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

Wolfgang Esser-Skala†

2024/06/17

CTAN: https://www.ctan.org/pkg/pgfmolbio

The experimental package pgfmolbio draws graphs typically found in molecular biology texts. Currently, the package contains three modules: chromatogram creates DNA sequencing chromatograms from files in standard chromatogram format (scf); domains draws protein domain diagrams; convert integrates pgfmolbio with TEX engines that lack Lua support.

*This document describes version v0.21a, dated 2024/06/17.

†Computational Systems Biology Group, Department of Biosciences and Medical Biology, University of Salzburg, Austria; Wolfgang.Esser-Skala@plus.ac.at

Contents

1 Introduction **1**
1.1 About `pgfmolbio` . . . 1
1.2 Getting Started . . . 2

2 The chromatogram module **3**
2.1 Overview . . . 3
2.2 Drawing Chromatograms . . . 3
2.3 Displaying Parts of the Chromatogram . . . 4
2.4 General Layout . . . 6
2.5 Traces . . . 8
2.6 Ticks . . . 10
2.7 Base Labels . . . 11
2.8 Base Numbers . . . 13
2.9 Probabilities . . . 14
2.10 Miscellaneous Keys . . . 15

3 The domains module **17**
3.1 Overview . . . 17
3.2 Domain Diagrams and Their Features . . . 17
3.3 General Layout . . . 18
3.4 Feature Styles and Shapes . . . 23
3.5 Standard Features . . . 28
3.6 Disulfides and Ranges . . . 30
3.7 Ruler . . . 33
3.8 Sequences . . . 34
3.9 Secondary Structure . . . 38
3.10 File Input . . . 43

4 The convert module **44**
4.1 Overview . . . 44
4.2 Converting Chromatograms . . . 44
4.3 Converting Domain Diagrams . . . 46

5 Implementation **50**
5.1 `pgfmolbio.sty` . . . 50
5.2 `pgfmolbio.lua` . . . 52
5.3 `pgfmolbio.chromatogram.tex` . . . 53

5.4 `pgfmolbio.chromatogram.lua` 58
5.4.1 Module-Wide Variables and Auxiliary Functions 59
5.4.2 The `Chromatogram` Class 60
5.4.3 Read the `scf` File 63
5.4.4 Set Chromatogram Parameters 66
5.4.5 Print the Chromatogram 68
5.5 `pgfmolbio.domains.tex` 75
5.5.1 Keys 75
5.5.2 Feature Shapes 77
5.5.3 Secondary Structure Elements 82
5.5.4 Adding Features 88
5.5.5 The Main Environment 90
5.5.6 Feature Styles 93
5.6 `pgfmolbio.domains.lua` 96
5.6.1 Predefined Feature Print Functions 97
5.6.2 The `SpecialKeys` Class 99
5.6.3 The `Protein` Class 102
5.6.4 Uniprot and GFF Files 103
5.6.5 Getter and Setter Methods 106
5.6.6 Adding Feature 111
5.6.7 Calculate Disulfide Levels 112
5.6.8 Print Domains 113
5.6.9 Converting a `Protein` to a String 117
5.7 `pgfmolbio.convert.tex` 119

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, protein domain diagrams, plasmid maps 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 the current development release, pgfmolbio is able to

- read DNA sequencing files in standard chromatogram format (scf) and draw the corresponding chromatogram;
- read protein domain information from Uniprot or general feature format files (gff) and draw domain diagrams.

To this end, pgfmolbio relies on routines from pgf's Ti*k*Z frontend and on the Lua scripting language implemented in LuaTEX. Consequently, the package will not work directly with traditional engines like pdfTEX. However, a converter module ensures a high degree of backward compatibility.

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

[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.

unable to predict when (and if) additional functions become available. Nevertheless, I would greatly appreciate any comments or suggestions.

1.2 Getting Started

Before you consider using `pgfmolbio`, please make sure that both your LuaTEX (at least 0.70.2) 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, three ⟨*module*⟩s are available:

- chromatogram (chapter 2) allows you to draw DNA sequencing chromatograms obtained by the Sanger sequencing method.
- domains (chapter 3) provides macros for drawing protein domain diagrams and is also able to read domain information from files in Uniprot or general feature format.
- Furthermore, convert (chapter 4) is used with one of the modules above and generates "pure" Ti*k*Z code suitable for TEX engines lacking Lua support.

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

Fine-tunes the graphs produced by each `pgfmolbio` module. The possible keys are described in the sections on the respective modules.

2 The chromatogram module

2.1 Overview

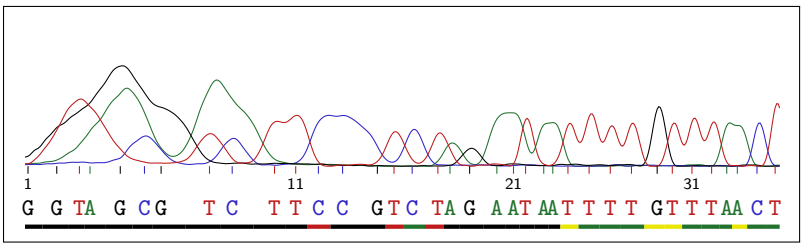
The chromatogram module draws DNA sequencing chromatograms stored in standard chromatogram format (`scf`), which was developed by Simon Dear and Rodger Staden[1]. The documentation for the Staden package[2] describes the current version of the `scf` format in detail. As far as they are crucial to understanding the Lua code, we will discuss some details of this file format in the documented source code (section 5.4). 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

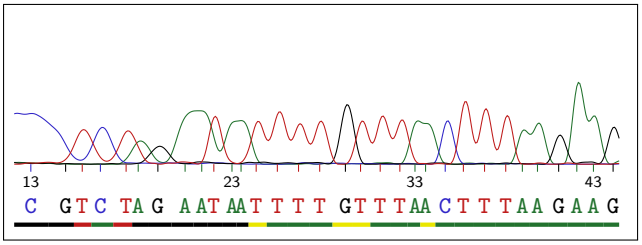
/pgfmolbio/chromatogram/sample range =⟨*lower*⟩-⟨*upper*⟩[step ⟨*int*⟩]

Default: 1-500 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 - 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

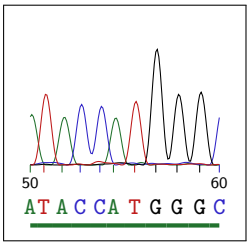



```
\pmbchromatogram[sample range=200-600]{SampleScf.scf}
```

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.

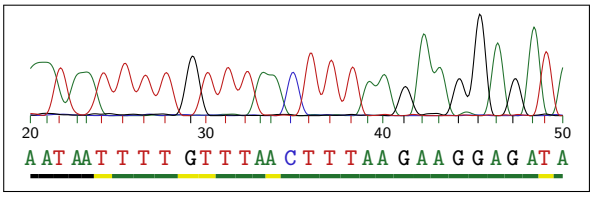
The optional third part of the value for sample range orders the package to draw every ⟨*int*⟩th sample point. If your document contains large chromatograms or a great number of them, drawing fewer sample points increases typesetting time at the cost of image quality (Example 2.4). Nevertheless, the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

Example 2.3

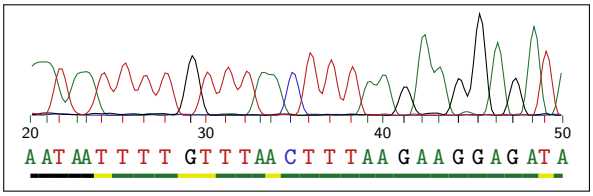



```
\pmbchromatogram[%
    sample range=base 50-base60
  ]{SampleScf.scf}
```

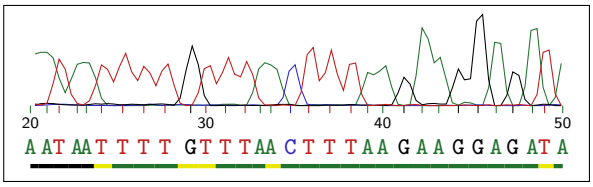
Example 2.4




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




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




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

2.4 General Layout

/pgfmolbio/chromatogram/x unit =⟨*dimension*⟩

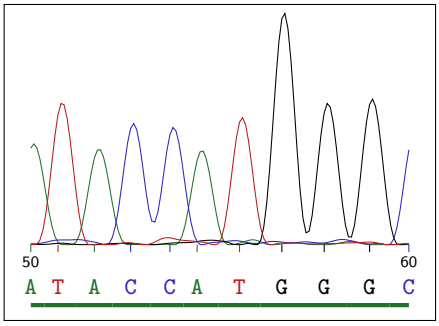
Default: `0.2mm`

/pgfmolbio/chromatogram/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-base 60,
    x unit=0.4mm,
    y unit=0.02mm
  ]{SampleScf.scf}
```

/pgfmolbio/chromatogram/samples per line =⟨*number*⟩

Default: `500`

/pgfmolbio/chromatogram/baseline skip =⟨*dimension*⟩

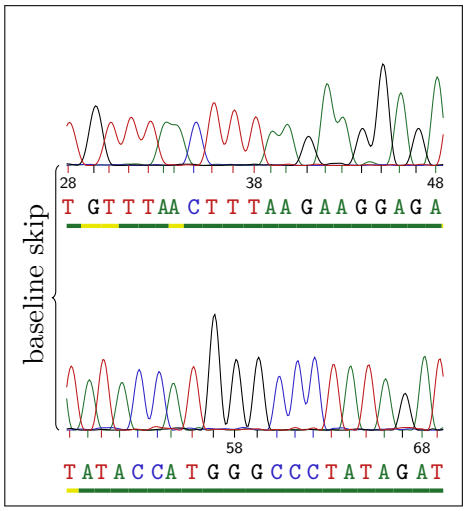
Default: `3cm`

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.

/pgfmolbio/chromatogram/canvas style /.style=⟨*style*⟩

Default: `draw=none, fill=none`

Example 2.6



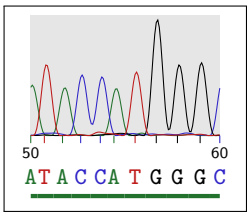

```
\begin{tikzpicture}%
    [decoration=brace]
  \pmbchromatogram[%
      sample range=401-900,
      samples per line=250,
      baseline skip=3.5cm
    ]{SampleScf.scf}
  \draw[decorate]
    (-0.1cm, -3.5cm) -- (-0.1cm, 0cm)
    node[pos=0.5, rotate=90, above=5pt]
      {baseline skip};
\end{tikzpicture}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

/pgfmolbio/chromatogram/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-base 60,
    canvas style/.style={draw=none, fill=black!10},
    canvas height=1.6cm
  ]{SampleScf.scf}
```

2.5 Traces

/pgfmolbio/chromatogram/trace A style /.style=⟨*style*⟩

Default: pmbTraceGreen

/pgfmolbio/chromatogram/trace C style /.style=⟨*style*⟩

Default: pmbTraceBlue

/pgfmolbio/chromatogram/trace G style /.style=⟨*style*⟩

Default: pmbTraceBlack

/pgfmolbio/chromatogram/trace T style /.style=⟨*style*⟩

Default: pmbTraceRed

/pgfmolbio/chromatogram/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*⟩

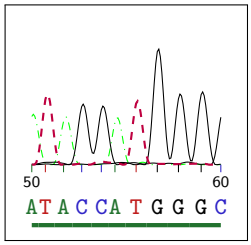
basewise, whereas `trace style` changes all styles simultaneously. Note the syntax differences between `trace style` and `trace A style` etc. The standard styles simply color the traces; Table 2.1 lists the color specifications.

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	

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



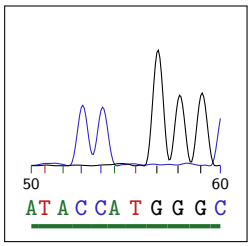

```
\pmbchromatogram[%
    sample range=base 50-base 60,
    trace style=thin,
    trace A style/.append style={dashdotted, green},
    trace T style/.style={thick, dashed, purple}
  ]{SampleScf.scf}
```

`/pgfmolbio/chromatogram/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-base 60,
    traces drawn=CG
  ]{SampleScf.scf}
```

2.6 Ticks

`/pgfmolbio/chromatogram/tick A style /.style=`⟨*style*⟩

Default: `thin, pmbTraceGreen`

`/pgfmolbio/chromatogram/tick C style /.style=`⟨*style*⟩

Default: `thin, pmbTraceBlue`

`/pgfmolbio/chromatogram/tick G style /.style=`⟨*style*⟩

Default: `thin, pmbTraceBlack`

`/pgfmolbio/chromatogram/tick T style /.style=`⟨*style*⟩

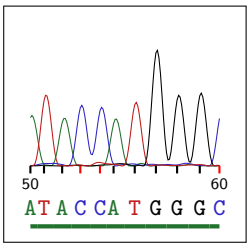
Default: `thin, pmbTraceRed`

`/pgfmolbio/chromatogram/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. Note the syntax differences between `tick style` and `tick A style` etc. Example 2.10 illustrates how one can draw thick ticks, which are red if they indicate a cytosine peak.

