i*].*diameter_x* ≡ *nebulae*[*i*] .*diameter_y*. It shouldn't be necessary, and at the rim of the view frame it's even wrong due to the different circular scale. FixMe: Improve this. (Via rotation matrices.)

```
⟨ Draw nebula shape 102 ⟩ ≡
  if (nebulae[i].diameter_x ≡ nebulae[i].diameter_y)
    OUT ≪ "\\pscircle(" ≪ nebulae[i].x ≪ ',' ≪ nebulae[i].y ≪ "){" ≪ nebulae[i].radius ≪ "}%\n";
  else {
    double rectascension[4], declination[4];
    const double r_scale = 1.0/cos(nebulae[i].declination ∗ M_PI/180.0);
    const double cos_angle = cos(nebulae[i].horizontal_angle ∗ M_PI/180.0);
    const double sin_angle = sin(nebulae[i].horizontal_angle ∗ M_PI/180.0);
    const double half_x = nebulae[i].diameter_x/2.0;
    const double half_y = nebulae[i].diameter_y/2.0;

    rectascension[0] = nebulae[i].rectascension − half_x ∗ cos_angle/15.0 ∗ r_scale;
    declination[0] = nebulae[i].declination − half_x ∗ sin_angle;
    rectascension[1] = nebulae[i].rectascension + half_y ∗ sin_angle/15.0 ∗ r_scale;
    declination[1] = nebulae[i].declination − half_y ∗ cos_angle;
    rectascension[2] = nebulae[i].rectascension + half_x ∗ cos_angle/15.0 ∗ r_scale;
    declination[2] = nebulae[i].declination + half_x ∗ sin_angle;
    rectascension[3] = nebulae[i].rectascension − half_y ∗ sin_angle/15.0 ∗ r_scale;
    declination[3] = nebulae[i].declination + half_y ∗ cos_angle;
    OUT ≪ "\\psccurve";
    for (int j = 0; j < 4; j++) {
      double x, y;

      mytransform.polar_projection(rectascension[j], declination[j], x, y);
      OUT ≪ '(' ≪ x ≪ ',' ≪ y ≪ ')';
    }
  }
  OUT ≪ "\\relax\n";
```

This code is cited in section 101.

This code is used in section 101.

103. Grid and other curves. It's boring to have only stars on the map. I want to have the usual coordinate grid with rectascension and declination lines, plus the ecliptic and maybe the galactic equator, plus the constellation borders and constellation lines. This is done here.

104. The first routine is basic for all the following: It helps to draw a smooth curve through the given points. Actually it's a mere code fragment that is used later several times, therefore it's declared **inline** and it has unfortunately many parameters.

rectascension and *declination* are the celestial coordinates of the point. *i* is the number of the scan point in the current curve. Later it'll be the loop variable. *within_curve* is *true* if I'm actually drawing a curve, and *false* if the current segment is not visible. *steps* is the number of *i*'s that I skip over before I deploy a curve point. (I *scan* much more points than I actually *draw*, because the resulting curve is smoothed anyway so a very high point density is not necessary.)

last_x and *last_y* simply contain the values of *x* and *y* from the last call, because I need them when I step outwards of the view frame: Then I must draw a curve ending point at the *last* coordinates rather than the current ones.

steps = 1 denotes the ending point of a curve.

FixMe: There is a very basic flaw here, namely that the curve directions in the endpoints may be rather suboptimal. This is particularly annoying when drawing a circle-like shape. However, a solution is not easy; so far the only help is to make the points denser.

```
inline void add_curve_point(const double rectascension, const double declination, const transformation
        &transform, const int i, bool &within_curve, const int steps)
{
  static double last_x, last_y;
  double x, y;
  if (transform.polar_projection(rectascension, declination, x, y)) {
    if (¬within_curve) {    /* start a new one */
      OUT ≪ "\\pscurve" ≪ "(" ≪ x ≪ ',' ≪ y ≪ ")";
      within_curve = true;
    }
    else if (i % steps ≡ 0) OUT ≪ "(" ≪ x ≪ ',' ≪ y ≪ ")";
    if (i % (steps * 4) ≡ 0 ∨ steps ≡ 1) OUT ≪ "%\n";    /* line break every four coordinates */
  }
  else if (within_curve) {    /* end the current curve */
    OUT ≪ "(" ≪ last_x ≪ ',' ≪ last_y ≪ ")" ≪ "\\relax\n";
    within_curve = false;
  }
  last_x = x;
  last_y = y;
}
```

105. This is the principal grid routine that is called from the *main*( ) function. *scans_per_cm* is the density of tests whether we are within the view frame or not. By default, we scan once every millimetre. *point_distance* is given in degrees and it's the distance between two actually drawn curve points.

scans_per_fullcircle is the number of scan points in one full arc. It is used in the following code for determining the number of loop repetitions. In order to keep the meaning of all quantities, the grid creating code should respect this variable. E. g., most declination circles are smaller than one full arc. Therefore they mustn't use the full value of *scans_per_fullcircle*.

steps and *within_curve* should be clear from the routine *add_curve_point*( ). *steps* is at least 2, because it must be at least 1 anyway and "1" has a special meaning in *add_curve_point*: It denotes the end of a curve. *within_curve* must be reset to *false* if a new set of lines is to be drawn.

For a plot with closed lines (e. g. of one of the poles), you may set *point_distance* to 0 and *scans_per_cm* to a higher value, for avoiding a kink at the joint.

```
void create_grid(const transformation transform, const double scans_per_cm = 10, const double
        point_distance = 5.0)
{
  if (¬params.show_grid ∧ ¬params.show_ecliptic) return;
  const double scans_per_fullcircle = scans_per_cm / transform.get_rad_per_cm() ∗ 2.0 ∗ M_PI;
  const int steps = int((point_distance ∗ M_PI/180.0) ∗ (scans_per_fullcircle/(2.0 ∗ M_PI))) + 2;
  bool within_curve;
  if (params.show_grid) {
    OUT ≪ "\\psset{linestyle=" ≪ params.linestyle_grid ≪ ",linecolor=gridcolor,linewidth=" ≪
        params.linewidth_grid ≪ "cm}%\n";
    ⟨ Create grid lines for equal declination 106 ⟩
    ⟨ Create grid lines for equal rectascension 107 ⟩
  }
  if (params.show_ecliptic) {
    ⟨ Draw the ecliptic 108 ⟩
  }
}
```

106. As mentioned before, declination circles are smaller than the full circle of the celestial sphere. Therefore I reduce the *scans_per_fullcircle* by *cos*(*declination*) in order to decrese the number of scan points. The equator is drawn with a slightly bigger line width.

This strange construction with "*i* ≡ *number_of_points* ? 1 : *steps*" is necessary because the very last point *must* be drawn.

```
⟨ Create grid lines for equal declination 106 ⟩ ≡
  for (int declination = −80; declination ≤ 80; declination += 10) {
    if (declination ≡ 0) OUT ≪ "\\psset{linewidth=" ≪ 2.0 ∗ params.linewidth_grid ≪ "cm}%\n";
    within_curve = false;
    const int number_of_points = int(cos(declination ∗ M_PI/180.0) ∗ scans_per_fullcircle);
    for (int i = 0; i ≤ number_of_points; i++) add_curve_point(double(i)/double(number_of_points) ∗ 24.0,
          declination, transform, i, within_curve, i ≡ number_of_points ? 1 : steps);
    if (declination ≡ 0) OUT ≪ "\\psset{linewidth=" ≪ params.linewidth_grid ≪ "cm}%\n";
  }
```

This code is used in section 105.

