The pgfmolbio package – Molecular Biology Graphs with Ti*k*Z*

Wolfgang Skala†

http://www.ctan.org/pkg/pgfmolbio

2011/09/20

The experimental package pgfmolbio draws graphs typically found in molecular biology texts. Currently, the package contains one module, which creates DNA sequencing chromatograms from files in standard chromatogram format (scf). Since scf files are binary, pgfmolbio relies on the LuaTEX engine for converting information from these files into Ti*k*Z drawing commands.

*This document describes version v0.1, dated 2011/09/20.

†Division of Structural Biology, Department of Molecular Biology, University of Salzburg, Austria;
Wolfgang.Skala@stud.sbg.ac.at

Contents

1 Introduction

1.1 About pgfmolbio

Over the decades, TEX has gained popularity across a large number of disciplines. Although originally designed as a mere typesetting system, packages such as `pgf`[1] and `pstricks`[2] have strongly extended its *drawing* abilities. Thus, one can create complicated charts that perfectly integrate with the text.

Texts on molecular biology include a range of special graphs, e. g. multiple sequence alignments, membrane protein topologies, DNA sequencing chromatograms, plasmid maps, protein domain diagrams and others. The `texshade`[3] and `textopo`[4] packages cover alignments and topologies, respectively, but packages dedicated to the remaining graphs are absent. Admittedly, one may create those images with various external programs and then include them in the TEX document. Nevertheless, purists (like the author of this document) might prefer a TEX-based approach.

The `pgfmolbio` package aims at becoming such a purist solution. In its first development release, `pgfmolbio` is able to read DNA sequencing files in standard chromatogram format (`.scf`) and draw the corresponding chromatogram using routines from `pgf`'s Ti*k*Z frontend. In order to convert the data from the `scf` input file to an image, `pgfmolbio` relies on the Lua scripting language implemented in LuaTEX. Consequently, the package will not work with traditional engines like pdfTEX.

Since this is a development release, `pgfmolbio` presumably includes a number of bugs, and its commands and features are likely to change in future versions. Moreover, the current version is far from complete, but since time is scarce, I am unable to predict when (and if) additional functions become available. Nevertheless, I would greatly appreciate any comments or suggestions.

[1] Tantau, T. (2010). The Ti*k*Z and PGF packages. `http://ctan.org/tex-archive/graphics/pgf/`.

[2] van Zandt, T., Niepraschk, R., and Voß, H. (2007). PSTricks: PostScript macros for Generic TEX. `http://ctan.org/tex-archive/graphics/pstricks`.

[3] Beitz, E. (2000). TEXshade: shading and labeling multiple sequence alignments using LATEX 2_ε. *Bioinformatics* **16**(2), 135–139. `http://ctan.org/tex-archive/macros/latex/contrib/texshade`.

[4] Beitz, E. (2000). TEXtopo: shaded membrane protein topology plots in LATEX 2_ε. *Bioinformatics* **16**(11), 1050–1051. `http://ctan.org/tex-archive/macros/latex/contrib/textopo`.

1.2 Getting Started

Before you consider using `pgfmolbio`, please make sure that both your LuaTEX (at least 0.70.1) and `pgf` (at least 2.10) installations are up-to-date. Once your TEX system meets these requirements, just load `pgfmolbio` as usual, i. e. by

`\usepackage[`⟨*module*⟩`]{pgfmolbio}`

The package is divided into *modules*, each of which produces a certain type of graph. Currently, only one ⟨*module*⟩ is available: chromatogram allows you to draw DNA sequencing chromatograms as obtained by the Sanger sequencing method. Thus, the only sensible way of including the package is currently `\usepackage[chromatogram]{pgfmolbio}`.

chromatogram

`\pgfmolbioset[`⟨*module*⟩`]{`⟨*key-value list*⟩`}`

The ⟨*key-value list*⟩ in the mandatory argument of this command allows you to fine-tune the graphs produced by each ⟨*module*⟩ of `pgfmolbio`. The possible keys are described in the sections on the respective modules.

2 The chromatogram module

2.1 Overview

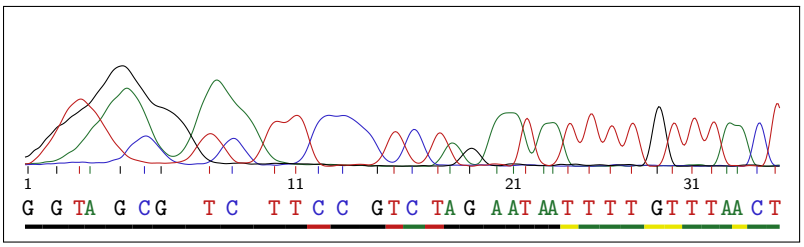
The chromatogram module draws DNA sequencing chromatograms stored in standard chromatogram format (`scf`), which was developed by Simon Dear and Rodger Staden[1]. The documentation for the Staden package[2] describes the current version of the `scf` format in detail. As far as they are crucial to understanding the Lua code, we will discuss some details of this file format in the documented source code (section 3.3). Note that `pgfmolbio` only supports `scf` version 3.00.

2.2 Drawing Chromatograms

`\pmbchromatogram[`⟨*key-value list*⟩`]{`⟨*scf file*⟩`}`

The chromatogram module defines a single command, which reads a chromatogram from an ⟨*scf file*⟩ and draws it with routines from Ti*k*Z (Example 2.1). The options, which are set in the ⟨*key-value list*⟩, configure the appearance of the chromatogram. The following sections will elaborate on the available keys.

Example 2.1




```
\begin{tikzpicture} % optional
  \pmbchromatogram{SampleScf.scf}
\end{tikzpicture} % optional
```

[1] Dear, S. and Staden, R. (1992). A standard file format for data from DNA sequencing instruments. *DNA Seq.* **3**(2), 107–110.

[2] `http://staden.sourceforge.net/`

Although you will often put \pmbchromatogram into a tikzpicture environment, you may actually use the macro on its own. pgfmolbio checks whether the command is surrounded by a tikzpicture and adds this environment if necessary.

2.3 Displaying Parts of the Chromatogram

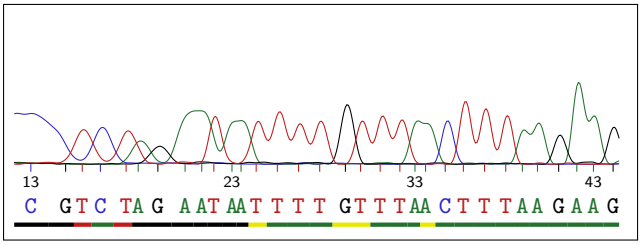
sample range =⟨*lower*⟩ to ⟨*upper*⟩[step ⟨*interval*⟩]

Default: 1 to 500 step 1

sample range selects the part of the chromatogram which pgfmolbio should display. The value for this key consists of two or three parts, separated by the keywords to and step. The package will draw the chromatogram data between the ⟨*lower*⟩ and ⟨*upper*⟩ boundary. There are two ways of specifying these limits:

1. If you enter a number, pgfmolbio includes the data from the ⟨*lower*⟩ to the ⟨*upper*⟩ sample point (Example 2.2). A *sample point* represents one measurement of the fluorescence signal along the time axis, where the first sample point has index 1. One peak comprises about 20 sample points.

Example 2.2

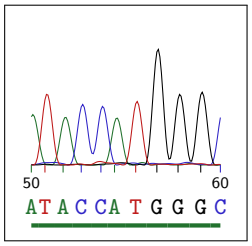



```
\pmbchromatogram[sample range=200 to 600]{SampleScf.scf}
```

2. If you enter the keyword base followed by an optional space and a number, the chromatogram starts or stops at the peak corresponding to the respective base. The first detected base peak has index 1. Compare Examples 2.2 and 2.3 to see the difference.