Example 2.10



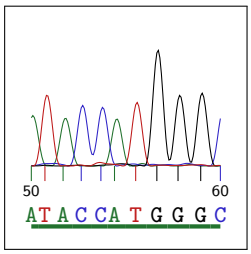

```
\pmbchromatogram[%
    sample range=base 50-base 60,
    tick style=thick,
    tick C style/.append style={red}
  ]{SampleScf.scf}
```

`/pgfmolbio/chromatogram/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



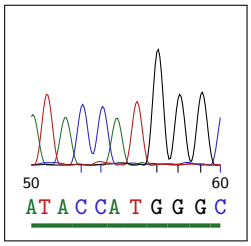

```
\pmbchromatogram[%
    sample range=base 50-base 60,
    tick length=2mm
  ]{SampleScf.scf}
```

/pgfmolbio/chromatogram/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-base 60,
    ticks drawn=CG
  ]{SampleScf.scf}
```

2.7 Base Labels

/pgfmolbio/chromatogram/base label A text =⟨*text*⟩

Default: \strut A

/pgfmolbio/chromatogram/base label C text =⟨*text*⟩

Default: \strut C

/pgfmolbio/chromatogram/base label G text =⟨*text*⟩

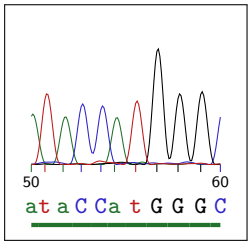
Default: \strut G

/pgfmolbio/chromatogram/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-base 60,
    base label A text=\strut a,
    base label T text=\strut t
  ]{SampleScf.scf}
```

/pgfmolbio/chromatogram/base label A style /.style=⟨*style*⟩

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

/pgfmolbio/chromatogram/base label C style /.style=⟨*style*⟩

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

/pgfmolbio/chromatogram/base label G style /.style=⟨*style*⟩

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

/pgfmolbio/chromatogram/base label T style /.style=⟨*style*⟩

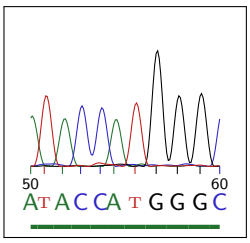
Default: `below=4pt, font=\ttfamily\footnotesize, pmbTraceRed`

/pgfmolbio/chromatogram/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



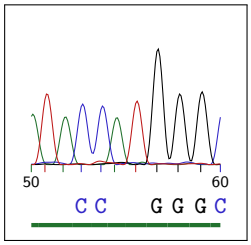

```
\pmbchromatogram[%
    sample range=base 50-base 60,
    base label style=%
      {below=2pt, font=\sffamily\footnotesize},
    base label T style/.append style=%
      {below=4pt, font=\tiny}
  ]{SampleScf.scf}
```

/pgfmolbio/chromatogram/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-base 60,
    base labels drawn=CG
  ]{SampleScf.scf}
```

2.8 Base Numbers

/pgfmolbio/chromatogram/show base numbers =⟨*boolean*⟩

Default: `true`

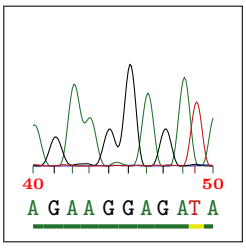
Turns the *base numbers* on or off, which indicate the indices of the base peaks below the traces.

/pgfmolbio/chromatogram/base number style /.style=⟨*style*⟩

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

Determines the placement and appearance of the base numbers. Example 2.16 contains bold red base numbers that are shifted slightly upwards.

Example 2.16



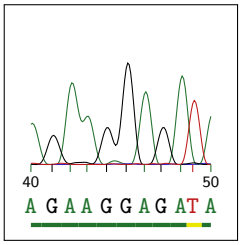

```
\pmbchromatogram[%
    sample range=base 40-base 50,
    base number style/.style={below=-3pt,%
      font=\rmfamily\bfseries\tiny, red}
  ]{SampleScf.scf}
```

/pgfmolbio/chromatogram/**base number range** =⟨*lower*⟩-⟨*upper*⟩[step ⟨*interval*⟩]

Default: `auto-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-base 50,
    base number range=42-46 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.

/pgfmolbio/chromatogram/**probability distance** =⟨*dimension*⟩

Default: `0.8cm`

Sets the distance between the base probability rules and the baseline.

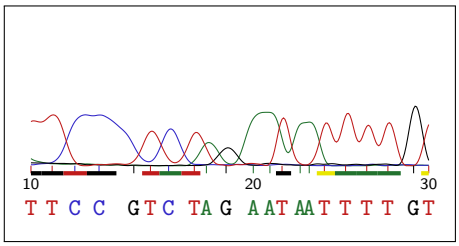
[3] Ewing, B., Hillier, L., Wendl, M. C., and Green, P. (1998). Base-calling of automated sequencer traces using phred. I. Accuracy assessment. *Genome Res.* **8**(3), 175–185.

/pgfmolbio/chromatogram/probabilities drawn =A|C|G|T|any combination thereof

Default: ACGT

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-base 30,
    probabilities drawn=CT,
    probability distance=1mm
  ]{SampleScf.scf}
```

/pgfmolbio/chromatogram/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.

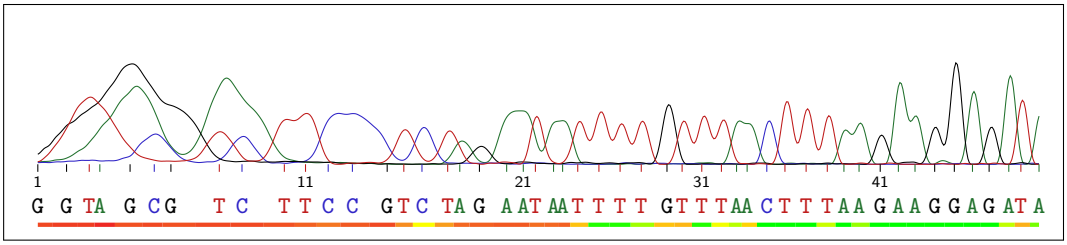
2.10 Miscellaneous Keys

/pgfmolbio/chromatogram/bases drawn =A|C|G|T|any combination thereof

Default: ACGT

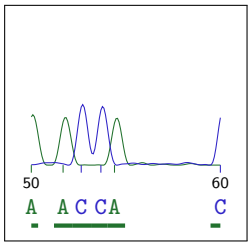
This key simultaneously sets traces drawn, ticks drawn, base labels drawn and probabilities drawn (see Example 2.20).

Example 2.19




```
\directlua{
  function probabilityGradient (prob)
    local minProb, maxProb = pmbChromatogram: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-base 50,
    probability style function=probabilityGradient
  ]{SampleScf.scf}
```

Example 2.20




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

3 The domains module

3.1 Overview

Protein domain diagrams appear frequently in databases such as Pfam[1] or PROSITE[2]. Domain diagrams are often drawn using standard graphics software or tools such as PROSITE's MyDomains image creator[3]. However, the domains module provides an integrated approach for generating domain diagrams from TEX code or from external files.

3.2 Domain Diagrams and Their Features

```
\begin{pmbdomains}[⟨key-value list⟩]{⟨sequence length⟩}
  ⟨features⟩
\end{pmbdomains}
```

Draws a domain diagram with the ⟨*features*⟩ given. The ⟨*key-value list*⟩ configures its appearance. ⟨*sequence length*⟩ is the total number of residues in the protein. (Although you must eventually specify a sequence length, you may actually leave the mandatory argument empty and use the `sequence length` key instead; see section 3.10).

You can put a `pmbdomains` environment into a `tikzpicture`, but you also may use the environment on its own. `pgfmolbio` checks whether it is surrounded by a `tikzpicture` and adds this environment if necessary.

```
/pgfmolbio/domains/name =⟨text⟩
```

Default: `Protein`

The name of the protein, which usually appears centered above the diagram.

[1] Finn, R. D., Mistry, J. *et al.* (2010). The Pfam protein families database. *Nucleic Acids Res.* **38**, D211–D222.

[2] Sigrist, C. J. A., Cerutti, L. *et al.* (2010). PROSITE, a protein domain database for functional characterization and annotation. *Nucleic Acids Res.* **38**, D161–D166.

[3] `http://prosite.expasy.org/mydomains/`

`/pgfmolbio/domains/show name =⟨boolean⟩`

Default: `true`

Determines whether both the name and sequence length are shown.

`\addfeature[⟨key-value list⟩]{⟨type⟩}{⟨start⟩}{⟨stop⟩}`

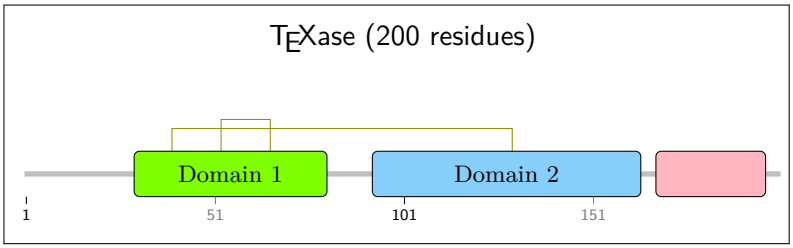
Adds a feature of the given ⟨*type*⟩ to the current domain diagram (only defined inside `pmbdomains`). The feature spans the residues from ⟨*start*⟩ to ⟨*stop*⟩. These arguments are either numbers, which refer to residues in the relative numbering scheme, or numbers in parentheses, which refer to absolute residue numbers (see section 3.3).

`/pgfmolbio/domains/description =⟨text⟩`

Default: (none)

Sets the feature description (Example 3.1).

Example 3.1




```
\begin{tikzpicture} % optional
  \begin{pmbdomains}[name=\TeX ase]{200}
    \addfeature{disulfide}{40}{129}
    \addfeature{disulfide}{53}{65}
    \addfeature[description=Domain 1]{domain}{30}{80}
    \addfeature[description=Domain 2]{domain}{93}{163}
    \addfeature{domain}{168}{196}
  \end{pmbdomains}
\end{tikzpicture} % optional
```

3.3 General Layout

`/pgfmolbio/domains/x unit =`⟨*dimension*⟩

Default: `0.5mm`

The width of a single residue.

`/pgfmolbio/domains/y unit =`⟨*dimension*⟩

Default: `6mm`

The height of a default `domain` feature.

`/pgfmolbio/domains/residues per line =`⟨*number*⟩

Default: `200`

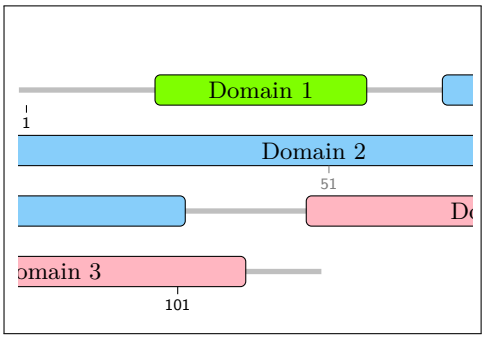
A new domain diagram "line" starts after ⟨*number*⟩ residues.

`/pgfmolbio/domains/baseline skip =`⟨*factor*⟩

Default: `3`

The baselines of consecutive lines (i. e., the main chain y-coordinates) are separated by ⟨*factor*⟩ times the value of `y unit`. In Example 3.2, you see four lines, each of which contains up to 30 residues. Note how domains are correctly broken across lines. Furthermore, the baselines are $2 \times 4 = 8$ mm apart.

Example 3.2




```
\begin{pmbdomains}%
    [show name=false, x unit=2mm, y unit=4mm,
    residues per line=30, baseline skip=2]{110}
  \addfeature[description=Domain 1]{domain}{10}{23}
  \addfeature[description=Domain 2]{domain}{29}{71}
  \addfeature[description=Domain 3]{domain}{80}{105}
\end{pmbdomains}
```

/pgfmolbio/domains/residue numbering =⟨*numbering scheme*⟩

Default: `auto`

A protein's amino acid residues are usually numbered consecutively starting from 1. However, there are different numbering schemes. For example, residue numbering in a serine protease related to chymotrypsin typically follows the numbering in chymotrypsinogen[4]. The target protease sequence is aligned to the chymotrypsinogen sequence, and equivalent residues receive the same number. Insertions into the target sequence are indicated by appending letters to the last aligned residue (e. g., 186, *186A*, *186B*, 187), whereas gaps in the target sequence cause gaps in the numbering (e. g., 124, 125, 128, 129).

In `pgfmolbio`, you can specify a relative ⟨*numbering scheme*⟩ via the `residue numbering` key. The keyword `auto` indicates that residues are numbered from 1 to (sequence length), i. e. absolute and relative numberings coincide. This is the case in all examples above. The complete syntax for the key is

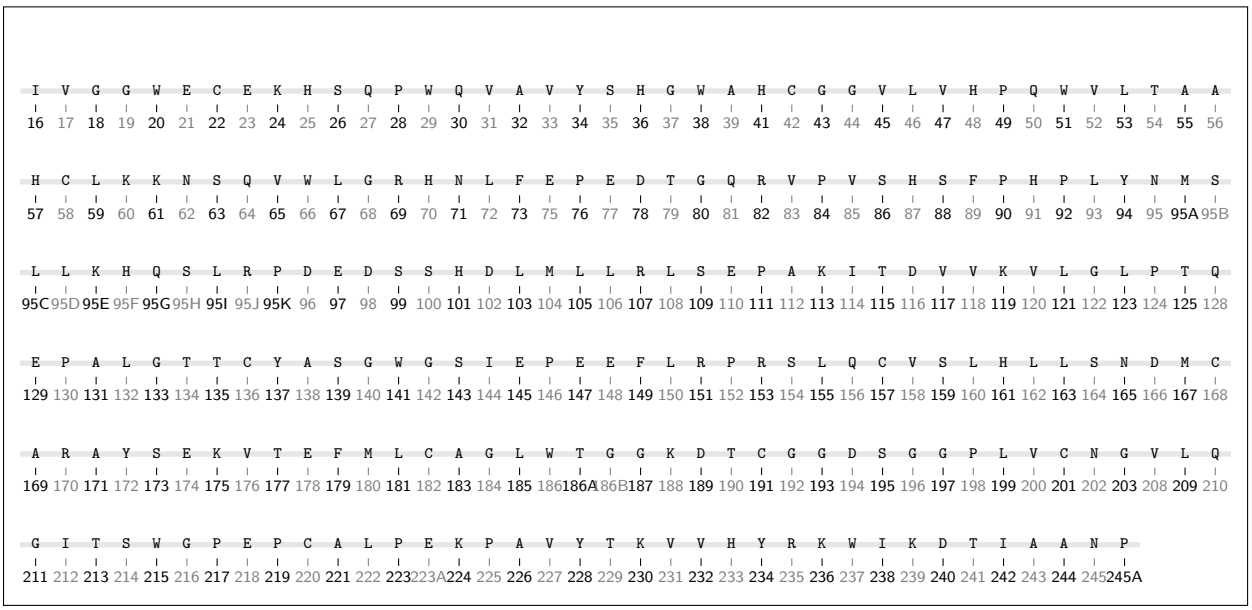
⟨*numbering scheme*⟩ := `{`⟨*range*⟩`[,`⟨*range*⟩`,...]``}`
⟨*range*⟩ := ⟨*start*⟩`-`⟨*end*⟩ | ⟨*start*⟩
⟨*start*⟩ := ⟨*number*⟩ | ⟨*number*⟩⟨*letter*⟩
⟨*end*⟩ := ⟨*number*⟩ | ⟨*letter*⟩

Example 3.3 shows a custom ⟨*numbering scheme*⟩, in this case for kallikrein-related peptidase 2 (KLK2), a chymotrypsin-like serine proteases. (In the following explanation, the subscripts 'abs' and 'rel' denote absolute and relative numbering, respectively).

- Residue 1_{abs} is labeled 16_{rel}, residue 2_{abs} is labeled 17_{rel} etc. until residue 24_{abs}, which is labeled 39_{rel} (range `16-39`).
- Residue 25_{abs} corresponds to 41_{rel} etc. until residue $57_{\text{abs}}/73_{\text{rel}}$ (range `41-73`).
- Residue 40_{rel} is missing – no residue in KLK2 is equivalent to residue 40 in chymotrypsinogen.
- An insertion of 11 amino acids follows residue 95_{rel}. These residues are numbered from 95A_{rel} to 95K_{rel}. Note that both `95A-K` and `95A-95K` are valid ranges.
- The number of the last residue is 245A_{rel}(range `245A`).