107. The only slightly interesting thing here is that I draw the lines of equal rectascension only from $-80°$ to $+80°$ of declination, because otherwise it gets too populated in the pole regions.

```
⟨ Create grid lines for equal rectascension 107 ⟩ ≡
  const int number_of_points = int(scans_per_fullcircle/2.0);
  for (int rectascension = 0; rectascension ≤ 23; rectascension++) {
    within_curve = false;
    for (int i = 0; i ≤ number_of_points; i++)
      add_curve_point(double(rectascension), double(i)/double(number_of_points) * 160.0 − 80.0, transform, i,
          within_curve, i ≡ number_of_points ? 1 : steps);
  }
```

This code is used in section 105.

108. Unfortunately the naive approach for drawing the ecliptic,

$$\mathit{declination} = \varepsilon \cdot \sin(\mathit{rectascension})$$

(with ε being the angle between equator and ecliptic) is wrong. So I have to take the equatorial declination circle, transform it into cartesian coordinates, apply a simple rotation matrix for turning it into the ecliptical circle, and transform it back into celestial coordinates. Fortunately the ecliptic lies very symmetrically, so the resulting formulae are rather simple:

Being φ_0 the right ascension angle (= *rectascension* $\cdot$ 15°/h) of the point on the equator, its right ascension φ and declination δ after becoming the ecliptic are

$$\varphi = \arctan_2 \frac{-\sin\varphi_0 \cos\varepsilon}{\cos\varphi_0}$$
$$\delta = \arcsin(-\sin\varphi_0 \sin\varepsilon).$$

$\arctan_2$ is the quadrant-aware version of arctan, because arctan only returns values between $-\pi/2$ and $+\pi/2$.

```
⟨ Draw the ecliptic 108 ⟩ ≡
  OUT ≪ "\\psset{linestyle=" ≪ params.linestyle_ecliptic ≪ ",linecolor=eclipticcolor," ≪
      "linewidth=" ≪ params.linewidth_ecliptic ≪ "cm}%\n";
  {
    const double epsilon = 23.44 * M_PI/180.0;
    const int number_of_points = int(scans_per_fullcircle);
    within_curve = false;
    for (int i = 0; i ≤ number_of_points; i++) {
      const double phi0 = double(i)/double(number_of_points) * 2.0 * M_PI;
      const double m_sin_phi0 = −sin(phi0);
      const double phi = atan2(m_sin_phi0 * cos(epsilon), cos(phi0));
      const double delta = asin(m_sin_phi0 * sin(epsilon));
      add_curve_point(phi * 12.0/M_PI, delta * 180.0/M_PI, transform, i, within_curve,
          i ≡ number_of_points ? 1 : steps);
    }
  }
```

This code is used in section 105.

109. Constellation boundaries. They are drawn at a quite early stage so that they are overprinted by more interesting stuff.

110. This routine is a helper and is used in *draw_constellation_boundaries*( ). It draws one boundary line *b*, but only the part that is visible. For this in the first part of this routine, the container *current_line* is filled with all points of *b* that are actually visible, with their screen coordinates in centimetres.

If *current_line* consists only of a start and an end point, a '`\psline`' is created which is a straight line. I cannot get a curvature from anywhere easily in this case. Otherwise it is a '`\pscurve`' which means that we go smoothly through all points.

I need this `"[liftpen=2]"` because eventually the `\pscurve` command may be part of a `\pscustom` command and thus be part of a bigger path that forms – in terms of the Porstscript language – one united path.[2] In order to get crisp coners, the `liftpen` option is necessary.

```
void draw_boundary_line(const boundary &b, const transformation &transform, objects_list &objects, bool
        highlighted = false)
{
  vector⟨point⟩ current_line;
  for (int j = 0; j < b.points.size(); j++) {
    const double rectascension = b.points[j].x;
    const double declination = b.points[j].y;
    double x, y;
    if (¬transform.polar_projection(rectascension, declination, x, y)) continue;
    current_line.push_back(point(x, y));
  }
  if (current_line.size() ≥ 2) {
    if (current_line.size() ≡ 2) OUT ≪ "\\psline";
    else OUT ≪ "\\pscurve";
    OUT ≪ "[liftpen=2]{c-c}";
    for (int j = 0; j < current_line.size(); j++) {
      OUT ≪ '(' ≪ current_line[j].x ≪ ',' ≪ current_line[j].y ≪ ')';
      if (j % 4 ≡ 3) OUT ≪ "%\n";
      if (highlighted) ⟨Create a boundary_atom for the objects 111⟩
    }
    OUT ≪ "\\relax\n";
  }
}
```

111. This is the point where the **boundary_atom**s are actually created. As one can see, it's very simple. I just draw a virtual line from the current point in the current path to the previous one. Unfortunately I create new objects here on the heap that are never deleted. But it's harmless nevertheless.

```
⟨Create a boundary_atom for the objects 111⟩ ≡
  if (j > 0) objects.push_back(new boundary_atom(current_line[j], current_line[j − 1]));
```

This code is cited in section 60.

This code is used in section 110.

[2] This means e. g. that a dashed line pattern won't be broken at subpath junctions.

112. This is the routine that is actually called from the main program. There are two possibilities: Either the string *constellation* is empty or not. If not, the caller wants to highlight the given constellation. Then we have to undertake a two step process:

(1) Draw all boundaries that do not belong to *constellation*.
(2) Draw all boundaries that enclose *constellation* in a hightlighted style and *as one path* via `\pscustom`.

If no *constellation* is given we simply draw all lines in *boundaries* in modus (1).

```
void draw_boundaries(const transformation &mytransform, boundaries_list &boundaries, objects_list
        &objects, string constellation = string(""))
{
  OUT ≪ "\\psset{linecolor=boundarycolor,linewidth=" ≪ params.linewidth_boundary ≪ "cm," ≪
      "linestyle=" ≪ params.linestyle_boundary ≪ "}%\n";
  if (¬constellation.empty()) {
    for (int i = 0; i < boundaries.size(); i++)
      if (¬boundaries[i].belongs_to_constellation(constellation))
        draw_boundary_line(boundaries[i], mytransform, objects);
    OUT ≪ "\\psset{linecolor=hlboundarycolor,linewidth=" ≪ params.linewidth_boundary ≪
        "cm," ≪ "linestyle=" ≪ params.linestyle_hlboundary ≪ "}%\n";
    OUT ≪ "\\pscustom{";
    for (int i = 0; i < boundaries.size(); i++)
      if (boundaries[i].belongs_to_constellation(constellation))
        draw_boundary_line(boundaries[i], mytransform, objects, true);
    OUT ≪ "}%\n";
  }
  else
    for (int i = 0; i < boundaries.size(); i++) draw_boundary_line(boundaries[i], mytransform, objects);
}
```

113. Constellation lines. In the loop I test all lines available in *connections*. The first thing in the loop is to assure that at least one of stars is actually visible. At the beginning $(x1, y1)$ and $(x2, y2)$ are the screen coordinates of the two stars that are supposed to be connected by a line. Then I move from there to the respective other star by the amount of *skip*. The current length of the connection line is stored in r which must have a minimal value (in particular it must be positive), otherwise it doesn't make sense to draw the line. Finally the line is drawn from the new $(x1, y1)$ to the new $(x2, y2)$.