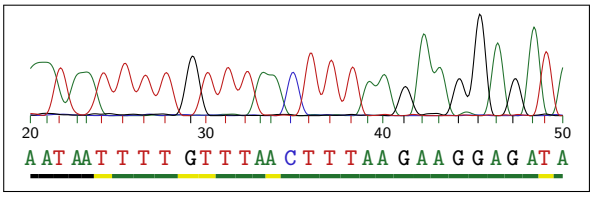
The optional third part of the value for sample range orders the package to draw every ⟨*interval*⟩th sample point. If your document contains large chromatograms or a great number of them, drawing fewer sample points increases typesetting time at the cost of image quality (Example 2.4). Nevertheless, the key may be especially useful while optimizing the layout of complex chromatograms.

Example 2.3

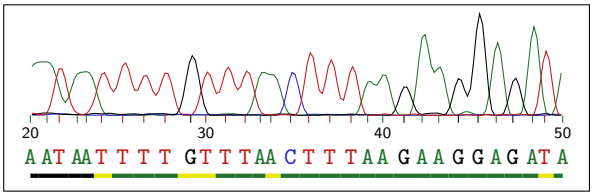



```
\pmbchromatogram[%
   sample range=base 50 to base60
 ]{SampleScf.scf}
```

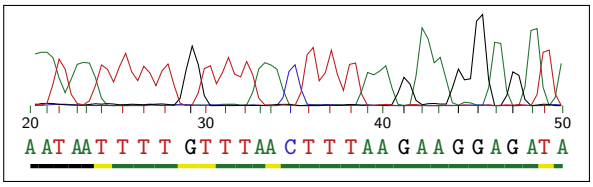
Example 2.4




```
\pmbchromatogram[%
   sample range=base 20 to base 50 step 1
 ]{SampleScf.scf}
```




```
\pmbchromatogram[%
   sample range=base 20 to base 50 step 2
 ]{SampleScf.scf}
```




```
\pmbchromatogram[%
   sample range=base 20 to base 50 step 4
 ]{SampleScf.scf}
```

2.4 General Layout

`x unit` =⟨*dimension*⟩

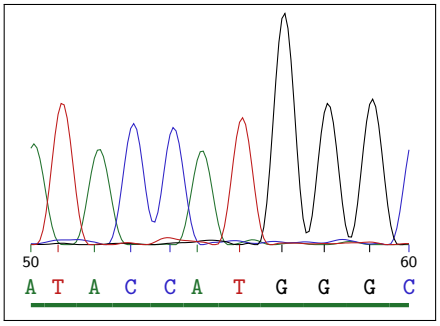
Default: `0.2mm`

`y unit` =⟨*dimension*⟩

Default: `0.01mm`

These keys set the horizontal distance between two consecutive sample points and the vertical distance between two fluorescence intensity values, respectively. Example 2.5 illustrates how you can enlarge a chromatogram twofold by doubling these values.

Example 2.5




```
\pmbchromatogram[%
   sample range=base 50 to base 60,
   x unit=0.4mm,
   y unit=0.02mm
 ]{SampleScf.scf}
```

`samples per line` =⟨*number*⟩

Default: `500`

`baseline skip` =⟨*dimension*⟩

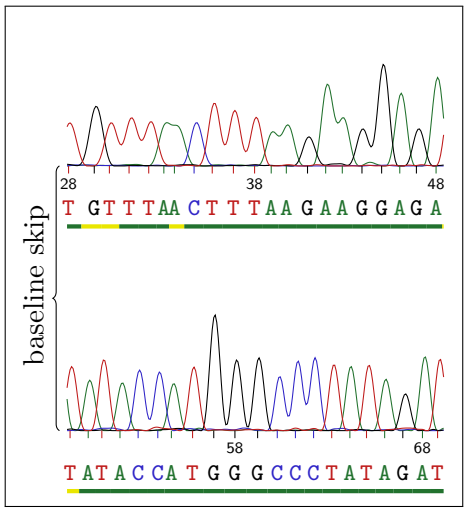
Default: `3cm`

A new chromatogram "line" starts after ⟨*number*⟩ sample points, and the baselines of adjacent lines (i. e., the y-value of fluorescence signals with zero intensity) are separated by ⟨*dimension*⟩. In Example 2.6, you see two lines, each of which contains 250 of the 500 sample points drawn. Furthermore, the baselines are 3.5 cm apart.

`canvas style` =⟨*style*⟩

Default: `draw=none, fill=none`

Example 2.6



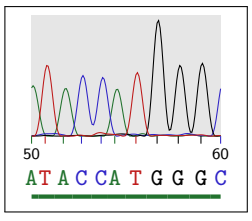

```
\begin{tikzpicture}%
    [decoration=brace]
  \pmbchromatogram[%
      sample range=401 to 900,
      samples per line=250,
      baseline skip=3.5cm
    ]{SampleScf.scf}
  \draw[decorate]
    (-0.1cm, -3.5cm) -- (-0.1cm, 0cm)
    node[pos=0.5, rotate=90, above=5pt]
      {baseline skip};
\end{tikzpicture}
```

`canvas height` =⟨*dimension*⟩

Default: `2cm`

The *canvas* is the background of the trace area. Its left and right boundaries coincide with the start and the end of the chromatogram, respectively. Its lower boundary is the baseline, and its upper border is separated from the lower one by ⟨*dimension*⟩. Although the canvas is usually transparent, its ⟨*style*⟩ can be changed. In Example 2.7, we decrease the height of the canvas and color it light gray.

Example 2.7




```
\pmbchromatogram[%
    sample range=base 50 to base 60,
    canvas style={draw=none, fill=black!10},
    canvas height=1.6cm
  ]{SampleScf.scf}
```

2.5 Traces

`trace A style` =⟨*style*⟩

Default: `pmbTraceGreen`

`trace C style` =⟨*style*⟩

Default: `pmbTraceBlue`

`trace G style` =⟨*style*⟩

Default: `pmbTraceBlack`

`trace T style` =⟨*style*⟩

Default: `pmbTraceRed`

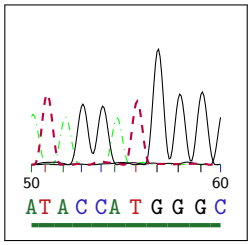
`trace style` =⟨*style*⟩

Default: (none)

The *traces* indicate variations in fluorescence intensity during chromatography, and each trace corresponds to a base. The first four keys set the respective ⟨*style*⟩ base-wise, whereas `trace style` changes all styles simultaneously. The standard styles simply color the traces; Table 2.1 lists the color specifications.

In Example 2.8, we change the style of all traces to a thin line and then add some patterns and colors to the A and T trace.

Example 2.8




```
\pmbchromatogram[%
   sample range=base 50 to base 60,
   trace style={thin},
   trace A style={dashdotted, green},
   trace T style={thick, dashed, purple}
 ]{SampleScf.scf}
```

Table 2.1: Colors defined by the chromatogram module.

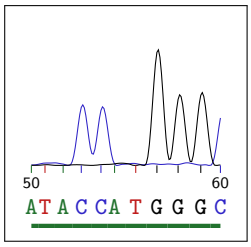
Name	xcolor model	Values	Example
pmbTraceGreen	RGB	34, 114, 46	
pmbTraceBlue	RGB	48, 37, 199	
pmbTraceBlack	RGB	0, 0, 0	
pmbTraceRed	RGB	191, 27, 27	
pmbTraceYellow	RGB	233, 230, 0	

`traces drawn` =A|C|G|T|any combination thereof

Default: `ACGT`

The value of this key governs which traces appear in the chromatogram. Any combination of the single-letter abbreviations for the standard bases will work. Example 2.9 only draws the cytosine and guanine traces.