[4]Bode, W., Mayr, I. *et al.* (1989). The refined 1.9 Å crystal structure of human α-thrombin: interaction with D-Phe-Pro-Arg chloromethylketone and significance of the Tyr-Pro-Pro-Trp insertion segment. *EMBO J.* **8**(11), 3467–3475.

Example 3.3




```
\begin{pmbdomains}[%
    sequence=IVGGWECEKHSQPWQVAVYSHGWAHCGGVLVHPQWVLTAAHCLK%
      KNSQVWLGRHNLFEPEDTGQRVPVSHSFPHPLYNMSLLKHQSLRPDEDSSH%
      DLMLLRLSEPAKITDVVKVLGLPTQEPALGTTCYASGWGSIEPEEFLRPRS%
      LQCVSLHLLSNDMCARAYSEKVTEFMLCAGLWTGGKDTCGGDSGGPLVCNG%
      VLQGITSWGPEPCALPEKPAVYTKVVHYRKWIKDTIAANP,
    residue numbering={16-39,41-73,75-95,95A-K,96-125,%
      128-186,186A-186B,187-203,208-223,223A,224-245,245A},
    x unit=4mm,
    residues per line=40,
    show name=false,
    ruler range=auto-auto step 1,
    ruler distance=-.3,
    baseline skip=2
  ]{237}
  \setfeaturestyle{other/main chain}{*1{draw, line width=2pt, black!10}}
  \addfeature{other/sequence}{16}{245A}
\end{pmbdomains}
```

/pgfmolbio/domains/residue range =⟨*lower*⟩-⟨*upper*⟩

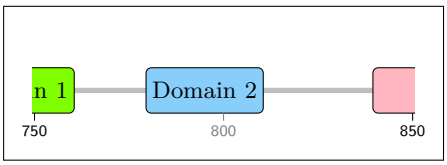
Default: `auto-auto`

All residues from ⟨*lower*⟩ to ⟨*upper*⟩ will appear in the output. Possible values for ⟨*lower*⟩ and ⟨*upper*⟩ are:

- `auto`, which indicates the first or last residue, respectively;
- a plain number, which denotes a residue in the *relative* numbering scheme set by `residue numbering`;
- a parenthesized number, which denotes a residue in the *absolute* numbering scheme.

In Example 3.4, only residues 650_{abs} to 850_{rel} are shown. If a domain boundary lies outside of the range shown, only the appropriate part of the domain appears.

Example 3.4




```
\begin{pmbdomains}[%
    show name=false, residue range=(650)-850,
    residue numbering={1-500,601-1100}]{1000}
  \addfeature[description=Domain 1]{domain}{(630)}{(660)}
  \addfeature[description=Domain 2]{domain}{(680)}{(710)}
  \addfeature[description=Domain 3]{domain}{840}{1000}
  \addfeature[description=Domain 4 (invisible)]{domain}{1010}{1040}
\end{pmbdomains}
```

/pgfmolbio/domains/enlarge left =⟨*dimension*⟩

Default: `0cm`

/pgfmolbio/domains/enlarge right =⟨*dimension*⟩

Default: `0cm`

/pgfmolbio/domains/enlarge top =⟨*dimension*⟩

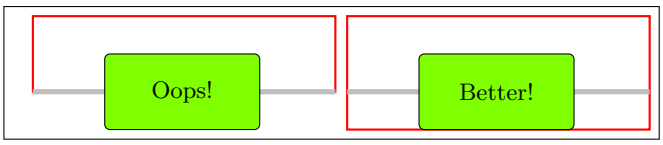
Default: `1cm`

/pgfmolbio/domains/enlarge bottom =⟨*dimension*⟩

Default: 0cm

pgfmolbio clips features that would protrude into the left or right margin. However, limits in the Ti*k*Z clipping mechanism prevent correct automatic updates of the bounding box for the domain diagram. Although the package tries hard to establish a bounding box that is sufficiently large, the process may require manual intervention. To this end, each enlarge ... key enlarges the bounding box at the respective side (Example 3.5).

Example 3.5




```
\tikzset{%
  baseline, tight background,%
  background rectangle/.style={draw=red, thick}%
}
\pgfmolbioset[domains]{show name=false, y unit=1cm, show ruler=false}

\begin{tikzpicture}[show background rectangle]
  \begin{pmbdomains}{80}
    \addfeature[description=Oops!]{domain}{20}{60}
  \end{pmbdomains}
\end{tikzpicture}
\begin{tikzpicture}[show background rectangle]
  \begin{pmbdomains}[enlarge bottom=-5mm]{80}
    \addfeature[description=Better!]{domain}{20}{60}
  \end{pmbdomains}
\end{tikzpicture}
```

3.4 Feature Styles and Shapes

Each (implicit and explicit) feature of a domain chart has a certain *shape* and *style*. For instance, you can see five different feature *shapes* in Example 3.1: We explicitly added two features of shape (and type) disulfide and three features of shape domain. Furthermore, the package implicitly included features of shape other/name, other/main chain and other/ruler.

Although the three domain features agree in shape, they differ in color, or (more generally) *style*. Since pgfmolbio distinguishes between shapes and styles, you may draw equally shaped features with different colors, strokes, shadings etc.

`\setfeaturestyle{⟨type⟩}{⟨style list⟩}`

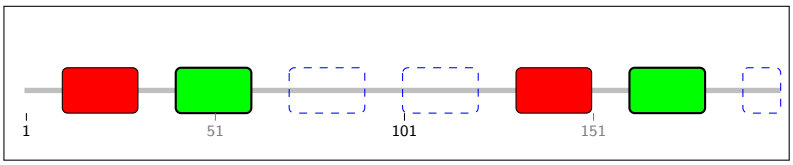
Specifies a ⟨*style list*⟩ for the given feature ⟨*type*⟩. The complete syntax ist

⟨*style list*⟩ := {⟨*style list item*⟩[,⟨*style list item*⟩,...]}
⟨*style list item*⟩ := ⟨*multiplier*⟩⟨*style*⟩
⟨*multiplier*⟩ := [*⟨*number*⟩]
⟨*style*⟩ := ⟨*single key-value pair*⟩ | {⟨*key-value list*⟩}

A style list item of the general form *⟨*n*⟩{⟨*style*⟩} instructs the package to repeat the ⟨*style*⟩ ⟨*n*⟩-times. (This syntax is reminiscent of column specifications in a `tabular` environment. However, do *not* enclose numbers with more than one digit in curly braces!) You may omit the trivial multiplier `*1`, but never forget the curly braces surrounding a ⟨*style*⟩ that contains two or more key-value pairs. Furthermore, `pgfmolbio` loops over the style list until all features have been drawn.

For instance, the style list in Example 3.6 fills the first feature red, then draws a green one with a thick stroke, and finally draws two dashed blue features.

Example 3.6



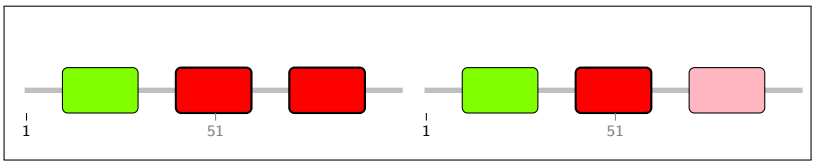

```
\begin{pmbdomains}[show name=false]{200}
  \setfeaturestyle{domain}%
    {fill=red, {thick, fill=green}, *2{blue, dashed}}
  \addfeature{domain}{11}{30}
  \addfeature{domain}{41}{60}
  \addfeature{domain}{71}{90}
  \addfeature{domain}{101}{120}
  \addfeature{domain}{131}{150}
  \addfeature{domain}{161}{180}
  \addfeature{domain}{191}{200}
\end{pmbdomains}
```

`/pgfmolbio/domains/style =`⟨*style*⟩

Default: (empty)

Although `\setfeaturestyle` may appear in a `pmbdomains` environment, changes introduced in this way are not limited to the current TEX group (since feature styles are stored in Lua variables). Instead, use the `style` key to locally override a feature style (Example 3.7).

Example 3.7




```
\begin{pmbdomains}[show name=false]{100}
  \addfeature{domain}{11}{30}
  \begingroup
    \setfeaturestyle{domain}{{thick, fill=red}}
    \addfeature{domain}{41}{60}
  \endgroup
  \addfeature{domain}{71}{90} % the new style persists ...
\end{pmbdomains}

\begin{pmbdomains}[show name=false]{100}
  \addfeature{domain}{11}{30}
  \addfeature[style={thick, fill=red}]{domain}{41}{60}
  \addfeature{domain}{71}{90} % correct solution
\end{pmbdomains}
```

`\setfeaturestylealias{`⟨*new type*⟩`}{`⟨*existing type*⟩`}`

After calling this macro, the ⟨*new type*⟩ and ⟨*existing type*⟩ share a common style, while they still differ in their shapes.

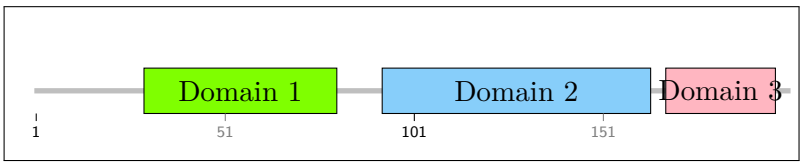
`\setfeatureshape{`⟨*type*⟩`}{`⟨*TikZ code*⟩`}`

Defines a new feature shape named ⟨*type*⟩ or changes an existing one. **Caution:** If you change a shape within `pmbdomains`, you will also change the features of equal type that you already added. Thus, it is best to use `\setfeatureshape` only outside of this environment.

Several commands that are only available in the ⟨*TikZ code*⟩ allow you to design generic feature shapes:

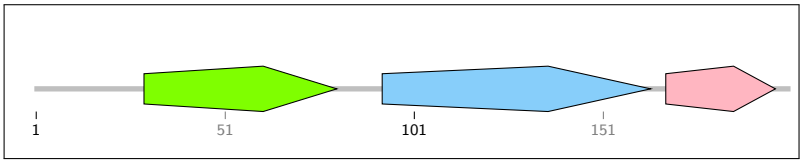
- `\xLeft`, `\xMid` and `\xRight` expand to the left, middle and right x-coordinate of the feature. The coordinates are in a format suitable for `\draw` and similar commands.
- `\yMid` expands to the y-coordinate of the feature, i. e. the y-coordinate of the current line.
- You can access any values stored in the package's ⟨*key*⟩s with the macro `\pmbdomvalueof{`⟨*key*⟩`}`.

Example 3.8



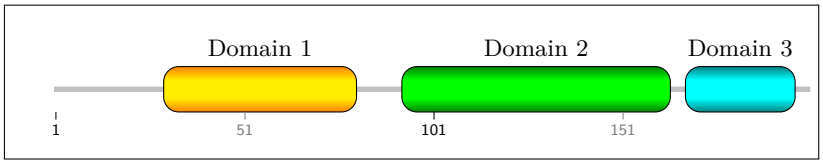

```
\setfeatureshape{domain}{%
  \draw [/pgfmolbio/domains/current style]
    (\xLeft, \yMid + .5 * \pmbdomvalueof{y unit}) rectangle
    (\xRight, \yMid - .5 * \pmbdomvalueof{y unit});
  \node at (\xMid, \yMid) {\pmbdomvalueof{description}};
}

\begin{pmbdomains}[show name=false]{200}
  \addfeature[description=Domain 1]{domain}{30}{80}
  \addfeature[description=Domain 2]{domain}{93}{163}
  \addfeature[description=Domain 3]{domain}{168}{196}
\end{pmbdomains}
```

Example 3.9




```
\setfeatureshape{domain}{%
  \pgfmathsetmacro\middlecorners{%
    \xLeft + (\xRight - \xLeft) * .618%
  }
  \draw [/pgfmolbio/domains/current style]
    (\xLeft, \yMid + 2mm) --
    (\middlecorners pt, \yMid + 3mm) --
    (\xRight, \yMid) --
    (\middlecorners pt, \yMid - 3mm) --
    (\xLeft, \yMid - 2mm) --
    cycle;
}

\begin{pmbdomains}[show name=false]{200}
  \addfeature[description=Domain 1]{domain}{30}{80}
  \addfeature[description=Domain 2]{domain}{93}{163}
  \addfeature[description=Domain 3]{domain}{168}{196}
\end{pmbdomains}
```

Example 3.10




```latex
\pgfdeclareverticalshading[bordercolor,middlecolor]{mydomain}{100bp}{
  color(0bp)=(bordercolor);
  color(25bp)=(bordercolor);
  color(40bp)=(middlecolor);
  color(60bp)=(middlecolor);
  color(75bp)=(bordercolor);
  color(100bp)=(bordercolor)
}

\tikzset{%
  domain middle color/.code=\colorlet{middlecolor}{#1},%
  domain border color/.code=\colorlet{bordercolor}{#1}%
}

\setfeatureshape{domain}{%
  \draw [shading=mydomain, rounded corners=2mm,
    /pgfmolbio/domains/current style]
    (\xLeft, \yMid + .5 * \pmbdomvalueof{y unit}) rectangle
    (\xRight, \yMid - .5 * \pmbdomvalueof{y unit});
  \node [above=3mm] at (\xMid, \yMid)
    {\pmbdomvalueof{domain font}{\pmbdomvalueof{description}}};
}

\begin{pmbdomains}[show name=false]{200}
  \setfeaturestyle{domain}{%
    {domain middle color=yellow!85!orange,%
    domain border color=orange},%
    {domain middle color=green,%
    domain border color=green!50!black}%
    {domain middle color=cyan,%
    domain border color=cyan!50!black}%
  }
  \addfeature[description=Domain 1]{domain}{30}{80}
  \addfeature[description=Domain 2]{domain}{93}{163}
  \addfeature[description=Domain 3]{domain}{168}{196}
\end{pmbdomains}
```

- The style key `/pgfmolbio/domains/current style` represents the current feature style selected from the associated style list.

The commands above are available for all features. By contrast, the following macros are limited to certain feature types:

- `\featureSequence` provides the amino acid sequence of the current feature. This macro is only available for explicitly added features and for `other/main chain`.
- `\residueNumber` equals the current residue number. This macro is only available for shape `other/ruler` (see section 3.7).
- `\currentResidue` expands to a single letter amino acid abbreviation. This macro is only available for shape `other/sequence` (see section 3.8).