I expect *has_valid_coordinates*( ) to be *false* if the star is on the non-visible hemisphere (i. e. $z < 0$).

```
void draw_constellation_lines(connections_list &connections, const stars_list &stars, objects_list &objects)
{
  OUT ≪ "\\psset{linecolor=clinecolor,linestyle=" ≪ params.linestyle_cline ≪ ",linewidth=" ≪
      params.linewidth_cline ≪ "cm}%\n";
  for (int i = 0; i < connections.size(); i++)
    if ((stars[connections[i].from].in_view ≡ visible ∨ stars[connections[i].to].in_view ≡ visible) ∧
          stars[connections[i].from].has_valid_coordinates() ∧ stars[connections[i].to].has_valid_coordinates())
      {
      double x1 = stars[connections[i].from].x;
      double y1 = stars[connections[i].from].y;
      double x2 = stars[connections[i].to].x;
      double y2 = stars[connections[i].to].y;
      const double phi = atan2(y2 − y1, x2 − x1);
      double r = hypot(x2 − x1, y2 − y1);
      double skip;

      skip = stars[connections[i].from].radius + stars[connections[i].from].skip;
      r −= skip;
      x1 += skip ∗ cos(phi);
      y1 += skip ∗ sin(phi);
      skip = stars[connections[i].to].radius + stars[connections[i].to].skip;
      r −= skip;
      x2 −= skip ∗ cos(phi);
      y2 −= skip ∗ sin(phi);
      connections[i].radius = r/2.0;
      connections[i].x = (x1 + x2)/2.0;
      connections[i].y = (y1 + y2)/2.0;
      if (r > params.shortest_constellation_line ∧ r > 0.0) {
        OUT ≪ "\\psline{cc-cc}(" ≪ x1 ≪ ',' ≪ y1 ≪ ")(" ≪ x2 ≪ ',' ≪ y2 ≪ ")%\n";
        connections[i].start = point(x1, y1);
        connections[i].end = point(x2, y2);
        objects.push_back(&connections[i]);
      }
    }
}
```

114. Drawing the Milky Way. If a proper data file is available, the milky way is a simple concept, however difficult to digest for LaTeX due to many many Postscipt objects. But for this program it's so simple that I can do the reading and drawing in one small routine and I even don't need any large data structures.

See ⟨ Reading the milkyway into *pixels* 68 ⟩ for more information on *pixels*.

```
void draw_milky_way(const transformation &mytransform)
{
   vector⟨vector⟨point⟩⟩ pixels(256);
   ⟨ Reading the milkyway into pixels 68 ⟩
   for (int i = 1; i < pixels.size( ); i++) {
      if (pixels[i].size( ) ≡ 0) continue;
      interpolate_colors(double(i)/255.0, params.bgcolor, params.milkywaycolor).set(OUT);
      for (int j = 0; j < pixels[i].size( ); j++)
         OUT ≪ "\\qdisk(" ≪ pixels[i][j].x ≪ "," ≪ pixels[i][j].y ≪ "){" ≪ radius ≪ "pt}%\n";
   }
}
```

115. The main function. This consists of six parts:

(1) Command line interpretation.
(2) Definition of the desired transformation, here called *mytransform*.
(3) Definition of the containers and
(4) the filling of them by reading from text files.
(5) Creating of the LaTeX file, first and foremost by calling the drawing routines.
(6) Possibly create an EPS or PDF file by calling the necessary programs.

That's it.

```
⟨ Routines for reading the input script 11 ⟩
int main(int argc, char **argv)
{
  try {
    ⟨ Dealing with command line arguments 116 ⟩
    read_parameters_from_script(*params.in);
    if (¬params.filename_output.empty()) params.out = new ofstream(params.filename_output.c_str());
    transformation mytransform(params.center_rectascension, params.center_declination,
        params.view_frame_width, params.view_frame_height, params.grad_per_cm);
    ⟨ Definition and filling of the containers 117 ⟩
    read_objects_and_labels(*params.in, dimensions, objects, stars, nebulae, texts, flexes, mytransform);
    OUT.setf(ios::fixed);     /* otherwise LaTeX gets confused */
    OUT.precision(3);
    ⟨ Create LaTeX header 119 ⟩
    OUT ≪ "\\psclip{\\psframe[linestyle=none](0bp,0bp)(" ≪
        params.view_frame_width_in_bp() ≪ "bp," ≪ params.view_frame_height_in_bp() ≪
        "bp)}%\n" ≪ "\\psframe*[linecolor=bgcolor,linestyle=none](-1bp,-1bp)(" ≪
        params.view_frame_width_in_bp()+1 ≪ "bp," ≪ params.view_frame_height_in_bp()+1 ≪ "bp)%\n";
    ⟨ Draw all celestial objects and labels 118 ⟩
    OUT ≪ "\\endpsclip\n";
    ⟨ Create LaTeX footer 121 ⟩
    if (params.input_file) delete params.in;
    ⟨ Create EPS or PDF file if requested 122 ⟩
  }
  catch(string s)
  {
    cerr ≪ "pp3:␣" ≪ s ≪ '.' ≪ endl;
    exit(1);
  }
  return 0;
}
```

116. PP3 needs exactly one argument. It must be either the file name of the input script or a '-' which means that it takes standard input for reading the input script.

```
⟨ Dealing with command line arguments 116 ⟩ ≡
  if (argc ≡ 2) {
    if (argv[1][0] ≠ '-') {
      params.in = new ifstream(argv[1]);
      if (¬params.in→good()) throw string("Input␣script␣file␣") + argv[1] + "␣not␣found";
      else params.input_file = true;
    }
    else if (¬strcmp(argv[1], "-")) params.in = &cin;
    else cerr ≪ "Invalid␣argument:␣" ≪ argv[1] ≪ endl;
  }
  if (params.in ≡ 0) {
    cerr ≪ "PP3␣1.3␣␣Copyright␣(c)␣2003␣Torsten␣Bronger\n"
     ≪ "␣␣␣␣␣␣␣␣␣␣http://pp3.sourceforge.net\n\n"
     ≪ "Syntax:\n␣␣pp3␣{input-file}\n\n"
     ≪ "{input-file}␣may␣be␣\"-\"␣to␣denote␣standard␣input.\n"
     ≪ "You␣may␣give␣an␣empty␣file␣to␣get␣a␣default␣plot.\n"
     ≪ "The␣plot␣is␣sent␣to␣standard␣output␣by␣default.\n";
    exit(0);
  }
```

This code is used in section 115.