Example 2.9




```
\pmbchromatogram[%
   sample range=base 50 to base 60,
   traces drawn=CG
 ]{SampleScf.scf}
```

2.6 Ticks

`tick A style` =⟨*style*⟩

Default: `thin, pmbTraceGreen`

`tick C style` =⟨*style*⟩

Default: `thin, pmbTraceBlue`

`tick G style` =⟨*style*⟩

Default: `thin, pmbTraceBlack`

`tick T style` =⟨*style*⟩

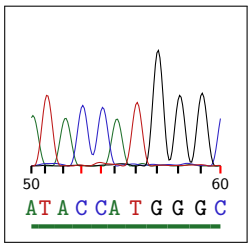
Default: `thin, pmbTraceRed`

`tick style` =⟨*style*⟩

Default: (none)

Ticks below the baseline indicate the maxima of the trace peaks. The first four keys set the respective ⟨*style*⟩ basewise, whereas `tick style` changes all styles simultaneously. Example 2.10 illustrates how one can draw thick ticks, which are red if they indicate a cytosine peak.

Example 2.10



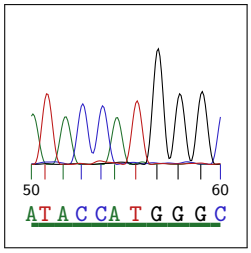

```
\pmbchromatogram[%
   sample range=base 50 to base 60,
   tick style={thick},
   tick C style={red, thick}
 ]{SampleScf.scf}
```

`tick length` =⟨*dimension*⟩

Default: `1mm`

This key determines the length of each tick. In Example 2.11, the ticks are twice as long as usual.

Example 2.11



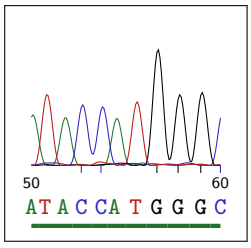

```
\pmbchromatogram[%
   sample range=base 50 to base 60,
   tick length=2mm
 ]{SampleScf.scf}
```

`ticks drawn` =A|C|G|T|any combination thereof

Default: `ACGT`

The value of this key governs which ticks appear in the chromatogram. Any combination of the single-letter abbreviations for the standard bases will work. Example 2.12 only displays the cytosine and guanine ticks.

Example 2.12




```
\pmbchromatogram[%
   sample range=base 50 to base 60,
   ticks drawn=CG
 ]{SampleScf.scf}
```

2.7 Base Labels

`base label A text` =⟨*text*⟩

Default: `\strut A`

`base label C text` =⟨*text*⟩

Default: `\strut C`

`base label G text` =⟨*text*⟩

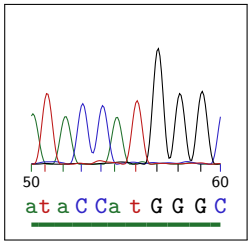
Default: `\strut G`

`base label T text` =⟨*text*⟩

Default: `\strut T`

Base labels below each tick spell the nucleotide sequence deduced from the traces. By default, the ⟨*text*⟩ that appears in these labels equals the single-letter abbreviation of the respective base. The `\strut` macro ensures equal vertical spacing. In Example 2.13, we print lowercase letters beneath adenine and thymine.

Example 2.13




```
\pmbchromatogram[%
   sample range=base 50 to base 60,
   base label A text=\strut a,
   base label T text=\strut t
 ]{SampleScf.scf}
```

`base label A style` =⟨*style*⟩

Default: `below=4pt, font=\ttfamily\footnotesize, pmbTraceGreen`

`base label C style` =⟨*style*⟩

Default: `below=4pt, font=\ttfamily\footnotesize, pmbTraceBlue`

`base label G style` =⟨*style*⟩

Default: `below=4pt, font=\ttfamily\footnotesize, pmbTraceBlack`

`base label T style` =⟨*style*⟩

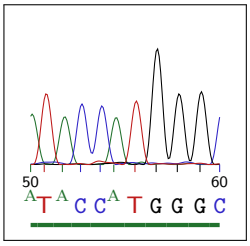
Default: `below=4pt, font=\ttfamily\footnotesize, pmbTraceRed`

`base label style` =⟨*style*⟩

Default: (none)

The first four keys set the respective ⟨*style*⟩ basewise, whereas `base label style` changes all styles simultaneously. Each base label is a Ti*k*Z node anchored to the lower end of the respective tick. Thus, the ⟨*style*⟩ should contain placement keys such as `below` or `anchor=south`. Example 2.14 shows some (imaginative) base label styles.

Example 2.14



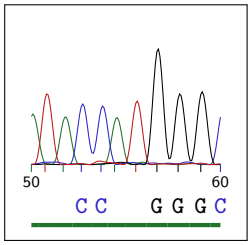

```
\pmbchromatogram[%
   sample range=base 50 to base 60,
   base label A style=%
     {below=2pt, font=\tiny},
   base label T style=%
     {below=4pt, font=\sffamily\footnotesize}
 ]{SampleScf.scf}
```

`base labels drawn` =A|C|G|T|any combination thereof

Default: `ACGT`

The value of this key governs which base labels appear in the chromatogram. Any combination of the single-letter abbreviations for the standard bases will work. Example 2.15 only displays cytosine and guanine base labels.

Example 2.15




```
\pmbchromatogram[%
   sample range=base 50 to base 60,
   base labels drawn=CG
 ]{SampleScf.scf}
```

2.8 Base Numbers

`show base numbers =`⟨*boolean*⟩

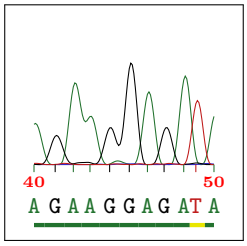
Default: `true`

`base number style =`⟨*style*⟩

Default: `pmbTraceBlack, below=-3pt, font=\sffamily\tiny`

Base numbers below the traces indicate the indices of the base peaks. `show base numbers` turns these numbers on or off, `base number style` determines their placement and appearance. Example 2.16 contains bold red base numbers that are shifted slightly upwards.

Example 2.16



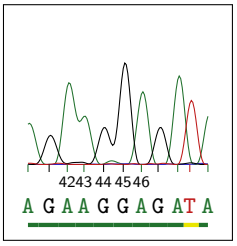

```
\pmbchromatogram[%
    sample range=base 40 to base 50,
    base number style={below=-3pt,%
      font=\rmfamily\bfseries\tiny, red}
  ]{SampleScf.scf}
```

`base number range =`⟨*lower*⟩ `to` ⟨*upper*⟩`[ step` ⟨*interval*⟩`]`

Default: `auto to auto step 10`

This key decides that every ⟨*interval*⟩th base number from ⟨*lower*⟩ to ⟨*upper*⟩ should show up in the output; the `step` part is optional. If you specify the keyword `auto` instead of a number for ⟨*lower*⟩ or ⟨*upper*⟩, the base numbers start or finish at the leftmost or rightmost base peak shown, respectively. In Example 2.17, only peaks 42 to 46 receive a number.

Example 2.17




```
\pmbchromatogram[%
    sample range=base 40 to base 50,
    base number range=42 to 46 step 1,
  ]{SampleScf.scf}
```

2.9 Probabilities

Programs such as `phred`[3] assign a *probability* or *quality value* Q to each called base after chromatography. Q is calculated from the error probability P_e by $Q = -10 \log_{10} P_e$. For example, a Q value of 20 means that 1 in 100 base calls is wrong.

`probability distance` =⟨*dimension*⟩

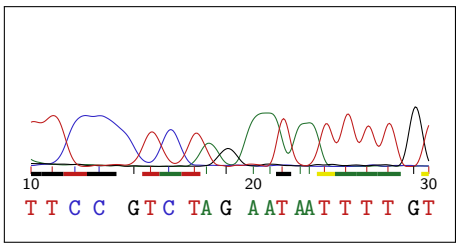
Default: `0.8cm`

`probabilities drawn` =A|C|G|T|any combination thereof

Default: `ACGT`

Base probabilities are indicated by thick rules below the base sequence. `probability distance` sets the distance between these rules and the baseline. The value of `probabilities drawn` governs which probabilities appear in the chromatogram. Any combination of the single-letter abbreviations for the standard bases will work. In Example 2.18, we shift the probability indicator upwards and only show the quality values of cytosine and thymine peaks.