In Example 3.8, we develop a simple `domain` shape, which is a rectangle containing a centered label with the feature description. Example 3.9 calculates an additional coordinate for a pentagonal domain shape and stores this coordinate in `\middlecorners`. Note that you have to insert "pt" after `\middlecorners` when using the stored coordinate. The domains in Example 3.10 display a custom shading and inherit their style from the style list.

`\setfeatureshapealias{`⟨*new type*⟩`}{`⟨*existing type*⟩`}`

After calling this macro, the ⟨*new type*⟩ and ⟨*existing type*⟩ share a common shape, while they still differ in their styles.

`\setfeaturealias{`⟨*new type*⟩`}{`⟨*existing type*⟩`}`

This is a shorthand for calling both `\setfeatureshape` and `\setfeaturestyle`.

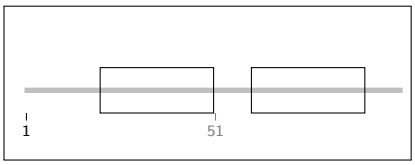
3.5 Standard Features

`pgfmolbio` provides a range of standard features. This section explains simple features (i. e., those that support no or only few options), while later sections cover advanced ones. Some features include predefined aliases, which facilitate inclusion of external files (see section 3.10).

Feature `default` (*no alias*)
A fallback for undefined features, in which case TEX issues a warning (Example 3.11).

Feature `domain` (*alias* `DOMAIN`)
A generic feature for protein domains. It consists of a rectangle with rounded corners and a label in the center, which shows the value of `description`.

Example 3.11



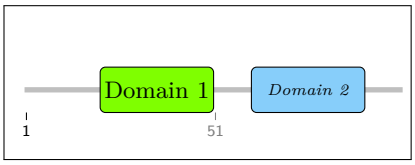

```
\begin{pmbdomains}[show name=false]{100}
  \addfeature{default}{21}{50}
  \addfeature{unknown}{61}{90} % i.e. default shape/style
\end{pmbdomains}
```

/pgfmolbio/domains/**domain font** =⟨*font commands*⟩

Default: `\footnotesize`

Sets the font for the label of a `domain` feature. The last command may take a single argument (Example 3.12).

Example 3.12



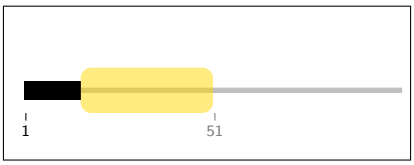

```
\begin{pmbdomains}[show name=false]{100}
  \addfeature[description=Domain 1]{domain}{21}{50}
  \addfeature[description=Domain 2,%
    domain font=\tiny\textit]{DOMAIN}{61}{90}
\end{pmbdomains}
```

Feature `signal peptide` (*alias* `SIGNAL`)
Adds a signal peptide (Example 3.13).

Feature `propeptide` (*alias* `PROPEP`)
Adds a propeptide (Example 3.13).

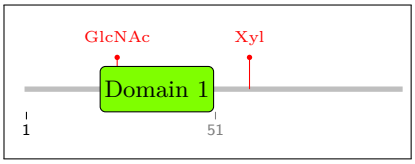
Feature `carbohydrate` (*alias* `CARBOHYD`)
Adds glycosylation (Example 3.14).

Example 3.13




```
\begin{pmbdomains}[show name=false]{100}
  \addfeature{signal peptide}{1}{15}
  \addfeature{propeptide}{16}{50}
\end{pmbdomains}
```

Example 3.14




```
\begin{pmbdomains}[show name=false]{100}
  \addfeature[description=GlcNAc]{carbohydrate}{25}{25}
  \addfeature[description=Xyl]{CARBOHYD}{60}{60}
  \addfeature[description=Domain 1]{domain}{21}{50}
\end{pmbdomains}
```

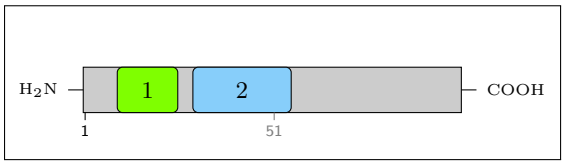
Feature `other/main chain` (*no alias*)
This feature is automatically added to the feature list at the end of each `pmbdomains` environment. It represents the protein main chain, which appears as a grey line by default. Nevertheless, you can alter the backbone just like any other feature (Example 3.15).

Feature `other/name` (*no alias*)
This feature is automatically added to the feature list at the end of each `pmbdomains` environment. It relates to the protein name, which is normally displayed at the top center of the chart, together with the number of residues (Example 3.16). The following auxiliary commands are available for the feature style Ti*k*Z code: `\xLeft`, `\xMid`, `\xRight` and `current style`.

3.6 Disulfides and Ranges

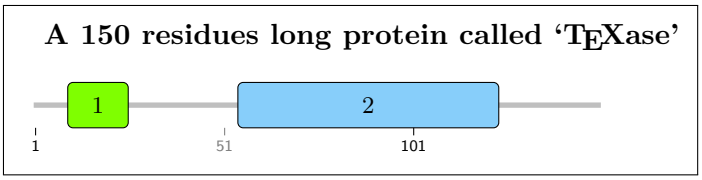
Feature `disulfide` (*alias* `DISULFID`)
`pgfmolbio` indicates disulfide bridges by brackets above the main chain. Since disulfides are often interleaved in linear representations of proteins, the package automatically stacks them in order to avoid overlaps (Example 3.17).

Example 3.15




```
\setfeatureshape{other/main chain}{%
  \draw [/pgfmolbio/domains/current style]
    (\xLeft, \yMid + .5 * \pmbdomvalueof{y unit}) rectangle
    (\xRight, \yMid - .5 * \pmbdomvalueof{y unit});
  \draw (\xLeft, \yMid) --
    (\xLeft - 2mm, \yMid)
    node [left] {\tiny H$_2$N};
  \draw (\xRight, \yMid) --
    (\xRight + 2mm, \yMid)
    node [right] {\tiny COOH};
}
\begin{pmbdomains}%
    [show name=false, enlarge left=-0.8cm, enlarge right=1.2cm]{100}
  \setfeaturestyle{other/main chain}{{draw=black,fill=black!20}}
  \addfeature[description=1]{domain}{10}{25}
  \addfeature[description=2]{domain}{30}{55}
\end{pmbdomains}
```

Example 3.16




```
\setfeatureshape{other/name}{%
  \node [/pgfmolbio/domains/current style]
    at (\xLeft, \pmbdomvalueof{baseline skip}
      * \pmbdomvalueof{y unit} / 2)
    {A \pmbdomvalueof{sequence length} residues long protein
      called `\pmbdomvalueof{name}'};
}
\begin{pmbdomains}[name=\TeX ase]{150}
  \setfeaturestyle{other/name}{{font=\bfseries, right}}
  \addfeature[description=1]{domain}{10}{25}
  \addfeature[description=2]{domain}{55}{123}
\end{pmbdomains}
```

`/pgfmolbio/domains/level =`⟨*number*⟩

Default: (empty)

Manually sets the level of a disulfide feature.

`/pgfmolbio/domains/disulfide base distance =`⟨*number*⟩

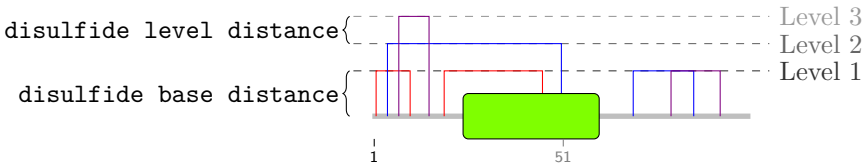
Default: `1`

The distance (as a multiple of y-units) between the main chain and the first level.

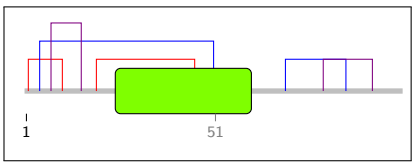
`/pgfmolbio/domains/disulfide level distance =`⟨*number*⟩

Default: `.2`

The space (as a multiple of y-units) between levels (see the figure below).



Example 3.17




```
\begin{pmbdomains}[show name=false,
    disulfide base distance=.7,
    disulfide level distance=.4]{100}
  \setfeaturestyle{disulfide}{draw=red, draw=blue, draw=violet}
  \addfeature{disulfide}{2}{10}
  \addfeature{disulfide}{5}{50}
  \addfeature{disulfide}{8}{15}
  \addfeature{disulfide}{20}{45}
  \addfeature[level=1]{disulfide}{70}{85}
  \addfeature[level=1]{disulfide}{80}{92}
  \addfeature{domain}{25}{60}
\end{pmbdomains}
```

```
\setdisulfidefeatures{⟨key list⟩}
\adddisulfidefeatures{⟨key list⟩}
\removedisulfidefeatures{⟨key list⟩}
```

These macros edit the list of "disulfide-like" features, i. e. those subject to the automatic stacking mechanism. `\setdisulfidefeatures` renews this list, replacing any previous contents. `\adddisulfidefeatures` adds the features in its ⟨*key list*⟩ to an existing list, while `\removedisulfidefeatures` removes selected features. By default, there are three disulfide-like features: `disulfide`, `DISULFID` and `range`. Note that `\setfeaturealias` and its relatives do not influence the list.

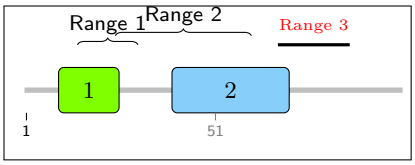
Feature `range` (*no alias*)
Indicates a range of residues. `range` features are disulfide-like in order to prevent them from overlapping.

```
/pgfmolbio/domains/range font =⟨font commands⟩
```

Default: `\sffamily\scriptsize`

Changes the font for the range label. The last command may take a single argument (Example 3.18).

Example 3.18




```
\begin{pmbdomains}[show name=false]{100}
  \addfeature[description=1]{domain}{10}{25}
  \addfeature[description=2]{domain}{40}{70}
  \addfeature[description=Range 1]{range}{15}{30}
  \addfeature[description=Range 2]{range}{25}{60}
  \addfeature[description=Range 3,%
    style={very thick, draw=black},%
    range font=\tiny\textcolor{red}]{range}{68}{86}
\end{pmbdomains}
```

3.7 Ruler

Feature `other/ruler` (*no alias*)
This feature is automatically added to the feature list at the end of each `pmbdomains`

environment. It draws a ruler below the main chain, which indicates the residue numbers (Example 3.19). The following auxiliary commands are available for the feature style Ti*k*Z code: `\xMid`, `\yMid`, `\residueNumber` and `current style`.

`/pgfmolbio/domains/show ruler =`⟨*boolean*⟩

Default: `true`

Determines whether the rule is drawn.

`/pgfmolbio/domains/ruler range =`⟨*ruler range list*⟩

Default: `auto-auto`

The complete syntax for `ruler range` is

⟨*ruler range list*⟩ := `{`⟨*ruler range*⟩`[,`⟨*ruler range*⟩`,...]}`
⟨*ruler range*⟩ := ⟨*lower*⟩`-`⟨*upper*⟩`[ step` ⟨*interval*⟩`]`
⟨*lower*⟩ := `auto | `⟨*number*⟩`[`⟨*letter*⟩`] | (`⟨*number*⟩`)`
⟨*upper*⟩ := `auto | `⟨*number*⟩`[`⟨*letter*⟩`] | (`⟨*number*⟩`)`
⟨*interval*⟩ := ⟨*number*⟩

Each ⟨*ruler range*⟩ tells the package to mark every ⟨*interval*⟩th residue from ⟨*lower*⟩ to ⟨*upper*⟩ by an `other/ruler` feature; the `step` part is optional. Possible values for ⟨*lower*⟩ and ⟨*upper*⟩ are:

- `auto`, which indicates the leftmost or rightmost residue shown, respectively;
- a plain number (with an optional letter), which denotes a residue in the *relative* numbering scheme set by `residue numbering`;
- a parenthesized number, which denotes a residue in the *absolute* numbering scheme.

`/pgfmolbio/domains/default ruler step size =`⟨*number*⟩

Default: `50`

Step size for a ⟨*ruler range*⟩ that lacks the optional `step` part.

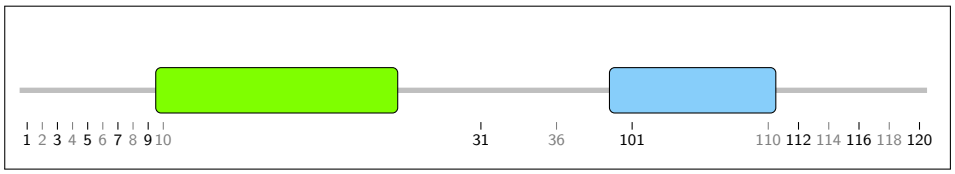
`/pgfmolbio/domains/ruler distance =`⟨*factor*⟩

Default: `-.5`

Separation (multiples of the y-unit) between ruler and main chain (Example 3.19).

3.8 Sequences

Example 3.19




```
\begin{pmbdomains}[x unit=2mm,
    show name=false,
    residue numbering={1-40,101-120},
    ruler range={auto-10 step 1, 31-(41), 110-120 step 2},
    default ruler step size=5,
    ruler distance=-.7]{60}
  \addfeature{domain}{10}{25}
  \addfeature{domain}{40}{(50)}
\end{pmbdomains}
```

/pgfmolbio/domains/**sequence** =⟨*sequence*⟩

Default: empty

Sets the amino acid ⟨*sequence*⟩ of a protein (single-letter abbreviations).

Feature `other/sequence` (*no alias*)
Displays a sequence which is vertically centered at the main chain. Since a residue is only 0.5 mm wide by default, you should increase the `x unit` when showing `sequence` features (Example 3.20).

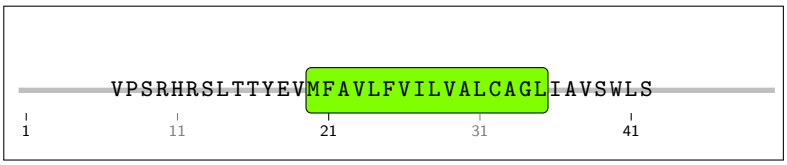
`\setfeatureprintfunction`{⟨*key list*⟩}{⟨*Lua function*⟩}
`\removefeatureprintfunction`{⟨*key list*⟩}
`\pmbdomdrawfeature`{⟨*type*⟩}

Some features require sophisticated coordinate calculations. Hence, you might ocasionally want to call a Lua function as "preprocessor" before executing the ⟨*TikZ code*⟩ of `\setfeatureshape`. For this purpose, `\setfeatureprintfunction` registers such a ⟨*Lua function*⟩ and `\removefeatureprintfunction` deletes the preprocessing function(s) for all features in the ⟨*key list*⟩.