117. I must define all containers, but of course I only read those data structures that are actually used.

```
⟨ Definition and filling of the containers 117 ⟩ ≡
  boundaries_list boundaries;
  dimensions_list dimensions;
  objects_list objects;
  stars_list stars;
  nebulae_list nebulae;
  connections_list connections;
  texts_list texts;
  flexes_list flexes;
  read_stars(stars);      /* Must come first, because it tests whether data files can be found */
  if (params.show_boundaries) read_constellation_boundaries(boundaries);
  read_label_dimensions(dimensions);
  if (params.nebulae_visible) read_nebulae(nebulae);
  if (params.show_lines) read_constellation_lines(connections, stars);
```

This code is used in section 115.

118. Four calls here are not preceded by an **if** clause: *create_grid*( ) contains such tests of its own (because it's divided into subsections), and in *draw_flex_labels*( ), *draw_text_labels*( ), and *draw_stars*( ) every single object is tested for visibility anyway.

```
⟨ Draw all celestial objects and labels 118 ⟩ ≡
  if (params.milkyway_visible) draw_milky_way(mytransform);
  create_grid(mytransform);
  if (params.show_boundaries) draw_boundaries(mytransform, boundaries, objects, params.constellation);
  draw_flex_labels(mytransform, flexes, objects, dimensions);
  draw_text_labels(mytransform, texts, objects);
  if (params.nebulae_visible) draw_nebulae(mytransform, nebulae, objects);
  draw_stars(mytransform, stars, objects);
  if (params.show_lines) draw_constellation_lines(connections, stars, objects);
  if (params.show_labels) {
    arrange_labels(objects, dimensions);
    print_labels(objects);
  }
```

This code is used in section 115.

119. This is the preamble and the beginning of the resulting LaTeX file. I use the `geometry` package and a dvips `\special` command to set the papersize to the actual view frame plus 2 millimetres. So I create a buffer border of 1 mm thickness.

```
⟨ Create LaTeX header 119 ⟩ ≡
  create_preamble(OUT);
  OUT ≪ "\\usepackage[nohead,nofoot,margin=0cm,"
   ≪ "paperwidth=" ≪ params.view_frame_width_in_bp() ≪ "bp,"
   ≪ "paperheight=" ≪ params.view_frame_height_in_bp() ≪ "bp"
   ≪ "]{geometry}\n\n"
   ≪ "\n\\begin{document}\\parindent0pt\n"
   ≪ "\\pagestyle{empty}\\thispagestyle{empty}%\n"
   ≪ "\\special{papersize=" ≪ params.view_frame_width_in_bp() − 0.1 ≪ "bp," ≪
      params.view_frame_height_in_bp() − 0.1 ≪ "bp}%\n"
   ≪ "\\vbox␣to␣" ≪ params.view_frame_height_in_bp() ≪ "bp{"
   ≪ "\\vfill"
   ≪ "\\hbox␣to␣" ≪ params.view_frame_width_in_bp() ≪ "bp{%\n"
   ≪ params.bgcolor ≪ params.gridcolor ≪ params.eclipticcolor ≪ params.boundarycolor ≪
      params.hlboundarycolor ≪ params.starcolor ≪ params.nebulacolor ≪ params.clinecolor;
```

This code is cited in section 121.

This code is used in section 115.

120. If no preamble filename was given in the input script, a default preamble is used. If a preamble filename was given, the file should contain only font specifying commands. A possible preamble may be:

```
\usepackage{eulervm}
\usepackage[T1]{fontenc}
\renewcommand*{\rmdefault}{pmy}
```

You may also have use for a command like `\AtBeginDocument{\sffamily}` or `\boldmath` or `\large` or whatever. Or let be inspired by this fragment:

```
\usepackage{relsize}
\renewcommand*{\NGC}[1]{\smaller #1}
\renewcommand*{\IC}[1]{\smaller{\smaller IC\,}#1}
\renewcommand*{\Messier}[1]{\smaller{\smaller M\,}#1}
```

The default preamble activates Adobe Symbol Greek letters, and Helvetica Roman letters. If you define a preamble of your own, you don't have to re-define that, because it's not defined then. You can assume a naked plain LaTeX font setup.

I define a couple of LaTeX commands as hooks here, all of them take exactly one argument. "`\Label`" encloses every label. In addition to that, "`\NGC`", "`\IC`", and "`\Messier`" are built around the nebula label of the respective type, "`\Starname`" around all star labels, "`\TextLabel`" around text labels, and "`\FlexLabel`" around flex labels. "`\TicMark`" encloses all coordinate labels generated with the `tics` option. By default, they do nothing. Well, nothing to speak of.

⟨ *create_preamble*( ) for writing the LaTeX preamble 120 ⟩ ≡

```
void create_preamble(ostream &out)
{
  out << "\\documentclass[" << params.font_size << "pt]{article}\n\n"
   << "\\nofiles"
   << "\\usepackage[dvips]{color}\n"
   << "\\usepackage{pstricks,pst-text}\n"
   << "\\newcommand*{\\DP}{.}\n"
   << "\\newcommand*{\\TicMark}[1]{#1}\n"
   << "\\newcommand*{\\Label}[1]{#1}\n"
   << "\\newcommand*{\\TextLabel}[1]{#1}\n"
   << "\\newcommand*{\\FlexLabel}[1]{#1}\n"
   << "\\newcommand*{\\Starname}[1]{#1}\n"
   << "\\newcommand*{\\Messier}[1]{M\\,#1}\n"
   << "\\newcommand*{\\NGC}[1]{NGC\\,#1}\n"
   << "\\newcommand*{\\IC}[1]{IC\\,#1}\n\n";
  if (params.filename_preamble.empty())
    out << "\\usepackage{mathptmx}\n"
     << "\\usepackage{helvet}\n"
     << "\\AtBeginDocument{\\sffamily}\n";
  else out << "\\input␣" << params.filename_preamble << '\n';
}
```

This code is cited in section 19.

This code is used in section 82.

121. This is almost trivial. I just close the box structure I began at the end of ⟨ Create LaTeX header 119 ⟩ and close the document.

⟨ Create LaTeX footer 121 ⟩ ≡

```
OUT ≪ "\\hfill}}\\end{document}\n";
```

This code is used in section 115.

122. Here I call LaTeX, dvips, and/or PS2PDF in order to create the output the user wanted to have eventually in the input script.

⟨ Create EPS or PDF file if requested 122 ⟩ ≡

```
if (¬params.filename_output.empty() ∧ (params.create_eps ∨ params.create_pdf)) {
   OUT.flush();
   string commandline = string("latex␣") + params.filename_output;
   if (system(commandline.c_str()) ≡ 0) {
      string base_filename(params.filename_output);
      if (base_filename.find('.') ≠ string::npos) base_filename.erase(base_filename.find('.'));
      commandline = string("dvips␣-o␣") + base_filename + ".eps␣" + base_filename;
      if (system(commandline.c_str()) ≡ 0) {
         if (params.create_pdf) {
            commandline = string("ps2pdf␣") + "-dCompatibility=1.3␣" + base_filename + ".eps␣" +
                  base_filename + ".pdf";
            if (system(commandline.c_str()) ≠ 0) throw string("ps2pdf␣call␣failed:␣") + commandline;
         }
      }
      else throw string("dvips␣call␣failed:␣") + commandline;
   }
   else throw string("LaTeX␣call␣failed:␣") + commandline;
}
```

This code is used in section 115.