Example 2.18




```
\pmbchromatogram[%
   sample range=base 10 to base 30,
   probabilities drawn=CT,
   probability distance=1mm
 ]{SampleScf.scf}
```

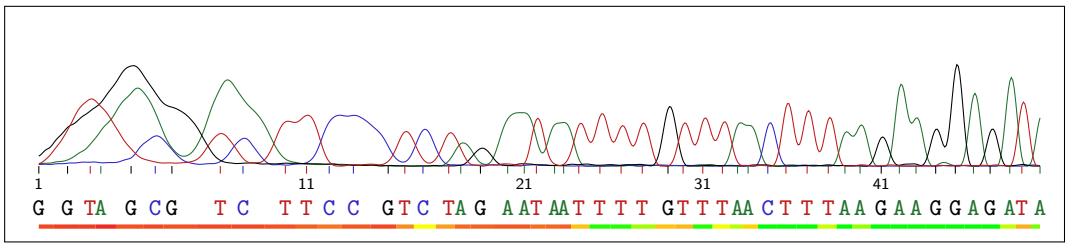
`probability style function` =⟨*Lua function name*⟩

Default: `nil`

By default, the probability rules are colored black, red, yellow and green for quality scores < 10, < 20, < 30 and ≥ 30, respectively. However, you can override this behavior by providing a ⟨*Lua function name*⟩ to `probability style function`. This Lua function must read a single argument of type number and return a string appropriate for the optional argument of Ti*k*Z's `\draw` command. For instance, the function shown in Example 2.19 determines the lowest and highest probability and colors intermediate values according to a red–yellow–green gradient.

[3] Ewing, B., Hillier, L., Wendl, M. C., and Green, P. (1998). Base-calling of automated sequencer traces using phred. I. Accuracy assessment. *Genome Res.* **8**(3), 175–185.

Example 2.19




```
\directlua{
  function probabilityGradient (prob)
    local minProb, maxProb = pgfmolbio.chromatogram.getMinMaxProbability()
    local scaledProb = prob / maxProb * 100
    local color = ""
    if scaledProb < 50 then
      color = "yellow!" .. scaledProb * 2 .. "!red"
    else
      color = "green!" .. (scaledProb - 50) * 2 .. "!yellow"
    end
    return "ultra thick, " .. color
  end
}
\pmbchromatogram[%
    samples per line=1000,
    sample range=base 1 to base 50,
    probability style function=probabilityGradient
  ]{SampleScf.scf}
```

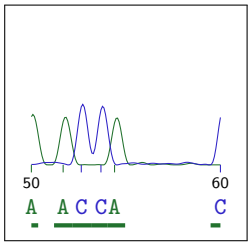
2.10 Miscellaneous Keys

`bases drawn` =A|C|G|T|any combination thereof

Default: `ACGT`

This key is a shortcut to simultaneously set `traces drawn`, `ticks drawn`, `base labels drawn` and `probabilities drawn` (see Example 2.20).

Example 2.20




```
\pmbchromatogram[%
   sample range=base 50 to base 60,
   bases drawn=AC
 ]{SampleScf.scf}
```

3 Implementation

3.1 pgfmolbio.sty

The options for the main style file determine which module(s) should be loaded. The only module so far is chromatogram.

```
\newif\ifpmb@loadmodule@chromatogram

\DeclareOption{chromatogram}{
  \pmb@loadmodule@chromatogramtrue
}
\ProcessOptions
```

The main style file also loads the following packages and Ti*k*Z libraries.

```
\RequirePackage{luatexbase-modutils}
  \RequireLuaModule{lualibs}
\RequirePackage{tikz}
  \usetikzlibrary{positioning}

\RequirePackage{xcolor}
```

\pgfmolbioset

#1: The ⟨*module*⟩ to which the options apply.
#2: A ⟨*key-value list*⟩ which configures the graphs.

```
\newcommand\pgfmolbioset[2][]{%
  \def\@tempa{#1}%
  \ifx\@tempa\@empty%
    \pgfqkeys{/pgfmolbio}{#2}%
  \else%
    \pgfqkeys{/pgfmolbio/#1}{#2}%
  \fi%
}
```

Finally, we load the module(s) requested by the user.

```
\ifpmb@loadmodule@chromatogram
  \input{pgfmolbio.chromatogram.tex}
\fi
```

3.2 pgfmolbio.chromatogram.tex

Since the Lua script of the chromatogram module does the bulk of the work, we can keep the TEX file relatively short.

```
\RequireLuaModule{pgfmolbio.chromatogram}
```

We define five custom colors for the traces and probability indicators (see Table 2.1).

```
\definecolor{pmbTraceGreen}{RGB}{34,114,46}
\definecolor{pmbTraceBlue}{RGB}{48,37,199}
\definecolor{pmbTraceBlack}{RGB}{0,0,0}
\definecolor{pmbTraceRed}{RGB}{191,27,27}
\definecolor{pmbTraceYellow}{RGB}{233,230,0}
```

\@pmb@chr@keydef

#1: ⟨*key*⟩ name

Most of the keys store their value in a macro. \@pmb@chr@keydef simplifies the declaration of such keys: The ⟨*key*⟩ defines the macro \pmb@chr@⟨*key*⟩, which expands to the value of the key.

```
\def\@pmb@chr@keydef#1{%
  \pgfkeysdef{/pgfmolbio/chromatogram/#1}{%
    \expandafter\def\csname pmb@chr@#1\endcsname{##1}%
  }%
}
```

\@pmb@chr@stylekeydef

#1: ⟨*key*⟩ name

This macro defines a ⟨*key*⟩ that saves its value (which is a key-value list) in the style key ⟨*key*⟩@style for internal usage.

```
\def\@pmb@chr@stylekeydef#1{%
  \pgfkeysdef{/pgfmolbio/chromatogram/#1}{%
    \pgfkeys{/pgfmolbio/chromatogram/#1@style/.style={##1}}%
  }%
}
```

\@pmb@chr@getkey

#1: ⟨*key*⟩ name

\@pmb@chr@getkey retrieves the value stored by the ⟨*key*⟩.

```
\def\@pmb@chr@getkey#1{\csname pmb@chr@#1\endcsname}
```