A suitable Lua function

- receives up to six arguments in the following order (see also section 5.6.1):
  1. A table describing the feature (see section 5.6.3 for its fields);
  2. the left x-coordinate of the feature (an integer);
  3. its right x-coordinate (an integer);

Example 3.20




```
1 \begin{pmbdomains}[%
2     sequence=MGSKRSVPSRHRSLTTYEVMFAVLFVILV%
3       ALCAGLIAVSWLSIQGSVKDAAFGKSHEARGTL,
4     residues per line=50,
5     x unit=2mm, show name=false,
6     ruler range=auto-auto step 10]{50}
7   \setfeaturestyle{other/sequence}{font=\ttfamily\footnotesize}
8   \addfeature{domain}{20}{35}
9   \addfeature{other/sequence}{7}{42}
10 \end{pmbdomains}
```

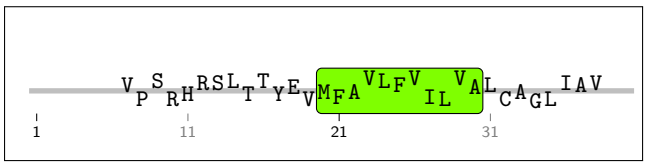
4. the y-coordinate of the current line (an integer);
5. the dimension stored in `x unit`, converted to scaled points (an integer);
6. the dimension stored in `y unit`, converted to scaled points (an integer);

- performs all necessary calculations and defines all TEX macros required by `\setfeatureshape`;
- may execute `\pmbdomdrawfeature` with the appropriate feature ⟨*type*⟩ to draw the feature.

Example 3.21 devises a new print function, `printFunnySequence` (lines 2–17). It is similar to the default print function for `other/sequence` features, but adds random values to the y-coordinate of the individual letters.

`printFunnySequence` is a function with six arguments (line 2). We add the width of half a residue to the left x-coordinate, `xLeft` (line 3), since each letter should be horizontally centered. We iterate over each letter in the `sequence` field of the `feature` table (lines 4–16). In each loop, calculated coordinates are stored in the TEX macros `\xMid` (lines 5–7) and `\yMid` (lines 8–10). The construction `\string\\...` is expanded to `\\...` when `tex.sprint` passes its argument back to TEX. `pgfmolbio.dimToString` converts a number representing a dimension in scaled points to a string (e. g., 65536 to "1pt", see section 5.2). The letter of the current residue is stored in `\currentResidue` (lines 11–13). Finally, each letter is drawn by calling `\pmbdomdrawfeature{other/sequence}` (line 14), and the x-coordinate increases by one (line 15). Line 25 registers `printFunnySequence` for `other/sequence` features.

Example 3.21




```
\directlua{
  function printFunnySequence (feature, xLeft, xRight, yMid, xUnit, yUnit)
    xLeft = xLeft + 0.5
    for currResidue in feature.sequence:gmatch(".") do
      tex.sprint("\string\\def\string\\xMid{" ..
        pgfmolbio.dimToString(xLeft * xUnit) ..
        "}")
      tex.sprint("\string\\def\string\\yMid{" ..
        pgfmolbio.dimToString((yMid + math.random(-5, 5) / 20) * yUnit) ..
        "}")
      tex.sprint("\string\\def\string\\currentResidue{" ..
        currResidue ..
        "}")
      tex.sprint("\string\\pmbdomdrawfeature{other/sequence}")
      xLeft = xLeft + 1
    end
  end
}

\begin{pmbdomains}[%
    sequence=MGSKRSVPSRHRSLTTYEVMFAVLFVILVALCAGLIAVSWLSIQGSVKDAAF,
    x unit=2mm, show name=false,
    ruler range=auto-auto step 10]{40}
  \setfeaturestyle{other/sequence}{font=\ttfamily\footnotesize}
  \setfeatureprintfunction{other/sequence}{printFunnySequence}
  \addfeature{domain}{20}{30}
  \addfeature{other/sequence}{7}{38}
\end{pmbdomains}
```

Feature `other/magnified sequence above` (*no alias*)
Displays its sequence as a single string above the main chain, with dashed lines indicating the sequence start and stop on the backbone. This feature allows you to show sequences without the need to increase the `x unit`.

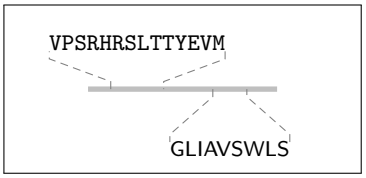
Feature `other/magnified sequence below` (*no alias*)
Displays the sequence *below* the backbone.

`/pgfmolbio/domains/magnified sequence font =`⟨*font commands*⟩

Default: `\ttfamily\footnotesize`

The font used for a magnified sequence (Example 3.22).

Example 3.22




```
\begin{pmbdomains}[%
    sequence=MGSKRSVPSRHRSLTTYEVMFAVLFVIL%
      VALCAGLIAVSWLSIQGSVKDAAFGKSHEARGTL,
    enlarge left=-1cm, enlarge right=1cm, enlarge bottom=-1cm,
    show name=false, show ruler=false]{50}
  \addfeature{other/magnified sequence above}{7}{20}
  \addfeature[magnified sequence font=\scriptsize\sffamily]%
    {other/magnified sequence below}{34}{42}
\end{pmbdomains}
```

3.9 Secondary Structure

`/pgfmolbio/domains/show secondary structure =`⟨*boolean*⟩

Default: `false`

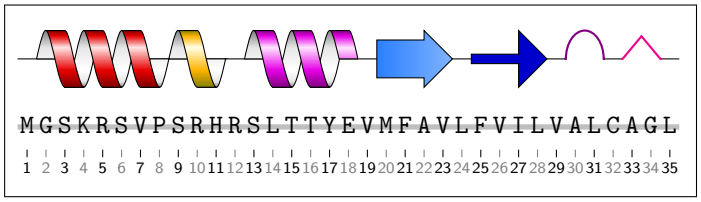
Determines whether the secondary structure is shown.

`/pgfmolbio/domains/secondary structure distance =`⟨*factor*⟩

Default: `1`

Secondary structures appear along a thin line ⟨*factor*⟩ times the value of `y unit` above the main chain. In accordance with the categories established by the Dictionary of Protein Secondary Structure[5], `pgfmolbio` provides seven features for displaying secondary structure types (Example 3.23):

Example 3.23




```
\begin{pmbdomains}[%
    show name=false,
    sequence=MGSKRSVPSRHRSLTTYEVMFAVLFVILVALCAGL,
    x unit=2.5mm,
    enlarge top=1.5cm,
    ruler range=auto-auto step 1,
    show secondary structure=true,
    secondary structure distance=1.5
  ]{35}
  \setfeaturestyle{other/sequence}{{font=\ttfamily\small}}
  \addfeature{alpha helix}{2}{8}
  \addfeature{pi helix}{9}{11}
  \addfeature{310 helix}{13}{18}
  \addfeature{beta strand}{20}{23}
  \addfeature{beta bridge}{25}{28}
  \addfeature{beta turn}{30}{31}
  \addfeature{bend}{33}{34}
  \addfeature{other/sequence}{1}{35}
\end{pmbdomains}
```

Feature `alpha helix` (*alias* `HELIX`)
Shows an α-helix.

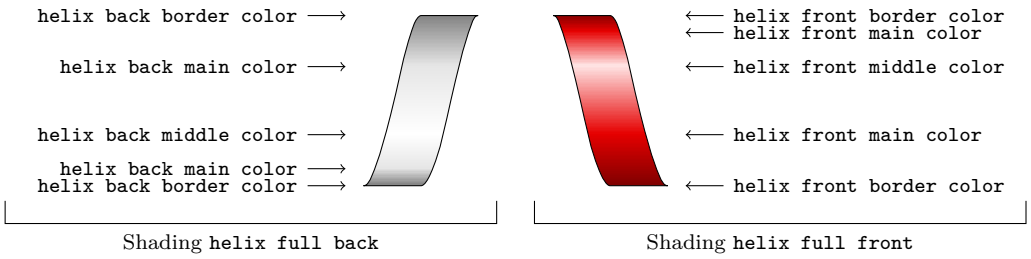
Feature `pi helix` (*no alias*)
Shows a π-helix.

Feature `310 helix` (*no alias*)
Shows a 3_{10}-helix.

[5] Kabsch, W. and Sander, C. (1983). Dictionary of protein secondary structure: pattern recognition of hydrogen-bonded and geometrical features. *Biopolymers* **22**(12), 2577–2637.

Figure 3.1: Shading colors of helix features.

Name	*xcolor definition*		
	α-helix	π-helix	3_{10}-helix
`helix back border color`		`white!50!black`	
`helix back main color`		`white!90!black`	
`helix back middle color`		`white`	
`helix front border color`	`red!50!black`	`yellow!50!black`	`magenta!50!black`
`helix front main color`	`red!90!black`	`yellow!70!red`	`magenta!90!black`
`helix front middle color`	`red!10!white`	`yellow!10!white`	`magenta!10!white`



Feature `beta strand` (*alias* `STRAND`)
Shows a β-strand.

Feature `beta turn` (*alias* `TURN`)
Shows a β-turn.

Feature `beta bridge` (*no alias*)
Shows a β-bridge.

Feature `bend` (*no alias*)
Shows a bend.

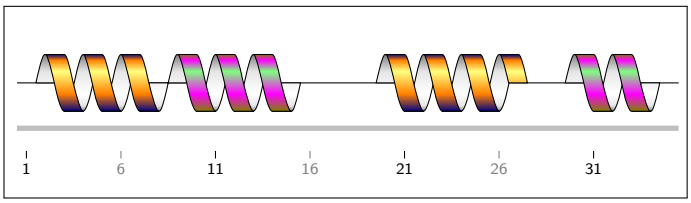
While changing the appearance of nonhelical secondary structure elements is simple, the complex helical features employ the print function `printHelixFeature` (section 5.6.1). However, their appearance can be customized on several levels:

1. The elements of a helical feature are drawn by five "subfeatures", which are called by `printHelixFeature` (Table 3.1a).
2. For each subfeature, there is a corresponding shading (Table 3.1b; see section 5.5.3 and section 83 of the Ti*k*Z manual for their definitions).
3. These shadings use six colors in total, three for front and three for back shadings (Figure 3.1). For each color, there is a key of the same name, so you can change helix colors in feature style lists (Example 3.24).

Table 3.1: Customizing helices in the domains module.

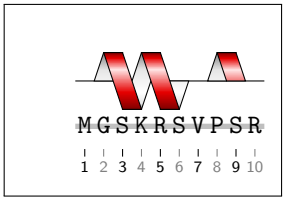
(a) Subfeatures	*(b) Corresponding shadings*	*(c) Coordinates*	
`helix/half upper back`	`helix half upper back`	`\xLeft`	`\yMid`
`helix/half lower back`	`helix half lower back`	`\xRight`	`\yMid`
`helix/full back`	`helix full back`	`\xMid`	`\yLower`
`helix/half upper front`	`helix half upper front`	`\xRight`	`\yMid`
`helix/full front`	`helix full front`	`\xMid`	`\yLower`

Example 3.24




```
\begin{pmbdomains}[%
    show name=false,
    x unit=2.5mm,
    enlarge top=1.5cm,
    ruler range=auto-auto step 5,
    show secondary structure
  ]{35}
  \setfeaturestyle{alpha helix}{%
    *1{helix front border color=blue!50!black,%
    helix front main color=orange,%
    helix front middle color=yellow!50},%
    *1{helix front border color=olive,%
    helix front main color=magenta,%
    helix front middle color=green!50}%
  }
  \addfeature{alpha helix}{2}{8}
  \addfeature{alpha helix}{9}{15}
  \addfeature{alpha helix}{20}{27}
  \addfeature{alpha helix}{30}{34}
\end{pmbdomains}
```

Example 3.25



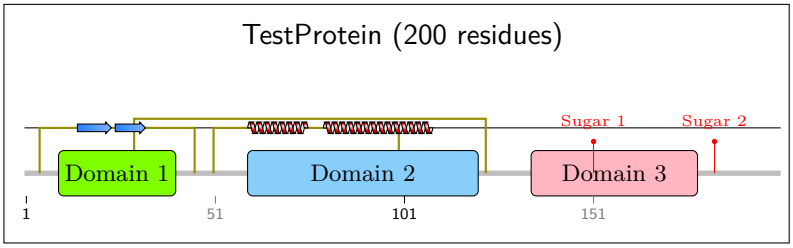

```latex
\pgfmathsetmacro\yShift{%
  \pmbdomvalueof{secondary structure distance}
    * \pmbdomvalueof{y unit}%
}

\setfeatureshape{helix/half upper back}{%
  \draw [shading=helix half upper back]
    (\xLeft, \yMid + \yShift pt) --
    (\xLeft + .5 * \pmbdomvalueof{x unit},
      \yMid + 1.5 * \pmbdomvalueof{x unit} + \yShift pt) --
    (\xLeft + 1.5 * \pmbdomvalueof{x unit},
      \yMid + 1.5 * \pmbdomvalueof{x unit} + \yShift pt) --
    (\xLeft + \pmbdomvalueof{x unit}, \yMid + \yShift pt) --
    cycle;
}

\setfeatureshape{helix/half lower back}{%
  \draw [shading=helix half lower back]
    (\xRight, \yMid + \yShift pt) --
    (\xRight - .5 * \pmbdomvalueof{x unit},
      \yMid - 1.5 * \pmbdomvalueof{x unit} + \yShift pt) --
    (\xRight - 1.5 * \pmbdomvalueof{x unit},
      \yMid - 1.5 * \pmbdomvalueof{x unit} + \yShift pt) --
    (\xRight - \pmbdomvalueof{x unit}, \yMid + \yShift pt) --
    cycle;
}

\setfeatureshape{helix/full back}{%
  \draw [shading=helix full back]
    (\xMid, \yLower + \yShift pt) --
    (\xMid - \pmbdomvalueof{x unit}, \yLower + \yShift pt) --
    (\xMid, \yLower + 3 * \pmbdomvalueof{x unit} + \yShift pt) --
    (\xMid + \pmbdomvalueof{x unit},
      \yLower + 3 * \pmbdomvalueof{x unit} + \yShift pt) --
    cycle;
}

\setfeatureshape{helix/half upper front}{%
  \draw [shading=helix half upper front]
    (\xRight, \yMid + \yShift pt) --
    (\xRight - .5 * \pmbdomvalueof{x unit},
      \yMid + 1.5 * \pmbdomvalueof{x unit} + \yShift pt) --
    (\xRight - 1.5 * \pmbdomvalueof{x unit},
      \yMid + 1.5 * \pmbdomvalueof{x unit} + \yShift pt) --
    (\xRight - \pmbdomvalueof{x unit}, \yMid + \yShift pt) --
    cycle;
}

\setfeatureshape{helix/full front}{%
  \draw [shading=helix full front]
    (\xMid, \yLower + \yShift pt) --
    (\xMid + \pmbdomvalueof{x unit}, \yLower + \yShift pt) --
    (\xMid, \yLower + 3 * \pmbdomvalueof{x unit} + \yShift pt) --
    (\xMid - \pmbdomvalueof{x unit},
      \yLower + 3 * \pmbdomvalueof{x unit} + \yShift pt) --
    cycle;
}

\begin{pmbdomains}[%
    show name=false, sequence=MGSKRSVPSR,
    x unit=2.5mm, enlarge top=1.5cm,
    ruler range=auto-auto step 1,
    show secondary structure
  ]{10}
  \setfeaturestyle{other/sequence}{{font=\ttfamily\small}}
  \addfeature{alpha helix}{2}{6}
  \addfeature{alpha helix}{8}{9}
  \addfeature{other/sequence}{1}{10}
\end{pmbdomains}
```

3.10 File Input

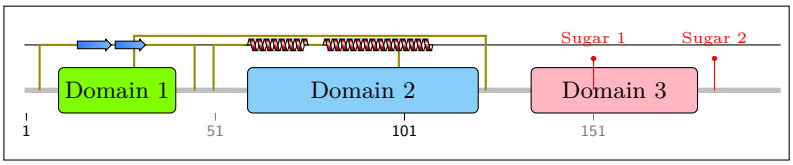
```
\inputuniprot{⟨Uniprot file⟩}
\inputgff{⟨gff file⟩}
```

Include the features defined in an ⟨*Uniprot file*⟩ or ⟨*gff file*⟩, respectively (Example 3.26). These macros are only defined in pmbdomains.