123. Index.

a: 75, 92, 93, 94.
a_unscaled: 75, 77, 78.
add: 26.
add_labels: 26.
add_curve_point: 104, 105, 106, 107, 108.
along: 35.
alpha: 77.
angle: 28, 29, 35, 36, 89, 90.
argc: 115, 116.
argv: 115, 116.
arrange_labels: 79, 82, 118.
asin: 108.
assure_valid_catalogue_index: 23, 24, 25.
atan2: 108, 113.
b: 92, 93, 94, 110.
b_v: 43, 45, 98, 99, 100.
background: 14.
base_filename: 122.
begin: 12, 88, 91, 97.
belongs_to_constellation: 54, 55, 112.
best_angle: 82.
best_penalty: 82.
bgcolor: 8, 9, 14, 114, 119.
blue: 69, 70, 71, 72, 73.
boolean: 11.
bottom: 39, 41, 42, 43, 44, 48, 49, 58, 60, 61, 62, 63, 85, 94, 96.
bottom1: 80.
bottom2: 42, 44, 49, 80.
boundaries: 14, 15, 16, 17, 18, 19.
boundaries: 57, 112, 117, 118.
boundaries_rim: 18.
boundaries_list: 54, 57, 112, 117.
boundary: 54, 55, 56, 57, 58, 110.
boundary_atom: 58, 59, 60, 111.
boundarycolor: 8, 9, 14, 119.
boundaryfile: 57.
box_height: 20.
box_width: 20.
bs: 43, 45.
bv0: 99.
bv1: 99.
bv2: 99.
bv3: 99.
c: 12, 71, 72, 89, 90, 92, 93, 94.
c_str: 19, 26, 46, 51, 57, 64, 68, 86, 88, 115, 122.
calculate_overlap: 80.
catalog: 52.
catalogue: 23, 52.
catalogue_index: 24, 25.
catalogue_name: 23, 24, 25.
ceil: 8.
center_declination: 20.
center_rectascension: 20.
center_declination: 8, 9, 20, 115.
center_rectascension: 8, 9, 20, 115.
cerr: 23, 35, 60, 63, 82, 115, 116.
ch: 45.
cin: 116.
clear: 24, 82.
clinecolor: 8, 9, 14, 119.
close: 88.
cm_per_grad: 68.
color: 13, 14.
color: 8, 39, 69, 70, 71, 72, 73, 89, 90, 92, 93, 94.
color_name: 14.
colored_stars: 17.
colored_stars: 8, 9, 17, 98.
commandline: 88, 122.
connection: 61, 63, 65.
connections: 64, 65, 113, 117, 118.
connections_list: 61, 64, 113, 117.
const_iterator: 88, 97.
constellation: 20.
constellation: 8, 9, 20, 22, 23, 43, 45, 48, 50, 54, 55, 66, 67, 82, 112, 118.
constellation_lines: 14, 15, 16, 17, 18, 19.
constellation_lines_rim: 18.
constellations: 54, 55, 56.
contents: 12, 35, 36, 37, 89, 90.
cooked_labels_file: 88.
coordinates: 37.
core: 39, 42, 43, 44, 48, 49, 58, 60, 61, 63, 94, 96.
cos: 41, 77, 78, 95, 102, 106, 108, 113.
cos_angle: 102.
cos_delta: 78.
cout: 9.
create_eps: 8, 9, 17, 122.
create_grid: 105, 118.
create_hs_colour: 98, 99.
create_pdf: 8, 9, 17, 122.
create_preamble: 88, 119, 120.
current_boundary: 57.
current_depth: 88.
current_height: 88.
current_line: 110, 111.

current_name: 88.
current_nebula: 51, 52.
current_object: 28, 30, 87.
current_star: 46.
current_width: 88.
c1: 73.
c2: 73.
d: 81, 89, 90, 92, 93, 94.
`dashed`: 16.
`DBL_EPSILON`: 78.
`DBL_MAX`: 39, 82.
`DEBUG`: 82, 85.
decimal point: 120.
declination: 35, 36, 37, 43, 45, 48, 50, 68, 75, 77, 78, 89, 90, 91, 92, 93, 95, 98, 101, 102, 104, 106, 108, 110.
`declination`: 35.
declination_flex: 36, 92, 94, 95, 96.
`delete`: 26.
`delete_labels`: 26.
delta: 77, 78, 108.
denominator: 62.
depth: 81, 86, 87, 88.
diameter_x: 48, 50, 101, 102.
diameter_y: 48, 50, 101, 102.
dimension: 38, 81.
dimensions: 26, 82, 86, 87, 88, 92, 94, 95, 97, 115, 117, 118.
dimensions_list: 26, 81, 82, 86, 88, 92, 94, 95, 97, 117.
dimensions_recalculated: 87.
distance: 83, 84.
`dotted`: 16.
`\DP`: 120.
draw: 92, 94, 95, 97.
*draw_*: 91.
draw_boundaries: 112, 118.
draw_boundary_line: 110, 112.
draw_constellation_boundaries: 110.
draw_constellation_lines: 61, 113, 118.
draw_flex_labels: 97, 118.
draw_milky_way: 114, 118.
draw_nebula: 102.
draw_nebulae: 101, 118.
draw_stars: 98, 118.
draw_text_labels: 91, 118.
dummy: 64, 86.
`ecliptic`: 14, 15, 16, 17.
eclipticcolor: 8, 9, 14, 119.
emission: 47, 101.
empty: 7, 19, 26, 60, 63, 72, 82, 84, 88, 98, 112, 115, 120, 122.
end: 23, 58, 59, 60, 61, 62, 63, 87, 88, 91, 95, 97, 113.
endl: 60, 63, 82, 115, 116.
`eps_output`: 17.
epsilon: 108.
erase: 122.
error_message: 24, 65.
exit: 115, 116.
`faintest_cluster_magnitude`: 20.
`faintest_diffuse_nebula_magnitude`: 20.
`faintest_star_disk_magnitude`: 20.
`faintest_star_magnitude`: 20.
`faintest_star_with_label_magnitude`: 20.
faintest_cluster_magnitude: 8, 9, 20, 101.
faintest_diffuse_nebula_magnitude: 8, 9, 20, 101.
faintest_star_disk_magnitude: 8, 9, 20, 98.
faintest_star_magnitude: 8, 9, 20, 98.
faintest_star_with_label_magnitude: 8, 9, 20, 98.
false: 9, 11, 23, 26, 28, 31, 35, 39, 44, 49, 55, 62, 78, 82, 87, 95, 104, 105, 106, 107, 108, 110, 113.
fdim: 42, 44, 49, 80.
file: 64, 68, 86.
`filename`: 13.
filename: 46, 51.
filename_boundaries: 8, 9, 19, 57.
filename_dimensions: 8, 9, 19, 86.
filename_include: 8, 9, 19, 26.
filename_lines: 8, 9, 19, 64.
filename_milkyway: 8, 9, 19, 68.
filename_nebulae: 8, 9, 19, 51.
filename_output: 8, 9, 19, 88, 115, 122.
filename_preamble: 8, 9, 19, 120.
filename_prefix: 7, 9.
filename_stars: 8, 9, 19, 46.
find: 23, 37, 87, 95, 122.
fixed: 37, 88, 115.
flamsteed: 22, 23, 24, 25, 28, 30, 31, 32, 33, 34, 43, 45, 66, 67.
flex_label: 92, 93, 94.
flexes: 26, 36, 97, 115, 117, 118.
flexes_list: 26, 92, 93, 97, 117.
`\FlexLabel`: 120.
floor: 35, 37.
flush: 122.
fmax: 39, 42, 44, 49, 80.
fmin: 42, 44, 49, 80.
font_size: 8, 9, 20, 120.