After providing these auxiliary macros, we define all keys of the chromatogram module.

```
\@pmb@chr@keydef{sample range}

\@pmb@chr@keydef{x unit}
\@pmb@chr@keydef{y unit}
\@pmb@chr@keydef{samples per line}
\@pmb@chr@keydef{baseline skip}
\@pmb@chr@stylekeydef{canvas style}
\@pmb@chr@keydef{canvas height}

\@pmb@chr@stylekeydef{trace A style}
\@pmb@chr@stylekeydef{trace C style}
\@pmb@chr@stylekeydef{trace G style}
\@pmb@chr@stylekeydef{trace T style}
\pgfkeysdef{/pgfmolbio/chromatogram/trace style}{%
  \pgfmolbioset[chromatogram]{
    trace A style={#1},
    trace C style={#1},
    trace G style={#1},
    trace T style={#1}
  }%
}
\@pmb@chr@keydef{traces drawn}

\@pmb@chr@stylekeydef{tick A style}
\@pmb@chr@stylekeydef{tick C style}
\@pmb@chr@stylekeydef{tick G style}
\@pmb@chr@stylekeydef{tick T style}
\pgfkeysdef{/pgfmolbio/chromatogram/tick style}{%
  \pgfmolbioset[chromatogram]{
    tick A style={#1},
    tick C style={#1},
    tick G style={#1},
    tick T style={#1}
  }%
}
\@pmb@chr@keydef{tick length}
\@pmb@chr@keydef{ticks drawn}

\@pmb@chr@keydef{base label A text}
\@pmb@chr@keydef{base label C text}
\@pmb@chr@keydef{base label G text}
\@pmb@chr@keydef{base label T text}
\@pmb@chr@stylekeydef{base label A style}
\@pmb@chr@stylekeydef{base label C style}
```

```
\@pmb@chr@stylekeydef{base label G style}
\@pmb@chr@stylekeydef{base label T style}
\pgfkeysdef{/pgfmolbio/chromatogram/base label style}{%
  \pgfmolbioset[chromatogram]{
    base label A style={#1},
    base label C style={#1},
    base label G style={#1},
    base label T style={#1}
  }%
}
\@pmb@chr@keydef{base labels drawn}

\newif\ifpmb@chr@showbasenumbers
\pgfkeys{/pgfmolbio/chromatogram/show base numbers/%
  .is if=pmb@chr@showbasenumbers}
\@pmb@chr@stylekeydef{base number style}
\@pmb@chr@keydef{base number range}

\@pmb@chr@keydef{probability distance}
\@pmb@chr@keydef{probabilities drawn}
\@pmb@chr@keydef{probability style function}

\pgfkeysdef{/pgfmolbio/chromatogram/bases drawn}{%
  \pgfmolbioset[chromatogram]{
    traces drawn=#1,
    ticks drawn=#1,
    base labels drawn=#1,
    probabilities drawn=#1
  }%
}
```

These keys receive a default value.

```
\pgfmolbioset[chromatogram]{%
  sample range=1 to 500 step 1,
  x unit=0.2mm,
  y unit=0.01mm,
  samples per line=500,
  baseline skip=3cm,
  canvas style={draw=none, fill=none},
  canvas height=2cm,
  trace A style={pmbTraceGreen},
  trace C style={pmbTraceBlue},
  trace G style={pmbTraceBlack},
  trace T style={pmbTraceRed},
  tick A style={thin, pmbTraceGreen},
  tick C style={thin, pmbTraceBlue},
  tick G style={thin, pmbTraceBlack},
  tick T style={thin, pmbTraceRed},
  tick length=1mm,
```

```
  base label A text=\strut A,
  base label C text=\strut C,
  base label G text=\strut G,
  base label T text=\strut T,
  base label A style=%
    {below=4pt, font=\ttfamily\footnotesize, pmbTraceGreen},
  base label C style=%
    {below=4pt, font=\ttfamily\footnotesize, pmbTraceBlue},
  base label G style=%
    {below=4pt, font=\ttfamily\footnotesize, pmbTraceBlack},
  base label T style=%
    {below=4pt, font=\ttfamily\footnotesize, pmbTraceRed},
  show base numbers,
  base number style={pmbTraceBlack, below=-3pt, font=\sffamily\tiny},
  base number range=auto to auto step 10,
  probability distance=0.8cm,
  probability style function=nil,
  bases drawn=ACGT
}
```

\pmbchromatogram

#1: A ⟨*key-value list*⟩ that configures the chromatogram.
#2: The name of an ⟨*scf file*⟩.

If \pmbchromatogram appears outside of a tikzpicture, we implicitly start this environment, otherwise we begin a new group. "Within a tikzpicture" means that \useasboundingbox is defined.

```
\newif\ifpmb@chr@tikzpicture

\newcommand\pmbchromatogram[2][]{%
  \@ifundefined{useasboundingbox}%
    {\pmb@chr@tikzpicturefalse\begin{tikzpicture}}%
    {\pmb@chr@tikzpicturetrue\begingroup}%
```

Of course, we consider the ⟨*key-value list*⟩ before drawing the chromatogram.

```
  \pgfmolbioset[chromatogram]{#1}%
```

We invoke three functions of the chromatogram Lua script: (1) readScfFile reads the given ⟨*scf file*⟩ (see section 3.3.4). (2) setParameters passes the values stored by the keys to the Lua script (section 3.3.5). (3) printTikzChromatogram returns the drawing commands for the chromatogram to the TEX input stream (section 3.3.6).

```
  \directlua{
    pgfmolbio.chromatogram.readScfFile("#2")
    pgfmolbio.chromatogram.setParameters{
      sampleRange = "\@pmb@chr@getkey{sample range}",
```

```
    xUnit = dimen("\@pmb@chr@getkey{x unit}")[1],
    yUnit = dimen("\@pmb@chr@getkey{y unit}")[1],
    samplesPerLine = \@pmb@chr@getkey{samples per line},
    baselineSkip = dimen("\@pmb@chr@getkey{baseline skip}")[1],
    canvasHeight = dimen("\@pmb@chr@getkey{canvas height}")[1],
    tracesDrawn = "\@pmb@chr@getkey{traces drawn}",
    tickLength = dimen("\@pmb@chr@getkey{tick length}")[1],
    ticksDrawn = "\@pmb@chr@getkey{ticks drawn}",
    baseLabelsDrawn = "\@pmb@chr@getkey{base labels drawn}",
    showBaseNumbers = \ifpmb@chr@showbasenumbers true\else false\fi,
    baseNumberRange = "\@pmb@chr@getkey{base number range}",
    probDistance = dimen("\@pmb@chr@getkey{probability distance}")[1],
    probabilitiesDrawn = "\@pmb@chr@getkey{probabilities drawn}",
    probStyle = \@pmb@chr@getkey{probability style function}
  }
  pgfmolbio.chromatogram.printTikzChromatogram()
}%
```

At the end of \pmbchromatogram, we either close the tikzpicture or the group, depending on how we started.

```
  \ifpmb@chr@tikzpicture\endgroup\else\end{tikzpicture}\fi%
}
```

3.3 pgfmolbio.chromatogram.lua

This Lua script is the true workhorse of the chromatogram module. Remember that the documentation for the Staden package[1] is the definite source for information on the scf file format.

3.3.1 Module-Wide Variables

- ALL_BASES: A table of four indexed string fields, which represent the nucleotide single-letter abbreviations.
- PGFKEYS_PATH: A string that contains the pgfkeys path for chromatogram keys.
- header: A table of 14 named number fields that save the information in the scf header (see section 3.3.3).
- samples: A table of four named subtables A, C, G, T. Each subtable contains header.samplesNumber indexed number fields that represent the fluorescence intensities along a trace.

[1] http://staden.sourceforge.net/

- `peaks`: A table of `header.basesNumber` indexed subtables which in turn contain three named fields:
  - `offset`: A number indicating the offset of the current peak.
  - `prob`: A table of four named number fields `A`, `C`, `G`, `T`. These numbers store the probability that the current peak is one of the four bases.
  - `base`: A string that states the base represented by the current peak.
- `parms`: A table of 25 named fields that comprise the parameters of the chromatogram. Most of the fields correspond to a key from the chromatogram module. For a detailed description, see section 3.3.5.
- `selectedPeaks`: A table of zero to `header.basesNumber` indexed subtables (section 3.3.6 explains how the exact number is determined). This variable is similar to `peaks`, but it only describes the peaks in the displayed part of the chromatogram, which is selected by the `samples range` key (hence the name). Each subtable of `selectedPeaks` consists of the following five named fields:
  - `offset`: A number indicating the offset of the current peak in "transformed" coordinates (i. e., the x-coordinate of the first sample point shown equals 1).
  - `base`: See `peaks.base` above.
  - `prob`: See `peaks.prob` above.
  - `baseIndex`: A number that stores the index of the current peak. The first detected peak in the chromatogram has index 1.
  - `probXRight`: A number corresponding to the right x-coordinate of the probability indicator.
- `lastScfFile`: A string that equals the name of the last `scf` file loaded.

```
local ALL_BASES = {"A", "C", "G", "T"}
local PGFKEYS_PATH = "/pgfmolbio/chromatogram/"

local header, samples,
  peaks, parms,
  selectedPeaks,
  lastScfFile
```