Example 3.26




```
\begin{pmbdomains}[show secondary structure]{}
  \setfeaturestyle{disulfide}{{draw=olive,thick}}
  \inputuniprot{SampleUniprot.txt}
\end{pmbdomains}
```




```
\begin{pmbdomains}[show name=false,show secondary structure]{200}
  \setfeaturestyle{disulfide}{{draw=olive,thick}}
  \inputgff{SampleGff.gff}
\end{pmbdomains}
```

/pgfmolbio/domains/sequence length =⟨*number*⟩

Default: (empty)

Note that in Example 3.26, we had to set a sequence length for the pmbdomains environment that contains the \inputgff macro. gff files lack a sequence length field. By contrast, pgfmolbio reads the sequence length from an Uniprot file, and thus the mandatory argument of pmbdomains may remain empty. In general, the sequence length is stored in the key of the same name.

4 The convert module

4.1 Overview

The convert module supports users who wish to include pgfmolbio graphs, but who do not want to typeset their documents with a TEX engine that implements Lua. To this end, the convert workflow comprises two steps: (1) Running LuaLATEX on an input file that contains at least one \pmbchromatogram or similar macros/environments. This will generate one tex file per graph macro/environment that contains only TikZ commands. (2) Including this file in another TEX document (via \input) which is then processed by any TEX engine that supports TikZ.

4.2 Converting Chromatograms

In order to create the external TikZ file, run an input file like the one below through LuaLATEX:

```
\documentclass{article}
\usepackage[chromatogram,convert]{pgfmolbio}

\begin{document}
  \pmbchromatogram[sample range=base 50-base 60]{SampleScf.scf}
  \pmbchromatogram[/pgfmolbio/convert/output file name=mytikzfile]%
    {SampleScf.scf}
  \pmbchromatogram[sample range=base 60-base 70]{SampleScf.scf}
\end{document}
```

The convert module disables pdf output and introduces the following keys:

/pgfmolbio/convert/output file name =⟨*text*⟩

Default: (auto)

/pgfmolbio/convert/output file extension =⟨*text*⟩

Default: tex

With the default value for output file name ("(auto)"), pgfmolbio creates files that are named pmbconverted and numbered consecutively (pmbconverted0.tex, pmbconverted1.tex etc.). Both keys can be changed locally (e. g., in the optional argument of \pmbchromatogram), but this turns off automatic numbering.

The code above produces the files `pmbconverted0.tex`, `mytikzfile.tex` and `pmbconverted2.tex`. Below is an annotated excerpt from `pmbconverted0.tex`:

```
\begin{tikzpicture}
  [canvas section]
  \draw [/pgfmolbio/chromatogram/canvas style] (0mm, -0mm) rectangle (25mm, 20mm);
  [traces section]
  \draw [/pgfmolbio/chromatogram/trace A style] (0mm, 6.37mm) -- (0.2mm, 6.66mm) -- [many
     coordinates] -- (25mm, 0mm);
  \draw [/pgfmolbio/chromatogram/trace C style] (0mm, 0.06mm) -- (0.2mm, 0.05mm) -- [...] --
     (25mm, 6.27mm);
  \draw [/pgfmolbio/chromatogram/trace G style] (0mm, 0.01mm) -- (0.2mm, 0.01mm) -- [...] --
     (25mm, 0.05mm);
  \draw [/pgfmolbio/chromatogram/trace T style] (0mm, 0mm) -- (0.2mm, 0mm) -- [...] -- (25mm,
     0.06mm);
  [ticks/base labels/probabilities section]
  \draw [/pgfmolbio/chromatogram/tick A style] (0mm, -0mm) -- (0mm, -1mm) node [/pgfmolbio/
     chromatogram/base label A style] {\pgfkeysvalueof{/pgfmolbio/chromatogram/base label A
     text}} node [/pgfmolbio/chromatogram/base number style] {\strut 50};
  \draw [ultra thick, pmbTraceGreen] (0mm, -8mm) -- (0.9mm, -8mm);
  \draw [/pgfmolbio/chromatogram/tick T style] (1.8mm, -0mm) -- (1.8mm, -1mm) node [/
     pgfmolbio/chromatogram/base label T style] {\pgfkeysvalueof{/pgfmolbio/chromatogram/base
     label T text}};
  \draw [ultra thick, pmbTraceGreen] (0.9mm, -8mm) -- (3mm, -8mm);
  \draw [/pgfmolbio/chromatogram/tick A style] (4.2mm, -0mm) -- (4.2mm, -1mm) node [/
     pgfmolbio/chromatogram/base label A style] {\pgfkeysvalueof{/pgfmolbio/chromatogram/base
     label A text}};
  \draw [ultra thick, pmbTraceGreen] (3mm, -8mm) -- (5.4mm, -8mm);
  [...]
  [more ticks, base labels and probability rules]
\end{tikzpicture}
```

You can change the format of the coordinates by the following keys:

`/pgfmolbio/coordinate unit =`⟨*unit*⟩

Default: `mm`

`/pgfmolbio/coordinate format string =`⟨*format string*⟩

Default: `%s%s`

`pgfmolbio` internally calculates dimensions in scaled points, but usually converts them before returning them to TEX. To this end, it selects the ⟨*unit*⟩ stored in `coordinate unit` (any of the standard TEX units of measurement: `bp`, `cc`, `cm`, `dd`, `in`, `mm`, `pc`, `pt` or `sp`). In addition, the package formats the dimension according to the ⟨*format string*⟩ given by `coordinate format string`. This string basically follows the syntax of C's `printf` function, as described in the Lua reference manual. (Note: Use `\letterpercent` instead of `%`, since TEX treats anything following a percent character as comment.)

Depending on the values of `coordinate unit` and `coordinate format string`, dimensions will be printed in different ways (Table 4.1).

The output files can be included in a file which is processed by pdfLATEX:

Table 4.1: Effects of `coordinate unit` and `coordinate format string` when converting an internal `pgfmolbio` dimension of 200000 [sp].

Values		*Output*	*Notes*
`sp`	`%s%s`	`200000sp`	simple conversion
`mm`	`%s%s`	`1.0725702011554mm`	default settings, may lead to a large number of decimal places
`mm`	`%.3f%s`	`1.073mm`	round to three decimal places
`cm`	`%.3f`	`0.107`	don't print any unit, i. e. use Ti*k*Z's `xyz` coordinate system

```
\documentclass{article}
\usepackage[chromatogram]{pgfmolbio}

\begin{document}
  \input{pmbconverted.tex}
\end{document}
```

Several keys of the chromatogram module must contain their final values before conversion, while others can be changed afterwards, i. e., before the generated file is loaded with `\input` (Table 4.2).

Table 4.2: Keys of the chromatogram module that require final values prior to conversion.

Required		*Not required*
`base labels drawn`	`sample range`	`base label style`
`base number range`	`samples per line`	`base label X style`
`baseline skip`	`show base numbers`	`base label X text`
`bases drawn`	`tick length`	`base number style`
`canvas height`	`ticks drawn`	`canvas style`
`probabilities drawn`	`traces drawn`	`tick style`
`probability distance`	`x unit`	`tick X style`
`probability style function`	`y unit`	`trace style`
		`trace X style`

4.3 Converting Domain Diagrams

`/pgfmolbio/convert/output code =pgfmolbio | tikz`

Default: `tikz`

In principle, domain diagrams are converted like sequencing chromatograms (section 4.2). However, `output code` lets you choose the kind of code convert writes

to the output file: pgfmolbio generates a pmbdomains environment containing \addfeature commands, tikz produces TikZ code.

"Converting" one pmbdomains environment in the input file to another one in the output file might seem pointless. Nonetheless, this conversion mechanism can be highly useful for extracting features from a Uniprot or gff file. For example, consider the following input file:

```
\documentclass{article}
\usepackage[domains,convert]{pgfmolbio}

\begin{document}
  \pgfmolbioset[convert]{output code=pgfmolbio}
  \begin{pmbdomains}{}
    \inputuniprot{SampleUniprot.txt}
  \end{pmbdomains}
\end{document}
```

The corresponding output is

```
\begin{pmbdomains}
    [name={TestProtein},
    sequence=MGSKRSVPSRHRSL[...]PLATPGNVSIECP]{200}
  \addfeature[description={Disulfide 1}]{DISULFID}{5}{45}
  \addfeature[description={Disulfide 2}]{DISULFID}{30}{122}
  \addfeature[description={Disulfide 3}]{DISULFID}{51}{99}
  \addfeature[description={Domain 1}]{DOMAIN}{10}{40}
  \addfeature[description={Domain 2}]{DOMAIN}{60}{120}
  \addfeature[description={Domain 3}]{DOMAIN}{135}{178}
  \addfeature[description={Strand 1}]{STRAND}{15}{23}
  \addfeature[description={Strand 2}]{STRAND}{25}{32}
  \addfeature[description={Helix 1}]{HELIX}{60}{75}
  \addfeature[description={Helix 2}]{HELIX}{80}{108}
  \addfeature[description={Sugar 1}]{CARBOHYD}{151}{151}
  \addfeature[description={Sugar 2}]{CARBOHYD}{183}{183}
\end{pmbdomains}
```

Obviously, this method is particularly suitable for Uniprot files containing many features.

/pgfmolbio/convert/include description =⟨*boolean*⟩

Default: true

Decides whether the feature description obtained from the input should appear in the output. Since the description field in FT entries of Uniprot files can be quite long, you may not wish to show it in the output. For example, the output of the example above with include description=false looks like

```
\begin{pmbdomains}
    [name={TestProtein},
    sequence=MGSKRSVPSRHRSL[...]PLATPGNVSIECP]{200}
```

```
  \addfeature{DISULFID}{5}{45}
  \addfeature{DISULFID}{30}{122}
  \addfeature{DISULFID}{51}{99}
  [...]
\end{pmbdomains}
```

With `output code=tikz`, we obtain the following (annotated) output file:

```
[set relevant keys]
\pgfmolbioset[domains]{name={TestProtein},sequence={MGSKRS[...]VSIECP},sequence length=200}
[the actual TikZ picture]
\begin{tikzpicture}
  [each feature appears within its own scope]
  \begin{scope}\begin{pgfinterruptboundingbox}
    \def\xLeft{0mm}
    \def\xMid{50mm}
    \def\xRight{100mm}
    \def\yMid{-0mm}
    \def\featureSequence{MGSKRS[...]VSIECP}
    \clip (-50mm, \yMid + 100mm) rectangle (150mm, \yMid - 100mm);
    \pgfmolbioset[domains]{style={{draw, line width=2pt, black!25}},@layer=1}
    \pmbdomdrawfeature{other/main chain}
  \end{pgfinterruptboundingbox}\end{scope}
  [more features]
  [...]
  [helix features require additional drawing commands]
  \begin{scope}\begin{pgfinterruptboundingbox}
    \def\xLeft{29.5mm}
    \def\xMid{33.5mm}
    \def\xRight{37.5mm}
    \def\yMid{-0mm}
    \def\featureSequence{GTLKIISGATYNPHLQ}
    \clip (-50mm, \yMid + 100mm) rectangle (87.5mm, \yMid - 100mm);
    \pgfmolbioset[domains]{style={{helix front border color=red!50!black,helix front main
     color=red!90!black,helix front middle color=red!10!white}},description={Helix 1}}
    \pgfmolbioset[domains]{current style}
    \def\xLeft{29.5mm}
    \def\yMid{-0mm}
    \pmbdomdrawfeature{helix/half upper back}
    \def\xMid{30.75mm}
    \def\yLower{-0.75mm}
    \pmbdomdrawfeature{helix/full back}
    [more helix parts]
  \end{pgfinterruptboundingbox}\end{scope}
  [...]
  [ruler section]
  \begin{scope}
    \pgfmolbioset[domains]{current style/.style={black}}
      \def\xMid{0.25mm}
      \let\xLeft\xMid\let\xRight\xMid
      \def\yMid{-0mm}
      \def\residueNumber{1}
      \pmbdomdrawfeature{other/ruler}
    \pgfmolbioset[domains]{current style/.style={black!50}}
      \def\xMid{25.25mm}
      \let\xLeft\xMid\let\xRight\xMid
      \def\yMid{-0mm}
      \def\residueNumber{51}
      \pmbdomdrawfeature{other/ruler}
    [more ruler numbers]
```

```
    [...]
  \end{scope}
  [name section]
  \begin{scope}
    \pgfmolbioset[domains]{current style/.style={font=\sffamily }}
    \def\xLeft{0mm}
    \def\xMid{50mm}
    \def\xRight{100mm}
    \def\yMid{0mm}
    \pmbdomdrawfeature{other/name}
  \end{scope}
  [adjust picture size]
  \pmbprotocolsizes{\pmbdomvalueof{enlarge left}}{\pmbdomvalueof{enlarge top}}
  \pmbprotocolsizes{100mm + \pmbdomvalueof{enlarge right}}{-0mm + \pmbdomvalueof{enlarge
     bottom}}
\end{tikzpicture}
```

Several keys of the domains module must contain their final values before conversion, and some macros can't be used afterwards (Table 4.3).