`fontsize`: 20.
found: 23.
found_nebulae: 24, 26, 31, 32, 33, 34.
found_stars: 24, 26, 31, 32, 33, 34.
from: 61, 113.
from_catalogue_index: 64, 65, 66.
from_catalogue_name: 64, 65, 66.
from_index: 65, 66, 67.
galaxy: 47, 101.
get: 12, 45.
get_label_boundaries: 39, 41, 42, 82, 85.
get_rad_per_cm: 68, 75, 95, 101, 105.
getenv: 7.
getline: 13, 26, 27, 45, 64, 86.
globular_cluster: 47, 101.
good: 116.
`grad_per_cm`: 20.
grad_per_cm: 8, 9, 20, 75, 76, 77, 115.
green: 69, 70, 71, 72, 73.
`grid`: 14, 15, 16, 17.
gridcolor: 8, 9, 14, 119.
half_x: 102.
half_y: 102.
has_valid_coordinates: 39, 113.
hd: 22, 43, 45, 66, 67.
height: 75, 77, 78, 81, 86, 87, 88, 95.
henry_draper: 22, 23, 24, 25, 28, 30, 31, 32, 33, 34.
hidden: 31, 34, 39, 59, 61, 82, 93, 98, 101.
highlighted: 110.
`highlighted_boundaries`: 14, 15, 16.
hlboundarycolor: 8, 9, 14, 119.
hooks in preamble: 120.
horizontal_angle: 48, 50, 101, 102.
hue: 99.
hue0: 99.
hue1: 99.
hue2: 99.
hue3: 99.
hypot: 59, 60, 63, 83, 113.
i: 22, 31, 32, 33, 34, 55, 56, 65, 77, 82, 85, 87, 88, 95, 98, 101, 104, 106, 107, 108, 112, 113, 114.
ic: 22, 24, 25, 28, 30, 31, 32, 33, 34, 48, 50, 52.
`\IC`: 120.
identify_object: 25, 28, 30.
ifstream: 19, 26, 46, 51, 57, 64, 68, 86, 88, 116.
in: 8, 9, 45, 50, 56, 71, 115, 116.
in_view: 33, 34, 39, 59, 61, 82, 83, 85, 91, 93, 98, 101, 113.
`include`: 19.
included: 26.
included_file: 26, 27.
included_script: 19.
index: 23.
index_list: 21, 22, 23, 24, 25.
input_file: 8, 9, 115, 116.
insert: 12.
interpolate_colors: 73, 114.
intersection: 60, 62, 63.
intersection_points: 60, 63.
ios: 37, 88, 115.
isspace: 12.
istream: 8, 11, 12, 13, 24, 25, 26, 45, 50, 56, 71.
iterator: 91.
j: 77, 82, 83, 102, 110, 114.
justification: 95.
k: 82.
kind: 48, 50, 101.
kind_flex_declination: 35, 36.
kind_text_label: 35, 36.
label: 30, 39, 46, 52, 82, 84, 85, 87, 88, 90, 93, 95.
`label_dimensions`: 19.
`label_skip`: 20.
`\Label`: 120.
label_angle: 28, 39, 41, 79, 82, 90, 93, 95.
label_arranged: 28, 31, 39, 42, 44, 49, 82, 85, 90, 93.
label_color: 39, 85, 90, 93, 95.
label_depth: 39, 41, 85, 87.
label_height: 39, 41, 82, 87, 95.
label_kind: 35, 36.
label_skip: 8, 9, 20, 39.
label_width: 39, 41, 82, 87, 95.
labelcolor: 8, 9, 14, 39.
`labels`: 14, 17, 18.
lable_angle: 39.
lambda: 62.
last_x: 104.
last_y: 104.
`latex_preamble`: 19.
left: 39, 41, 42, 43, 44, 48, 49, 58, 60, 61, 62, 63, 85, 94, 96.
left1: 80.
left2: 42, 44, 49, 80.
length: 29, 60, 63, 88.
`line_style`: 13, 16.
`line_width`: 13, 15.
line_intersection: 60, 62, 63.
line_name: 15, 16.
lines: 61.

linestyle_boundary: 8, 9, 16, 112.
linestyle_cline: 8, 9, 16, 113.
linestyle_ecliptic: 8, 9, 16, 108.
linestyle_grid: 8, 9, 16, 105.
linestyle_hlboundary: 8, 9, 16, 112.
linestyle_nebula: 8, 9, 16, 101.
linewidth_boundary: 8, 9, 15, 112.
linewidth_cline: 8, 9, 15, 113.
linewidth_ecliptic: 8, 9, 15, 108.
linewidth_grid: 8, 9, 15, 105, 106.
linewidth_hlboundary: 8, 9, 15.
linewidth_nebula: 8, 9, 15, 101.
list: 88, 89, 93.
lower: 95.
m_dot: 98.
`M_PI`: 68, 77, 78, 95, 101, 102, 105, 106, 108.
`M_PI_2`: 59, 77.
`M_PI_4`: 41.
m_sin_phi0: 108.
magnitude: 43, 45, 48, 50, 98, 101.
main: 105, 115.
map: 21, 22, 23, 24, 25, 81.
max: 62.
max_ic: 22.
max_messier: 22.
max_ngc: 22.
max_saturation: 99.
messier: 22, 24, 25, 28, 30, 31, 32, 33, 34, 48, 50, 52, 101.
`\Messier`: 120.
`milky_way`: 14, 17, 19.
milkyway_visible: 8, 9, 17, 118.
milkywaycolor: 8, 9, 14, 114.
min: 62.
min_saturation: 99.
`minimal_nebula_radius`: 20.
`minimal_star_radius`: 20.
minimal_nebula_radius: 8, 9, 20, 101.
minimal_star_radius: 8, 9, 20, 98.
mytransform: 26, 68, 91, 92, 94, 95, 97, 98, 101, 102, 112, 114, 115, 118.
n: 50.
name: 43, 45, 46, 69, 72, 86, 98.
nebula: 48, 49, 50, 51, 74.
nebula_kind: 47, 48, 50.
nebulacolor: 8, 9, 14, 119.
nebulae: 22, 25, 26, 28, 30, 31, 32, 33, 34, 51, 101, 102, 115, 117, 118.
`nebulae`: 14, 15, 16, 17, 18, 19.
nebulae_file: 51.
nebulae_list: 25, 26, 48, 51, 101, 117.
nebulae_visible: 8, 9, 17, 24, 117, 118.
new_name: 73.
ngc: 22, 24, 25, 28, 30, 31, 32, 33, 34, 48, 50, 52.
`\NGC`: 120.
npos: 37, 122.
number: 52.
number_of_points: 106, 107, 108.
numerator: 62.
object_name: 19.
objects: 26, 74, 82, 83, 84, 85, 87, 88, 91, 92, 94, 95, 97, 98, 101, 110, 111, 112, 113, 115, 117, 118.
objects_list: 26, 40, 82, 85, 88, 91, 92, 94, 95, 97, 98, 101, 110, 112, 113, 117.
`off`: 17.
ofstream: 88, 115.
`on`: 17.
on_baseline: 28, 29, 35, 36, 39, 41, 89, 90, 93.
on_object_bottom: 82.
on_object_left: 82.
on_object_right: 82.
on_object_scope: 83.
on_object_top: 82.
opcode: 13, 14, 15, 16, 17, 18, 19, 20, 26, 28, 30, 31, 32, 33, 34, 35.
open_cluster: 47, 101.
origin_x: 41.
origin_y: 41.
ostream: 8, 69, 70, 72, 120.
out: 6, 8, 9, 69, 70, 72, 115, 120.
`OUT`: 6, 85, 95, 98, 99, 101, 102, 104, 105, 106, 108, 110, 112, 113, 114, 115, 119, 121, 122.
`output`: 19.
overlap_bottom: 42, 44, 49, 80.
overlap_left: 42, 44, 49, 80.
overlap_right: 42, 44, 49, 80.
overlap_top: 42, 44, 49, 80.
overlap_x: 42, 44, 49, 80.
overlap_y: 42, 44, 49, 80.
p: 56.
param_name: 20.
parameters: 8.
params: 6, 8, 14, 15, 16, 17, 18, 19, 20, 24, 26, 35, 36, 39, 42, 44, 46, 49, 51, 57, 60, 63, 64, 68, 82, 85, 86, 88, 98, 101, 105, 106, 108, 112, 113, 114, 115, 116, 117, 118, 119, 120, 122.
path_point_rectascension: 95.
`pdf_output`: 17.