3.3.2 Auxiliary Functions

`baseToSampleIndex` converts its argument to an x-coordinate. If `baseIndex` is a number, the function simply returns it. However, if the argument is a string of the form `"base` ⟨*number*⟩`"` (as in a valid value for the `sample range` key), `baseToSampleIndex` returns the offset of the ⟨*number*⟩-th peak.

```
local function baseToSampleIndex (baseIndex)
  local result = tonumber(baseIndex)
  if result then
    return result
  else
    result = string.match(baseIndex, "base%s*(%d+)")
    if tonumber(result) then
      return peaks[tonumber(result)].offset
    end
  end
end
```

stdProbStyle is the default probability style function. It returns a string representing an optional argument of \draw. Depending on the value of prob, the ultra thick probability rule thus drawn is colored black, red, yellow or green for quality scores < 10, < 20, < 30 or ≥ 30, respectively (see also section 2.9).

```
local function stdProbStyle (prob)
  local color = ""
  if prob >= 0 and prob < 10 then
    color = "black"
  elseif prob >= 10 and prob < 20 then
    color = "pmbTraceRed"
  elseif prob >= 20 and prob < 30 then
    color = "pmbTraceYellow"
  else
    color = "pmbTraceGreen"
  end
  return "ultra thick, " .. color
end
```

findBasesInStr searches for nucleotide single-letter abbreviations in its string argument. It returns a table of zero to four indexed string fields (one field per character found, which contains that letter).

```
local function findBasesInStr (target)
  if not target then return end
  local result = {}
  for _, v in ipairs(ALL_BASES) do
    if string.find(string.upper(target), v) then
      table.insert(result, v)
    end
  end
  return result
end
```

`getMinMaxProbability` is the only non-local auxiliary function (thus, we were able to call it in Example 2.19). It returns the minimum and maximum probability value in the current chromatogram.

```
function getMinMaxProbability ()
  local minProb = 0
  local maxProb = 0
  for _, currPeak in ipairs(selectedPeaks) do
    for __, currProb in pairs(currPeak.prob) do
      if currProb > maxProb then maxProb = currProb end
      if currProb < minProb then minProb = currProb end
    end
  end
  return minProb, maxProb
end
```

`getRange` extracts the strings ⟨*lower*⟩, ⟨*upper*⟩ and ⟨*interval*⟩ from `rangeInput` by applying the pattern in `regExp`. `rangeInput` contains the value of either the `sample range` or the `base number range` key (see sections 2.3 and 2.8).

```
local function getRange (rangeInput, regExp)
  local lower, upper = string.match(rangeInput, regExp)
  local step = string.match(rangeInput, "step%s*(%d*)")
  return lower, upper, step
end
```

`readInt` reads `n` bytes from a `file`, starting at `offset` or at the current position if `offset` is `nil`. By assuming big-endian byte order, the byte sequence is converted to a number and returned.

```
local function readInt (file, n, offset)
  if offset then file:seek("set", offset) end
  local result = 0
  for i = 1, n do
    result = result * 0x100 + string.byte(file:read(1))
  end
  return result
end
```

3.3.3 Evaluate the `scf` File

`evaluateScfFile` collects the relevant data from an open `scf file`. *Firstly*, the global variable `header` saves the information in the file header:

- `magicNumber`: Each `scf` file must start with the four bytes `2E736366`, which is the string "`.scf`". If this sequence is absent, the chromatogram module raises an error.

- `samplesNumber` – The number of sample points.
- `samplesOffset` – The offset of the sample data start.
- `basesNumber` – The number of recognized bases.
- `version`: Since the chromatogram module currently only supports `scf` version 3.00 (the string "`3.00`" equals `332E3030`), TEX stops with an error message if the file version is different.
- `sampleSize` – The size of each sample point in bytes.

```
local function evaluateScfFile (file)
  samples = {A = {}, C = {}, G = {}, T = {}}
  peaks = {}
  header = {
    magicNumber = readInt(file, 4, 0),
    samplesNumber = readInt(file, 4),
    samplesOffset = readInt(file, 4),
    basesNumber = readInt(file, 4),
    leftClip = readInt(file, 4),
    rightClip = readInt(file, 4),
    basesOffset = readInt(file, 4),
    comments = readInt(file, 4),
    commentsOffset = readInt(file, 4),
    version = readInt(file, 4),
    sampleSize = readInt(file, 4),
    codeSet = readInt(file, 4),
    privateSize = readInt(file, 4),
    privateOffset = readInt(file, 4)
  }
  if header.magicNumber ~= 0x2E736366 then
    tex.error("Magic number in scf file '" .. lastScfFile .. "' corrupt!")
  end
  if header.version ~= 0x332E3030 then
    tex.error("Scf file '" .. lastScfFile .. "' is not version 3.00!")
  end
```

Secondly, the global variable `samples` receives the samples data from the file. Note that the values of the sample points are stored as unsigned integers representing second derivatives (i. e., differences between differences between two consecutive sample points). Hence, we convert them back to signed, absolute values.

```
  file:seek("set", header.samplesOffset)
  for baseIndex, baseName in ipairs(ALL_BASES) do
    for i = 1, header.samplesNumber do
      samples[baseName][i] = readInt(file, header.sampleSize)
    end
```

```
    for _ = 1, 2 do
      local preValue = 0
      for i = 1, header.samplesNumber do
        samples[baseName][i] = samples[baseName][i] + preValue
        if samples[baseName][i] > 0xFFFF then
          samples[baseName][i] = samples[baseName][i] - 0x10000
        end
        preValue = samples[baseName][i]
      end
    end
  end
```

Finally, we store the peak information in the global variable `peaks`.

```
  for i = 1, header.basesNumber do
    peaks[i] = {
      offset = readInt(file, 4),
      prob = {A, C, G, T},
      base
    }
  end

  for i = 1, header.basesNumber do
    peaks[i].prob.A = readInt(file, 1)
  end

  for i = 1, header.basesNumber do
    peaks[i].prob.C = readInt(file, 1)
  end

  for i = 1, header.basesNumber do
    peaks[i].prob.G = readInt(file, 1)
  end

  for i = 1, header.basesNumber do
    peaks[i].prob.T = readInt(file, 1)
  end

  for i = 1, header.basesNumber do
    peaks[i].base = string.char(readInt(file, 1))
  end
end
```

3.3.4 Read the `scf` File

The public function `readScfFile` checks whether the requested `scf` file "`filename`" corresponds to the most recently opened one. In this case, the variables `peaks` and

`samples` already contain the relevant data, so we can refrain from re-reading the file. Otherwise, the program tries to open and evaluate the specified file, raising an error on failure.

```
function readScfFile (filename)
  if filename ~= lastScfFile then
    lastScfFile = filename
    local scfFile, errorMsg = io.open(filename, "rb")
    if not scfFile then tex.error(errorMsg) end
    evaluateScfFile(scfFile)
    scfFile:close()
  end
end
```

3.3.5 Set Chromatogram Parameters

The public function `setParameters` provides an interface between the key-value configuration system of the chromatogram module and the Lua function that actually draws the chromatogram.

First, `getRange` extracts the range and step values from `sample range` and `base number range`. For example, assume that the value of `sample range` is `"base 10 to base 50 step 2"`. Consequently, the three local variables `sampleRangeMin`, `sampleRangeMax` and `sampleRangeStep` receive the values `"base 10"`, `"base 50"` and `"2"`, respectively.

```
function setParameters (newParms)
  local sampleRangeMin, sampleRangeMax, sampleRangeStep =
    getRange(
      newParms.sampleRange or "1 to 500 step 1",
      "([base]*%s*%d+)%s*to%s*([base]*%s*%d+)"
    )
  local baseNumberRangeMin, baseNumberRangeMax, baseNumberRangeStep =
    getRange(
      newParms.baseNumberRange or "auto to auto step 10",
      "([auto%d]*)%s+to%s+([auto%d]*)"
    )
```

Most fields of the table `parms` are self-explanatory, since their name is similar to their corresponding key. Note that:

- We assign a default value to each field of `parms`.
- All dimensions are converted to scaled points (via the `dimen` function provided by `lualibs`).
- If the ⟨*lower*⟩ or ⟨*upper*⟩ limit of `base number range` equals the string `"auto"`, the corresponding field is set to `-1`.