Table 4.3: Keys and macros of the domain module that require final values prior to conversion or can't be used afterwards, respectively.

Required		*Not required*
baseline skip	ruler distance	domain font
default ruler step size	ruler range	enlarge bottom
description	secondary structure distance	enlarge left
disulfide base distance	sequence	enlarge right
disulfide level distance	sequence length	enlarge top
level	show ruler	magnified sequence font
name	style	range font
residue numbering	x unit	show secondary structure
residue range	y unit	
residues per line		
\adddisulfidefeatures	\setfeatureprintfunction	\setfeaturealias
\removedisulfidefeatures	\setfeaturestyle	\setfeatureshape
\removefeatureprintfunction	\setfeaturestylealias	\setfeatureshapealias
\setdisulfidefeatures		

5 Implementation

5.1 pgfmolbio.sty

The options for the main style file determine which module(s) should be loaded.

```
\newif\ifpmb@loadmodule@chromatogram
\newif\ifpmb@loadmodule@domains
\newif\ifpmb@loadmodule@convert

\DeclareOption{chromatogram}{%
  \pmb@loadmodule@chromatogramtrue%
}
\DeclareOption{domains}{%
  \pmb@loadmodule@domainstrue%
}
\DeclareOption{convert}{%
  \pmb@loadmodule@converttrue%
}

\ProcessOptions

```

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

```
\RequirePackage{ifluatex}
\ifluatex
  \RequirePackage{luatexbase-modutils}
    \RequireLuaModule{lualibs}
    \RequireLuaModule{pgfmolbio}
\fi
\RequirePackage[svgnames,dvipsnames]{xcolor}
\RequirePackage{tikz}
  \usetikzlibrary{positioning,svg.path}

```

`\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%
}

```

We introduce two package-wide keys.

```
\pgfkeyssetvalue{/pgfmolbio/coordinate unit}{mm}
\pgfkeyssetvalue{/pgfmolbio/coordinate format string}{\letterpercent s
    \letterpercent s}

```

Furthermore, we define two scratch token registers. Strictly speaking, the two conditionals belong to the convert module, but all modules need to know them.

```
\newtoks\@pmb@toksa
\newtoks\@pmb@toksb
\newif\ifpmb@con@includedescription
\newif\ifpmb@con@outputtikzcode

```

\pmbprotocolsizes

#1: x-coordinate.
#2: y-coordinate.

An improved version of \pgf@protocolsizes that accepts coordinate calculations.

```
\def\pmbprotocolsizes#1#2{%
  \pgfpoint{#1}{#2}%
  \pgf@protocolsizes{\pgf@x}{\pgf@y}%
}

```

Finally, we load the modules requested by the user.

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

5.2 pgfmolbio.lua

Identification of the Lua module.

```lua
if luatexbase then
  luatexbase.provides_module({
    name          = "pgfmolbio",
    version       = "0.21a",
    date          = "2014/06/17",
    description   = "Molecular biology graphs wit LuaLaTeX",
    author        = "Wolfgang Esser-Skala",
    copyright     = "Wolfgang Esser-Skala",
    license       = "LPPL",
  })
end

```

`setCoordinateFormat` sets the output format of `dimToString` (see below). Both its parameters `unit` and `fmtString` are strings, which correspond to the values of `coordinate unit` and `coordinate format string`.

```lua
local coordUnit, coordFmtStr

function setCoordinateFormat(unit, fmtString)
  coordUnit = unit
  coordFmtStr = fmtString
end

```

`stringToDim` converts a string describing a TEX dimension to a number corresponding to scaled points. `dimToString` converts a dimension in scaled points to a string, formatting it according to the values of the local variables `coordUnit` and `coordFmtString`.

```lua
function stringToDim(x)
  if type(x) == "string" then
    return dimen(x)[1]
  end
end

function dimToString(x)
  return number.todimen(x, coordUnit, coordFmtStr)
end

```

`getRange` extracts a variable number of strings from `rangeInput` by applying the regular expressions in the table `matchStrings`, which derives from the varargs. `rangeInput` contains the values of any of the `... range` keys.

```lua
function getRange(rangeInput, ...)
  if type(rangeInput) ~= "string" then return end
  local result = {}
  local matchStrings = table.pack(...)
  for i = 1, matchStrings.n do
    if type(matchStrings[i]) == "string" then
      table.insert(result, rangeInput:match(matchStrings[i]))
    end
  end
  return unpack(result)
end

```

`packageWarning` and `packageError` throw TEX warnings and errors, respectively. `packageError` also sets the global variable `errorCatched` to `true`. Some Lua functions check the value of this variable and terminate if an error has occurred.

```lua
function packageWarning(message)
  tex.sprint("\\PackageWarning{pgfmolbio}{" .. message .. "}")
end

function packageError(message)
  tex.error("Package pgfmolbio Error: " .. message)
  errorCatched = true
end

errorCatched = false

```

We extend the `string` table by the function `string.trim`, which removes leading and trailing spaces.

```lua
if not string.trim then
  string.trim = function(self)
    return self:match("^%s*(.-)%s*$")
  end
end

```

`outputFileId` is a counter to enumerate several output files by the convert module.

```lua
outputFileId = 0
```

5.3 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.

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

```

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
#2: default ⟨*value*⟩

Most of the keys simply store their value. \@pmb@chr@keydef simplifies the declaration of such keys by calling **\pgfkeyssetvalue** with the appropriate path, ⟨*key*⟩ and ⟨*value*⟩.

```
\def\@pmb@chr@keydef#1#2{%
  \pgfkeyssetvalue{/pgfmolbio/chromatogram/#1}{#2}%
}
```

\@pmb@chr@stylekeydef

#1: ⟨*key*⟩ name
#2: default ⟨*value*⟩

This macro initializes a style ⟨*key*⟩ with a ⟨*value*⟩.

```
\def\@pmb@chr@stylekeydef#1#2{%
  \pgfkeys{/pgfmolbio/chromatogram/#1/.style={#2}}%
}
```

\@pmb@chr@getkey

#1: ⟨*key*⟩ name

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

```
\def\@pmb@chr@getkey#1{%
  \pgfkeysvalueof{/pgfmolbio/chromatogram/#1}%
}

```

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

```
\@pmb@chr@keydef{sample range}{1-500 step 1}

\@pmb@chr@keydef{x unit}{0.2mm}
\@pmb@chr@keydef{y unit}{0.01mm}
\@pmb@chr@keydef{samples per line}{500}
\@pmb@chr@keydef{baseline skip}{3cm}
\@pmb@chr@stylekeydef{canvas style}{draw=none, fill=none}
\@pmb@chr@keydef{canvas height}{2cm}

\@pmb@chr@stylekeydef{trace A style}{pmbTraceGreen}
\@pmb@chr@stylekeydef{trace C style}{pmbTraceBlue}
\@pmb@chr@stylekeydef{trace G style}{pmbTraceBlack}
\@pmb@chr@stylekeydef{trace T style}{pmbTraceRed}
\pgfmolbioset[chromatogram]{%
  trace style/.code=\pgfkeysalso{
    trace A style/.style={#1},
    trace C style/.style={#1},
    trace G style/.style={#1},
    trace T style/.style={#1}
  }%
}
\@pmb@chr@keydef{traces drawn}{}

\@pmb@chr@stylekeydef{tick A style}{thin, pmbTraceGreen}
\@pmb@chr@stylekeydef{tick C style}{thin, pmbTraceBlue}
\@pmb@chr@stylekeydef{tick G style}{thin, pmbTraceBlack}
\@pmb@chr@stylekeydef{tick T style}{thin, pmbTraceRed}
\pgfmolbioset[chromatogram]{%
  tick style/.code=\pgfkeysalso{
    tick A style/.style={#1},
    tick C style/.style={#1},
    tick G style/.style={#1},
    tick T style/.style={#1}
  }%
}
\@pmb@chr@keydef{tick length}{1mm}
\@pmb@chr@keydef{ticks drawn}{}

\@pmb@chr@keydef{base label A text}{\strut A}
\@pmb@chr@keydef{base label C text}{\strut C}
\@pmb@chr@keydef{base label G text}{\strut G}
```

```
\@pmb@chr@keydef{base label T text}{\strut T}
\@pmb@chr@stylekeydef{base label A style}%
  {below=4pt, font=\ttfamily\footnotesize, pmbTraceGreen}
\@pmb@chr@stylekeydef{base label C style}%
  {below=4pt, font=\ttfamily\footnotesize, pmbTraceBlue}
\@pmb@chr@stylekeydef{base label G style}%
  {below=4pt, font=\ttfamily\footnotesize, pmbTraceBlack}
\@pmb@chr@stylekeydef{base label T style}%
  {below=4pt, font=\ttfamily\footnotesize, pmbTraceRed}
\pgfmolbioset[chromatogram]{%
  base label style/.code=\pgfkeysalso{
    base label A style/.style={#1},
    base label C style/.style={#1},
    base label G style/.style={#1},
    base label T style/.style={#1}
  }%
}
\@pmb@chr@keydef{base labels drawn}{}

\newif\ifpmb@chr@showbasenumbers
\pgfmolbioset[chromatogram]{%
  show base numbers/.is if=pmb@chr@showbasenumbers,
  show base numbers
}
\@pmb@chr@stylekeydef{base number style}%
  {pmbTraceBlack, below=-3pt, font=\sffamily\tiny}
\@pmb@chr@keydef{base number range}{auto-auto step 10}

\@pmb@chr@keydef{probability distance}{0.8cm}
\@pmb@chr@keydef{probabilities drawn}{}
\@pmb@chr@keydef{probability style function}{nil}

\pgfmolbioset[chromatogram]{
  bases drawn/.code=\pgfkeysalso{
    traces drawn=#1,
    ticks drawn=#1,
    base labels drawn=#1,
    probabilities drawn=#1
  },
  bases drawn=ACGT
}

```

If pgfmolbio is used with a TEX engine that does not support Lua, the package ends here.

```
\ifluatex\else\expandafter\endinput\fi

```

\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 generate a new Chromatogram object and invoke several Lua functions: (1) readScfFile reads the given ⟨*scf file*⟩ (section 5.4.3). (2) setParameters passes the values stored by the keys to the Lua script. Note that this function is called twice, since baseNumberRange requires that sampleRange has been already set, and the implementation of setParameters does not ensure this (section 5.4.4). (3) pgfmolbio.setCoordinateFormat sets the coordinate output format (section 5.2).

```
  \directlua{
    pmbChromatogram = pgfmolbio.chromatogram.Chromatogram:new()
    pmbChromatogram:readScfFile("#2")
    pmbChromatogram:setParameters{
      sampleRange = "\@pmb@chr@getkey{sample range}",
      xUnit = "\@pmb@chr@getkey{x unit}",
      yUnit = "\@pmb@chr@getkey{y unit}",
      samplesPerLine = "\@pmb@chr@getkey{samples per line}",
      baselineSkip = "\@pmb@chr@getkey{baseline skip}",
      canvasHeight = "\@pmb@chr@getkey{canvas height}",
      tracesDrawn = "\@pmb@chr@getkey{traces drawn}",
      tickLength = "\@pmb@chr@getkey{tick length}",
      ticksDrawn = "\@pmb@chr@getkey{ticks drawn}",
      baseLabelsDrawn = "\@pmb@chr@getkey{base labels drawn}",
      showBaseNumbers = "\ifpmb@chr@showbasenumbers true\else false\fi",
      probDistance = "\@pmb@chr@getkey{probability distance}",
      probabilitiesDrawn = "\@pmb@chr@getkey{probabilities drawn}",
      probStyle = \@pmb@chr@getkey{probability style function}
    }
    pmbChromatogram:setParameters{
      baseNumberRange = "\@pmb@chr@getkey{base number range}",
    }
    pgfmolbio.setCoordinateFormat(
```

```
      "\pgfkeysvalueof{/pgfmolbio/coordinate unit}",
      "\pgfkeysvalueof{/pgfmolbio/coordinate format string}"
    )
```

If the convert module is loaded, we open the appropriate output file, change `tex.sprint` so that the function writes to this file and then call `printTikzChromatogram`. Without the convert module, `printTikzChromatogram` simply returns the drawing commands for the chromatogram to the TEX input stream (section 5.4.5).

```
    \ifpmb@loadmodule@convert
      local filename =
        "\pgfkeysvalueof{/pgfmolbio/convert/output file name}"
      if filename == "(auto)" then
        filename = "pmbconverted" .. pgfmolbio.outputFileId
      end
      filename = filename ..
        ".\pgfkeysvalueof{/pgfmolbio/convert/output file extension}"
      outputFile, ioError = io.open(filename, "w")
      if ioError then
        tex.error(ioError)
      end
      tex.sprint = function (a) outputFile:write(a) end
      tex.sprint("\string\\begin{tikzpicture}")
      pmbChromatogram:printTikzChromatogram()
      tex.sprint("\string\n\string\\end{tikzpicture}")
      outputFile:close()
      pgfmolbio.outputFileId = pgfmolbio.outputFileId + 1
    \else
      pmbChromatogram:printTikzChromatogram()
    \fi
  }%
```

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%
}
```

5.4 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.

```
if luatexbase then
  luatexbase.provides_module{
```

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

```
    name          = "pgfmolbio.chromatogram",
    version       = "0.21a",
    date          = "2014/06/17",
    description   = "DNA sequencing chromatograms",
    author        = "Wolfgang Esser-Skala",
    copyright     = "Wolfgang Esser-Skala",
    license       = "LPPL",
  }
end

```

5.4.1 Module-Wide Variables and Auxiliary Functions

- `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.

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

```

These local functions point to functions in `pgfmolbio.lua` (section 5.2).

```
local stringToDim = pgfmolbio.stringToDim
local dimToString = pgfmolbio.dimToString
local packageError = pgfmolbio.packageError
local packageWarning = pgfmolbio.packageWarning
local getRange = pgfmolbio.getRange

```

`stdProbStyle` is the default `probability style function`. It returns a string representing an optional argument of `\draw`. Depending on the value of `prob`, the 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 target:upper():find(v) then
      table.insert(result, v)
    end
  end
  return result
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 + file:read(1):byte()
  end
  return result
end

```

5.4.2 The `Chromatogram` Class

The Chromatogram class (table) represents a single `scf` chromatogram. The constructor Chromatogram:new returns a new instance and initializes its variables, which store the values of chromatogram keys. Most variables are self-explanatory, since their name is similar to their corresponding key.