penalties: 44, 49.
`penalties`: 13, 18.
penalties_boundary: 8, 9, 18, 60.
penalties_boundary_rim: 8, 9, 18, 60.
penalties_cline: 8, 9, 18, 63.
penalties_cline_rim: 8, 9, 18, 63.
penalties_label: 8, 9, 18, 42.
penalties_nebula: 8, 9, 18, 49.
penalties_rim: 8, 9, 18, 82.
penalties_star: 8, 9, 18, 44.
penalties_threshold: 8, 9, 18, 82.
penalties_with: 39, 40, 42, 43, 44, 48, 49, 58, 60, 61, 63, 82, 94, 96.
penalty: 79, 82.
penalty_name: 18.
phi: 77, 78, 108, 113.
phi0: 108.
pixel: 68.
pixels: 68, 114.
point: 53, 54, 58, 59, 60, 61, 63, 68, 110, 113, 114.
point_distance: 105.
points: 54, 56, 110.
polar_projection: 68, 75, 77, 78, 91, 95, 98, 101, 102, 104, 110.
position: 28, 29, 35.
`PP3DATA`: 7.
pp3data: 7.
pp3data_env: 7.
pp3data_env_raw: 7.
preamble (LaTeX): 120.
precision: 82, 88, 115.
print_labels: 79, 85, 91, 93, 118.
push_back: 24, 36, 46, 51, 57, 60, 63, 65, 68, 74, 83, 88, 91, 97, 98, 101, 110, 111, 113.
r: 60, 63, 89, 90, 92, 93, 94, 113.
r_min: 98.
r_scale: 102.
rad_per_cm: 75, 77.
radius: 39, 41, 44, 49, 59, 68, 90, 98, 101, 102, 113, 114.
raw_labels_file: 88.
*read_*: 1.
read_boolean: 11, 17.
read_constellation_boundaries: 57, 117.
read_constellation_lines: 64, 117.
read_label_dimensions: 86, 117.
read_nebulae: 51, 117.
read_objects_and_labels: 26, 115.
read_parameters_from_script: 13, 19, 115.
read_stars: 46, 117.
read_string: 12, 19, 20, 30, 35.
recalculate_dimensions: 87, 88, 95.
rectascension: 35, 36, 37, 43, 45, 48, 50, 68, 75, 77, 78, 89, 90, 91, 92, 93, 95, 98, 101, 102, 104, 107, 108, 110.
rectascension_halfwidth: 95.
red: 69, 70, 71, 72, 73.
reflection: 47, 101.
replace: 37.
`reposition`: 26.
required_names: 88.
resize: 56.
rest_of_line: 13, 26.
right: 39, 41, 42, 43, 44, 48, 49, 58, 60, 61, 62, 63, 85, 94, 96.
right1: 80.
right2: 42, 44, 49, 80.
`rim`: 18.
rim_width: 82.
s: 45, 82, 92, 93, 94, 115.
saturation: 99.
scans_per_cm: 105.
scans_per_fullcircle: 105, 106, 107, 108.
scope: 39, 83.
script: 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 28, 30, 31, 32, 33, 34, 35.
search_objects: 24, 31, 32, 33, 34.
semicolon: 28.
`set`: 13, 20.
set: 69, 70, 85, 95, 114.
`set_label_text`: 26.
setf: 37, 88, 115.
`shortest_constellation_line`: 20.
shortest_constellation_line: 8, 9, 20, 113.
show_boundaries: 8, 9, 17, 117, 118.
show_ecliptic: 8, 9, 17, 105.
show_grid: 8, 9, 17, 105.
show_labels: 8, 9, 17, 118.
show_lines: 8, 9, 17, 117, 118.
sin: 41, 77, 78, 102, 108, 113.
sin_angle: 102.
size: 22, 31, 32, 33, 34, 55, 56, 60, 63, 65, 82, 83, 85, 87, 88, 98, 101, 110, 112, 113, 114.
skip: 18, 39, 41, 59, 61, 82, 84, 90, 113.
slope: 62.
`solid`: 16.
spectral_class: 43, 45, 98, 99, 100.
sqrt: 98, 101.