- `coordUnit` and `coordFmtStr` tell the `number.todimen` function that it should convert a dimension in scaled points to a dimension in millimeters and format its output as the string "⟨*value*⟩`mm`".

```
parms = {
  sampleMin = baseToSampleIndex(sampleRangeMin) or 1,
  sampleMax = baseToSampleIndex(sampleRangeMax) or 500,
  sampleStep = sampleRangeStep or 1,
  xUnit = newParms.xUnit or dimen("0.2mm")[1],
  yUnit = newParms.yUnit or dimen("0.01mm")[1],
  samplesPerLine = newParms.samplesPerLine or 500,
  baselineSkip = newParms.baselineSkip or dimen("3cm")[1],
  canvasHeight= newParms.canvasHeight or dimen("2cm")[1],
  traceStyle = {
    A = PGFKEYS_PATH .. "trace A style@style",
    C = PGFKEYS_PATH .. "trace C style@style",
    G = PGFKEYS_PATH .. "trace G style@style",
    T = PGFKEYS_PATH .. "trace T style@style"
  },
  tickStyle = {
    A = PGFKEYS_PATH .. "tick A style@style",
    C = PGFKEYS_PATH .. "tick C style@style",
    G = PGFKEYS_PATH .. "tick G style@style",
    T = PGFKEYS_PATH .. "tick T style@style"
  },
  tickLength = newParms.tickLength or dimen("1mm")[1],
  baseLabelText = {
    A = "\\csname pmb@chr@base label A text\\endcsname",
    C = "\\csname pmb@chr@base label C text\\endcsname",
    G = "\\csname pmb@chr@base label G text\\endcsname",
    T = "\\csname pmb@chr@base label T text\\endcsname"
  },
  baseLabelStyle = {
    A = PGFKEYS_PATH .. "base label A style@style",
    C = PGFKEYS_PATH .. "base label C style@style",
    G = PGFKEYS_PATH .. "base label G style@style",
    T = PGFKEYS_PATH .. "base label T style@style"
  },
  showBaseNumbers = newParms.showBaseNumbers,
  baseNumberMin = tonumber(baseNumberRangeMin) or -1,
  baseNumberMax = tonumber(baseNumberRangeMax) or -1,
  baseNumberStep = tonumber(baseNumberRangeStep) or 10,
  probDistance = newParms.probDistance or dimen("0.8cm")[1],
  probStyle = newParms.probStyle or stdProbStyle,
  tracesDrawn = findBasesInStr(newParms.tracesDrawn) or ALL_BASES,
  ticksDrawn = newParms.ticksDrawn or "ACGT",
  baseLabelsDrawn = newParms.baseLabelsDrawn or "ACGT",
  probabilitiesDrawn = newParms.probabilitiesDrawn or "ACGT",
  coordUnit = "mm",
```

```
    coordFmtStr = "%s%s"
  }
end
```

3.3.6 Print the Chromatogram

The global function `printTikzChromatogram` writes all commands that draw the chromatogram to the TEX input stream (via `tex.sprint`).

```
function printTikzChromatogram ()
```

(1) Select peaks to draw In order to simplify the drawing operations, we select the peaks that appear in the final output and store information on them in the table `selectedPeaks`.

```
  selectedPeaks = {}
  local tIndex = 1
  for rPeakIndex, currPeak in ipairs(peaks) do
    if currPeak.offset >= parms.sampleMin
        and currPeak.offset <= parms.sampleMax then
      selectedPeaks[tIndex] = {
        offset = currPeak.offset + 1 - parms.sampleMin,
        base = currPeak.base,
        prob = currPeak.prob,
        baseIndex = rPeakIndex,
        probXRight = parms.sampleMax + 1 - parms.sampleMin
      }
```

The right x-coordinate of the probability indicator (`probXRight`) is the mean between the offsets of the adjacent peaks. For the last peak, `probXRight` equals the largest transformed x-coordinate.

```
      if tIndex > 1 then
        selectedPeaks[tIndex-1].probXRight =
          (selectedPeaks[tIndex-1].offset
          + selectedPeaks[tIndex].offset) / 2
      end
      tIndex = tIndex + 1
    end
  end
```

Furthermore, we adjust `parms.baseNumberMin` and `parms.baseNumberMax` if any peak was detected in the displayed part of the chromatogram. The value `-1`, which indicates the keyword `auto`, is replaced by the index of the first or last peak, respectively.

```
if tIndex > 1 then
  if parms.baseNumberMin == -1 then
    parms.baseNumberMin = selectedPeaks[1].baseIndex
  end
  if parms.baseNumberMax == -1 then
    parms.baseNumberMax = selectedPeaks[tIndex-1].baseIndex
  end
end
```

(2) Canvas For each line, we draw a rectangle in `canvas style` whose left border coincides with the y-axis.
`yLower`, `yUpper`, `xRight`: rectangle coordinates;
`currLine`: current line, starting from 0;
`samplesLeft`: sample points left to draw after the end of the current line.

```
local samplesLeft = parms.sampleMax - parms.sampleMin + 1
local currLine = 0
while samplesLeft > 0 do
  local yLower = -currLine * parms.baselineSkip
  local yUpper = -currLine * parms.baselineSkip + parms.canvasHeight
  local xRight =
    (math.min(parms.samplesPerLine, samplesLeft) - 1) * parms.xUnit
  tex.sprint(
    "\\draw[" .. PGFKEYS_PATH .. "canvas style@style] (" ..
    number.todimen(0, parms.coordUnit, parms.coordFmtStr) ..
    ", " ..
    number.todimen(yLower, parms.coordUnit, parms.coordFmtStr) ..
    ") rectangle (" ..
    number.todimen(xRight, parms.coordUnit, parms.coordFmtStr) ..
    ", " ..
    number.todimen(yUpper, parms.coordUnit, parms.coordFmtStr) ..
    ");\n"
  )
  samplesLeft = samplesLeft - parms.samplesPerLine
  currLine = currLine + 1
end
```

(3) Traces The traces in `parms.tracesDrawn` are drawn sequentially.
`currSampleIndex`: original x-coordinate of a sample point;
`sampleX`: transformed x-coordinate of a sample point, starting at 1;
`x` and `y`: "real" coordinates (in scaled points) of a sample point;
`currLine`: current line, starting at 0;
`firstPointInLine`: boolean that indicates if the current sample point is the first in the line.

```
for _, baseName in ipairs(parms.tracesDrawn) do
  tex.sprint("\\draw[" .. parms.traceStyle[baseName] .. "] ")
  local currSampleIndex = parms.sampleMin
  local sampleX = 1
  local x = 0
  local y = 0
  local currLine = 0
  local firstPointInLine = true
```

We iterate over each sample point. As long as the current sample point is within the selected range, we calculate the real coordinates of the sample point; add the lineto operator `--` if at least one sample point has already appeared in the current line; and write the point to the TEX input stream in Ti*k*Z's canvas coordinate system.

```
  while currSampleIndex <= parms.sampleMax do
    x = ((sampleX - 1) % parms.samplesPerLine) * parms.xUnit
    y = samples[baseName][currSampleIndex] * parms.yUnit
      - currLine * parms.baselineSkip
    if sampleX % parms.sampleStep == 0 then
      if not firstPointInLine then
        tex.sprint(" -- ")
      else
        firstPointInLine = false
      end
      tex.sprint(
        "(" ..
        number.todimen(x, parms.coordUnit, parms.coordFmtStr) ..
        ", " ..
        number.todimen(y, parms.coordUnit, parms.coordFmtStr) ..
        ")"
      )
    end
```