star: 43, 44, 45, 46, 74, 82.
`star_scaling`: 20.
star_scaling: 8, 9, 20, 98.
starcolor: 8, 9, 14, 98, 119.
`\Starname`: 120.
stars: 22, 25, 26, 28, 30, 31, 32, 33, 34, 46, 61, 64, 65, 66, 67, 74, 98, 113, 115, 117, 118.
`stars`: 14, 18, 19.
stars_file: 46.
stars_list: 25, 26, 44, 46, 64, 98, 113, 117.
start: 35, 58, 59, 60, 61, 62, 63, 113.
start_y: 62.
std: 5.
step_declination: 35.
step_rectascension: 35.
steps: 104, 105, 106, 107, 108.
str: 24, 37, 52, 65, 82.
strcmp: 116.
stretch: 76, 78.
stretch_factor: 75, 76, 78.
string: 7, 8, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 22, 23, 24, 25, 26, 27, 28, 29, 35, 37, 39, 43, 46, 48, 52, 54, 55, 64, 69, 71, 72, 73, 81, 86, 87, 88, 89, 90, 92, 93, 94, 95, 99, 112, 115, 116, 122.
stringstream: 24, 37, 52, 65, 82.
strtod: 88.
substr: 55, 88.
`switch`: 13, 17.
switch_name: 17.
system: 88, 122.
t: 89, 90.
temp_file: 88.
`text`: 26, 35.
text: 36, 89, 90, 91.
`text_labels`: 14.
`\TextLabel`: 120.
textlabelcolor: 8, 9, 14, 35, 36.
texts: 26, 36, 91, 115, 117, 118.
texts_list: 26, 89, 91, 117.
`threshold`: 18.
`\TicMark`: 120.
`tics`: 35.
to: 61, 113.
to_catalogue_index: 64, 65, 67.
to_catalogue_name: 64, 65, 67.
to_index: 65, 66, 67.
token: 27, 35.
top: 39, 41, 42, 43, 44, 48, 49, 58, 60, 61, 62, 63, 85, 94, 96.
top1: 80.
top2: 42, 44, 49, 80.
transform: 104, 105, 106, 107, 108, 110.
transformation: 26, 75, 76, 77, 78, 91, 92, 94, 95, 97, 98, 101, 104, 105, 110, 112, 114, 115.
true: 9, 11, 23, 26, 28, 31, 39, 43, 48, 55, 58, 61, 78, 82, 87, 90, 93, 94, 95, 104, 112, 116.
undecided: 39, 98, 101.
UnexpectedEOS: 12.
unique: 88.
unknown: 47, 48.
user_pattern: 35, 37.
value: 18.
vector: 24, 26, 38, 40, 44, 48, 54, 60, 61, 63, 68, 83, 110, 114.
vectors: 21.
vicinity: 82, 83.
view_data: 25, 28, 30, 39, 40, 41, 42, 43, 44, 48, 49, 58, 61, 74, 87, 89, 90, 92.
view_frame_height: 8, 9, 20, 82, 85, 115.
view_frame_height_in_bp: 8, 115, 119.
view_frame_width: 8, 9, 20, 82, 85, 115.
view_frame_width_in_bp: 8, 115, 119.
visibility: 39.
visible: 28, 32, 33, 39, 42, 59, 61, 82, 83, 84, 85, 87, 90, 91, 93, 98, 101, 113.
width: 75, 77, 78, 81, 86, 87, 88, 95.
with_label: 28, 31, 32, 39, 42, 59, 61, 82, 84, 85, 87, 90, 93, 98, 101.
within_curve: 104, 105, 106, 107, 108.
x: 39, 53, 68, 73, 75, 78, 91, 95, 102, 104, 110.
x0: 78.
x1: 78, 113.
x2: 113.
y: 39, 53, 68, 73, 75, 78, 91, 95, 102, 104, 110.
y0: 78.
y1: 78, 113.
y2: 113.
z: 75, 76.
zero_point: 62.
$\tilde{z}$: 76.
z0: 78.
z1: 78.

⟨ Celestial object activation 33 ⟩ Cited in section 34. Used in section 26.
⟨ Celestial object deletion 34 ⟩ Used in section 26.
⟨ Change label text 30 ⟩ Used in section 26.
⟨ Condition to exclude double stars and such 84 ⟩ Used in section 83.
⟨ Create EPS or PDF file if requested 122 ⟩ Used in section 115.
⟨ Create LaTeX footer 121 ⟩ Used in section 115.
⟨ Create LaTeX header 119 ⟩ Cited in section 121. Used in section 115.
⟨ Create a **boundary_atom** for the *objects* 111 ⟩ Cited in section 60. Used in section 110.
⟨ Create grid lines for equal declination 106 ⟩ Used in section 105.
⟨ Create grid lines for equal rectascension 107 ⟩ Used in section 105.
⟨ Create mapping structures for direct catalogue access 22 ⟩ Used in section 26.
⟨ Create nebula label 52 ⟩ Used in section 51.
⟨ Create one connection 65 ⟩ Used in section 64.
⟨ Dealing with command line arguments 116 ⟩ Used in section 115.
⟨ Default values of all global parameters 9 ⟩ Used in section 8.
⟨ Define **color** data structure 69 ⟩ Used in section 8.
⟨ Definition and filling of the containers 117 ⟩ Used in section 115.
⟨ Definition of *line_intersection*( ) for intersection of two lines 62 ⟩ Cited in section 60. Used in section 60.
⟨ Draw all celestial objects and labels 118 ⟩ Used in section 115.
⟨ Draw nebula shape 102 ⟩ Cited in section 101. Used in section 101.
⟨ Draw the ecliptic 108 ⟩ Used in section 105.
⟨ Find objects in vicinity of *objects*[*i*] 83 ⟩ Cited in section 39. Used in section 82.
⟨ Generate *contents* from current coordinates 37 ⟩ Cited in section 35. Used in section 35.
⟨ Handle missing B−V value 100 ⟩ Used in section 99.
⟨ Helping routines for nebulae labels 88 ⟩ Used in section 82.
⟨ Insert text label into label container structure 36 ⟩ Used in section 35.
⟨ Label activation 32 ⟩ Used in section 26.
⟨ Label deletion 31 ⟩ Used in section 26.
⟨ Label repositioning 28 ⟩ Used in section 26.
⟨ Map a wind rose *position* to an *angle* and determine *on_baseline* 29 ⟩ Used in sections 28 and 35.
⟨ Reading the milkyway into *pixels* 68 ⟩ Cited in section 114. Used in section 114.
⟨ Routines for reading the input script 11, 12, 13, 21, 23, 24, 25, 26 ⟩ Used in section 115.
⟨ Set color parameters 14 ⟩ Used in section 13.
⟨ Set filename parameters 19 ⟩ Used in section 13.
⟨ Set label dimensions 87 ⟩ Used in section 82.
⟨ Set line styles 16 ⟩ Used in section 13.
⟨ Set line widths 15 ⟩ Used in section 13.
⟨ Set on/off parameters 17 ⟩ Used in section 13.
⟨ Set penalty parameters 18 ⟩ Used in section 13.
⟨ Set single value parameters 20 ⟩ Used in section 13.
⟨ Skip everything till `"objects_and_labels"` 27 ⟩ Used in section 26.
⟨ Test whether *stars*[*i*] is the 'from' star 66 ⟩ Cited in section 67. Used in section 65.
⟨ Test whether *stars*[*i*] is the 'to' star 67 ⟩ Used in section 65.
⟨ Text labels 35 ⟩ Used in section 26.
⟨ *create_hs_colour*( ) for star colour determination 99 ⟩ Used in section 98.
⟨ *create_preamble*( ) for writing the LaTeX preamble 120 ⟩ Cited in section 19. Used in section 82.

The Sky Map Creator **PP3**

(Version 1.3)

	Section	Page
Introduction	1	1
Global parameters and default values	6	2
The input script	10	6
Part I: Global parameters	11	6
Part II: Change printed objects and labels	21	14
Data structures and file formats	38	25
View data: Positioning and labels	39	26
Stars	43	28
Nebulae	47	30
Constellation boundaries	53	32
Constellation lines	61	36
The Milky Way	68	41
Colour	69	42
Drawing the chart	74	44
Coordinate transformations	75	44
Label organisation	79	47
Determining label dimensions	87	52
User labels	89	54
Flex labels	92	55
Drawing stars	98	59
Drawing nebulae	101	62
Grid and other curves	103	63
Constellation boundaries	109	66
Constellation lines	113	69
Drawing the Milky Way	114	70
The main function	115	71
Index	123	76