Besides, we add line breaks at the appropriate positions.

```
    if sampleX ~= parms.sampleMax + 1 - parms.sampleMin then
      if sampleX >= (currLine + 1) * parms.samplesPerLine then
        currLine = currLine + 1
        tex.sprint(";\n\\draw[" .. parms.traceStyle[baseName] .. "] ")
        firstPointInLine = true
      end
    else
      tex.sprint(";\n")
    end
  sampleX = sampleX + 1
  currSampleIndex = currSampleIndex + 1
  end
end
```

(4) Annotations We iterate over each selected peak and start by finding the line in which the first peak resides.
`currLine`: current line, starting at 0;
`lastProbX`: right x-coordinate of the probability rule of the last peak;
`probRemainder`: string that draws the remainder of a probability indicator following a line break;
`x`, `yUpper`, `yLower`: "real" tick coordinates;
`tickOperation`: string that equals either Ti*k*Z's moveto or lineto operation, depending on whether the current peak should be marked with a tick.

```
local currLine = 0
local lastProbX = 1
local probRemainder = false

for _, currPeak in ipairs(selectedPeaks) do
  while currPeak.offset > (currLine + 1) * parms.samplesPerLine do
    currLine = currLine + 1
  end

  local x = ((currPeak.offset - 1) % parms.samplesPerLine) * parms.xUnit
  local yUpper = -currLine * parms.baselineSkip
  local yLower = -currLine * parms.baselineSkip - parms.tickLength
  local tickOperation = ""
  if string.find(string.upper(parms.ticksDrawn), currPeak.base) then
    tickOperation = "--"
  end
```

(4a) Ticks and labels Having calculated all coordinates, we draw the tick and the base label, given the latter has been specified by `base labels drawn`.

```
  tex.sprint(
    "\\draw[" ..
    parms.tickStyle[currPeak.base] ..
    "] (" ..
    number.todimen(x, parms.coordUnit, parms.coordFmtStr) ..
    ", " ..
    number.todimen(yUpper, parms.coordUnit, parms.coordFmtStr) ..
    ") " ..
    tickOperation ..
    " (" ..
    number.todimen(x, parms.coordUnit, parms.coordFmtStr) ..
    ", " ..
    number.todimen(yLower, parms.coordUnit, parms.coordFmtStr) ..
    ")"
  )
  if string.find(string.upper(parms.baseLabelsDrawn), currPeak.base) then
    tex.sprint(
```

```
      " node[" ..
      parms.baseLabelStyle[currPeak.base] ..
      "] {" ..
      parms.baseLabelText[currPeak.base] ..
      "}"
    )
  end
```

(4b) Base numbers If `show base numbers` is true and the current base number is within the interval given by `base number range`, a base number is printed.

```
  if parms.showBaseNumbers
      and currPeak.baseIndex >= parms.baseNumberMin
      and currPeak.baseIndex <= parms.baseNumberMax
      and (currPeak.baseIndex - parms.baseNumberMin)
        % parms.baseNumberStep == 0 then
    tex.sprint(
      " node[" .. PGFKEYS_PATH .. "base number style@style] {\\strut " ..
      currPeak.baseIndex ..
      "}"
    )
  end
  tex.sprint(";\n")
```

(4c) Probabilities First, we draw the remainder of the last probability rule. Such a remainder has been stored in `probRemainder` if the last rule had protruded into the right margin (see below). Furthermore, we determine if a probability rule should appear beneath the current peak.

```
  if probRemainder then
    tex.sprint(probRemainder)
    probRemainder = false
  end
  local drawCurrProb = string.find(
    string.upper(parms.probabilitiesDrawn),
    currPeak.base
  )
```

Now comes the tricky part. Whenever we choose to paint a probability rule, we may envision three scenarios. *Firstly*, the probability rule starts in the left margin of the current line (i. e., `xLeft` is negative). This means that the part protruding into the left margin must instead appear at the end of the last line. Therefore, we calculate the coordinates of this part (storing them in `xLeftPrev`, `xRightPrev` and `yPrev`) and draw the segment. Since the remainder of the rule necessarily starts at the left border of the current line, we set `xLeft` to zero.

```
local xLeft = lastProbX - 1 - currLine * parms.samplesPerLine
if xLeft < 0 then
  local xLeftPrev = (parms.samplesPerLine + xLeft) * parms.xUnit
  local xRightPrev = (parms.samplesPerLine - 1) * parms.xUnit
  local yPrev = -(currLine-1) * parms.baselineSkip - parms.probDistance
  if drawCurrProb then
    tex.sprint(
      "\\draw[" ..
      parms.probStyle(currPeak.prob[currPeak.base]) ..
      " ] (" ..
      number.todimen(xLeftPrev, parms.coordUnit, parms.coordFmtStr) ..
      ", " ..
      number.todimen(yPrev, parms.coordUnit, parms.coordFmtStr) ..
      ") -- (" ..
      number.todimen(xRightPrev, parms.coordUnit, parms.coordFmtStr) ..
      ", " ..
      number.todimen(yPrev, parms.coordUnit, parms.coordFmtStr) ..
      ");\n"
    )
  end
  xLeft = 0
else
  xLeft = xLeft * parms.xUnit
end
```

Secondly, the probability rule ends in the right margin of the current line (i. e., `xRight` at least equals `parms.samplesPerLine`). This means that the part protruding into the right margin must instead appear at the start of the following line. Therefore, we calculate the coordinates of this part (storing them in `xRightNext` and `yNext`) and save the drawing command in `probRemainder` (whose contents were printed above). Since the remainder of the rule necessarily ends at the right border of the current line, we set `xRight` to this coordinate.

```
local xRight = currPeak.probXRight - 1 - currLine * parms.samplesPerLine
if xRight >= parms.samplesPerLine then
  if drawCurrProb then
    local xRightNext = (xRight - parms.samplesPerLine) * parms.xUnit
    local yNext = -(currLine+1) * parms.baselineSkip - parms.probDistance
    probRemainder =
      "\\draw[" ..
      parms.probStyle(currPeak.prob[currPeak.base]) ..
      " ] (" ..
      number.todimen(0, parms.coordUnit, parms.coordFmtStr) ..
      ", " ..
      number.todimen(yNext, parms.coordUnit, parms.coordFmtStr) ..
      ") -- (" ..
      number.todimen(xRightNext, parms.coordUnit, parms.coordFmtStr) ..
      ", " ..
```

```
        number.todimen(yNext, parms.coordUnit, parms.coordFmtStr) ..
        ");\n"
    end
    xRight = (parms.samplesPerLine - 1) * parms.xUnit
  else
    xRight = xRight * parms.xUnit
  end
```

Thirdly, the probability rule starts and ends within the boundaries of the current line. In this lucky case, the y-coordinate is the only one missing, since we previously calculated `xLeft` (case 1) and `xRight` (case 2). Drawing of the probability rule proceeds as usual.

```
  local y = -currLine * parms.baselineSkip - parms.probDistance
  if drawCurrProb then
    tex.sprint(
      "\\draw[" ..
      parms.probStyle(currPeak.prob[currPeak.base]) ..
      " ] (" ..
      number.todimen(xLeft, parms.coordUnit, parms.coordFmtStr) ..
      ", " ..
      number.todimen(y, parms.coordUnit, parms.coordFmtStr) ..
      ") -- (" ..
      number.todimen(xRight, parms.coordUnit, parms.coordFmtStr) ..
      ", " ..
      number.todimen(y, parms.coordUnit, parms.coordFmtStr) ..
      ");\n"
    )
  end
  lastProbX = currPeak.probXRight
 end
end
```