```
Chromatogram = {}

function Chromatogram:new()
  newChromatogram = {
    sampleMin = 1,
```

```
    sampleMax = 500,
    sampleStep = 1,
    peakMin = -1,
    peakMax = -1,
    xUnit = stringToDim("0.2mm"),
    yUnit = stringToDim("0.01mm"),
    samplesPerLine = 500,
    baselineSkip = stringToDim("3cm"),
    canvasHeight = stringToDim("2cm"),
    traceStyle = {
      A = PGFKEYS_PATH .. "trace A style",
      C = PGFKEYS_PATH .. "trace C style",
      G = PGFKEYS_PATH .. "trace G style",
      T = PGFKEYS_PATH .. "trace T style"
    },
    tickStyle = {
      A = PGFKEYS_PATH .. "tick A style",
      C = PGFKEYS_PATH .. "tick C style",
      G = PGFKEYS_PATH .. "tick G style",
      T = PGFKEYS_PATH .. "tick T style"
    },
    tickLength = stringToDim("1mm"),
    baseLabelText = {
      A = "\\pgfkeysvalueof{" .. PGFKEYS_PATH .. "base label A text}",
      C = "\\pgfkeysvalueof{" .. PGFKEYS_PATH .. "base label C text}",
      G = "\\pgfkeysvalueof{" .. PGFKEYS_PATH .. "base label G text}",
      T = "\\pgfkeysvalueof{" .. PGFKEYS_PATH .. "base label T text}"
    },
    baseLabelStyle = {
      A = PGFKEYS_PATH .. "base label A style",
      C = PGFKEYS_PATH .. "base label C style",
      G = PGFKEYS_PATH .. "base label G style",
      T = PGFKEYS_PATH .. "base label T style"
    },
    showBaseNumbers = true,
    baseNumberMin = -1,
    baseNumberMax = -1,
    baseNumberStep = 10,
    probDistance = stringToDim("0.8cm"),
    probStyle = stdProbStyle,
    tracesDrawn = ALL_BASES,
    ticksDrawn = "ACGT",
    baseLabelsDrawn = "ACGT",
    probabilitiesDrawn = "ACGT",
  }
  setmetatable(newChromatogram, self)
  self.__index = self
  return newChromatogram
end
```

```

```

getMinMaxProbability returns the minimum and maximum probability value in the current chromatogram.

```
function Chromatogram:getMinMaxProbability()
  local minProb = 0
  local maxProb = 0
  for _, currPeak in ipairs(self.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

```

getSampleAndPeakIndex returns the sample (sampleId) and peak index (peakId) that correspond to baseIndex. If baseIndex is a number, the function simply returns it as sample index. However, if baseIndex is a string of the form "base ⟨*number*⟩" (as in a valid value for the sample range key), the function returns the offset of the ⟨*number*⟩-th peak. isLowerLimit must be true if the function should return the indices of the lower end of a range.

```
function Chromatogram:getSampleAndPeakIndex(baseIndex, isLowerLimit)
  local sampleId, peakId

  sampleId = tonumber(baseIndex)
  if sampleId then
    for i, v in ipairs(self.peaks) do
      if isLowerLimit then
        if v.offset >= sampleId then
          peakId = i
          break
        end
      else
        if v.offset == sampleId then
          peakId = i
          break
        elseif v.offset > sampleId then
          peakId = i - 1
          break
        end
      end
    end
  else
    peakId = tonumber(baseIndex:match("base%s*(%d+)"))
```

```
    if peakId then
      sampleId = self.peaks[peakId].offset
    end
  end
  return sampleId, peakId
end

```

5.4.3 Read the scf File

`Chromatogram:readScfFile` introduces three further fields to `Chromatogram`:

- `header`: A table of 14 named number fields that save the information in the `scf` header.
- `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.
- `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.

`Chromatogram:readScfFile` checks whether the requested `scf` file "`filename`" corresponds to the most recently opened one (via `lastScfFile`). 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 Chromatogram:readScfFile(filename)
  if filename ~= self.lastScfFile then
    self.lastScfFile = filename
    local scfFile, errorMsg = io.open(filename, "rb")
    if not scfFile then packageError(errorMsg) end

    self.samples = {A = {}, C = {}, G = {}, T = {}}
    self.peaks = {}
```

The function collects the relevant data from the file. *Firstly*, `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.

```
4.161   self.header = {
4.162     magicNumber = readInt(scfFile, 4, 0),
4.163     samplesNumber = readInt(scfFile, 4),
4.164     samplesOffset = readInt(scfFile, 4),
4.165     basesNumber = readInt(scfFile, 4),
4.166     leftClip = readInt(scfFile, 4),
4.167     rightClip = readInt(scfFile, 4),
4.168     basesOffset = readInt(scfFile, 4),
4.169     comments = readInt(scfFile, 4),
4.170     commentsOffset = readInt(scfFile, 4),
4.171     version = readInt(scfFile, 4),
4.172     sampleSize = readInt(scfFile, 4),
4.173     codeSet = readInt(scfFile, 4),
4.174     privateSize = readInt(scfFile, 4),
4.175     privateOffset = readInt(scfFile, 4)
4.176   }
4.177   if self.header.magicNumber ~= 0x2E736366 then
4.178     packageError(
4.179       "Magic number in scf scfFile '" ..
4.180       self.lastScfFile ..
4.181       "' corrupt!"
4.182     )
4.183   end
4.184   if self.header.version ~= 0x332E3030 then
4.185     packageError(
4.186       "Scf scfFile '" ..
4.187       self.lastScfFile ..
4.188       "' is not version 3.00!"
4.189     )
4.190   end
```

Secondly, `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.

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

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

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

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

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

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

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

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

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

```

```
    scfFile:close()
  end
end

```

5.4.4 Set Chromatogram Parameters

`Chromatogram:setParameters` passes options from the chromatogram module to the Lua script. Each field of the table `keyHash` is named after a `Chromatogram` attribute and represents a function that receives one string parameter (the value of a LaTeX key). For instance, `keyHash.sampleRange` extracts the range and step values from the value stored in the `sample range` key.

```
function Chromatogram:setParameters(newParms)
  local keyHash = {
    sampleRange = function(v)
      local sampleRangeMin, sampleRangeMax, sampleRangeStep =
        getRange(
          v:trim(),
          "^([base]*%s*%d+)%s*%-",
          "%-%s*([base]*%s*%d+)",
          "step%s*(%d+)$"
        )
      self.sampleMin, self.peakMin =
        self:getSampleAndPeakIndex(sampleRangeMin, true)
      self.sampleMax, self.peakMax =
        self:getSampleAndPeakIndex(sampleRangeMax, false)
      if self.sampleMin >= self.sampleMax then
        packageError("Sample range is smaller than 1.")
      end
      self.sampleStep = sampleRangeStep or self.sampleStep
    end,
    xUnit = stringToDim,
    yUnit = stringToDim,
    samplesPerLine = tonumber,
    baselineSkip = stringToDim,
    canvasHeight = stringToDim,
    tickLength = stringToDim,
    showBaseNumbers = function(v)
      if v == "true" then return true else return false end
    end,
    baseNumberRange = function(v)
      local baseNumberRangeMin, baseNumberRangeMax, baseNumberRangeStep =
        getRange(
          v:trim(),
          "^([auto%d]*)%s+%-",
          "%-%s+([auto%d]*$)"
        )
```

```
      if tonumber(baseNumberRangeMin) then
        self.baseNumberMin = tonumber(baseNumberRangeMin)
      else
        self.baseNumberMin = self.peakMin
      end
      if tonumber(baseNumberRangeMax) then
        self.baseNumberMax = tonumber(baseNumberRangeMax)
      else
        self.baseNumberMax = self.peakMax
      end
      if self.baseNumberMin >= self.baseNumberMax then
        packageError("Base number range is smaller than 1.")
      end
      if self.baseNumberMin < self.peakMin then
        self.baseNumberMin = self.peakMin
        packageWarning("Lower base number range is smaller than lower sample range. It was adjusted to " .. self.baseNumberMin .. ".")
      end
      if self.baseNumberMax > self.peakMax then
        self.baseNumberMax = self.peakMax
        packageWarning("Upper base number range exceeds upper sample range. It was adjusted to " .. self.baseNumberMax .. ".")
      end
      self.baseNumberStep = tonumber(baseNumberRangeStep)
        or self.baseNumberStep
    end,
    probDistance = stringToDim,
    probStyle = function(v) return v end,
    tracesDrawn = findBasesInStr,
    ticksDrawn = function(v) return v end,
    baseLabelsDrawn = function(v) return v end,
    probabilitiesDrawn = function(v) return v end,
    probStyle = function(v) return v end
  }
```

We iterate over all fields in the argument of `setParameters`. If a field of the same name exists in `keyHash`, we call this field with the value of the corresponding field in `newParms` as parameter.

```
  for key, value in pairs(newParms) do
    if keyHash[key] then
      self[key] = keyHash[key](value)
    end
  end
end

```

5.4.5 Print the Chromatogram

`Chromatogram:printTikzChromatogram` writes all commands that draw the chromatogram to the TEX input stream (via `tex.sprint`), but only if no error has occurred previously.

```
function Chromatogram:printTikzChromatogram()
  if pgfmolbio.errorCatched then return end
```

(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 `selectedPeaks`. `selectedPeaks` is a table of zero to `header.basesNumber` indexed subtables. It is similar to `peaks` but only describes the peaks in the displayed part of the chromatogram, which is selected by the `samples range` key. 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` (section 5.4.3).
- `prob`: See `peaks.prob` (section 5.4.3).
- `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.

```
  self.selectedPeaks = {}
  local tIndex = 1
  for rPeakIndex, currPeak in ipairs(self.peaks) do
    if currPeak.offset >= self.sampleMin
        and currPeak.offset <= self.sampleMax then
      self.selectedPeaks[tIndex] = {
        offset = currPeak.offset + 1 - self.sampleMin,
        base = currPeak.base,
        prob = currPeak.prob,
        baseIndex = rPeakIndex,
        probXRight = self.sampleMax + 1 - self.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
        self.selectedPeaks[tIndex-1].probXRight =
          (self.selectedPeaks[tIndex-1].offset
          + self.selectedPeaks[tIndex].offset) / 2
      end
      tIndex = tIndex + 1
    end
  end

```

Furthermore, we adjust `baseNumberMin` and `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 self.baseNumberMin == -1 then
      self.baseNumberMin = self.selectedPeaks[1].baseIndex
    end
    if self.baseNumberMax == -1 then
      self.baseNumberMax = self.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 = self.sampleMax - self.sampleMin + 1
  local currLine = 0
  while samplesLeft > 0 do
    local yLower = -currLine * self.baselineSkip
    local yUpper = -currLine * self.baselineSkip + self.canvasHeight
    local xRight =
      (math.min(self.samplesPerLine, samplesLeft) - 1) * self.xUnit
    tex.sprint(
      "\n\t\\draw [" .. PGFKEYS_PATH .. "canvas style] (" ..
      dimToString(0) ..
      ", " ..
      dimToString(yLower) ..
      ") rectangle (" ..
      dimToString(xRight) ..
      ", " ..
      dimToString(yUpper) ..
```

```
      ");"
    )
    samplesLeft = samplesLeft - self.samplesPerLine
    currLine = currLine + 1
  end

```

(3) Traces The traces in `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(self.tracesDrawn) do
    tex.sprint("\n\t\\draw [" .. self.traceStyle[baseName] .. "] ")
    local currSampleIndex = self.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.

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

```
      end
```

Besides, we add line breaks at the appropriate positions.

```
      if sampleX ~= self.sampleMax + 1 - self.sampleMin then
        if sampleX >= (currLine + 1) * self.samplesPerLine then
          currLine = currLine + 1
          tex.sprint(";\n\t\\draw [" .. self.traceStyle[baseName] .. "] ")
          firstPointInLine = true
        end
      else
        tex.sprint(";")
      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(self.selectedPeaks) do
    while currPeak.offset > (currLine + 1) * self.samplesPerLine do
      currLine = currLine + 1
    end

    local x = ((currPeak.offset - 1) % self.samplesPerLine) * self.xUnit
    local yUpper = -currLine * self.baselineSkip
    local yLower = -currLine * self.baselineSkip - self.tickLength
    local tickOperation = ""
    if self.ticksDrawn:upper():find(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(
      "\n\t\\draw [" ..
      self.tickStyle[currPeak.base] ..
      "] (" ..
      dimToString(x) ..
      ", " ..
      dimToString(yUpper) ..
      ") " ..
      tickOperation ..
      " (" ..
      dimToString(x) ..
      ", " ..
      dimToString(yLower) ..
      ")"
    )
    if self.baseLabelsDrawn:upper():find(currPeak.base) then
      tex.sprint(
        " node [" ..
        self.baseLabelStyle[currPeak.base] ..
        "] {" ..
        self.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 self.showBaseNumbers
        and currPeak.baseIndex >= self.baseNumberMin
        and currPeak.baseIndex <= self.baseNumberMax
        and (currPeak.baseIndex - self.baseNumberMin)
          % self.baseNumberStep == 0 then
      tex.sprint(
        " node [" ..
        PGFKEYS_PATH ..
        "base number style] {\\strut " ..
        currPeak.baseIndex ..
        "}"
      )
    end
    tex.sprint(";")

```

(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 =
      self.probabilitiesDrawn:upper():find(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 * self.samplesPerLine
    if xLeft < 0 then
      local xLeftPrev = (self.samplesPerLine + xLeft) * self.xUnit
      local xRightPrev = (self.samplesPerLine - 1) * self.xUnit
      local yPrev = -(currLine-1) * self.baselineSkip - self.probDistance
      if drawCurrProb then
        tex.sprint(
          "\n\t\\draw [" ..
          self.probStyle(currPeak.prob[currPeak.base]) ..
          "] (" ..
          dimToString(xLeftPrev) ..
          ", " ..
          dimToString(yPrev) ..
          ") -- (" ..
          dimToString(xRightPrev) ..
          ", " ..
          dimToString(yPrev) ..
          ");"
        )
      end
      xLeft = 0
    else
      xLeft = xLeft * self.xUnit
    end

```

Secondly, the probability rule ends in the right margin of the current line (i. e., `xRight` at least equals `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 * self.samplesPerLine
    if xRight >= self.samplesPerLine then
      if drawCurrProb then
        local xRightNext = (xRight - self.samplesPerLine) * self.xUnit
        local yNext = -(currLine+1) * self.baselineSkip - self.probDistance
        probRemainder =
          "\n\t\\draw [" ..
          self.probStyle(currPeak.prob[currPeak.base]) ..
          "] (" ..
          dimToString(0) ..
          ", " ..
          dimToString(yNext) ..
          ") -- (" ..
          dimToString(xRightNext) ..
          ", " ..
          dimToString(yNext) ..
          ");"
      end
      xRight = (self.samplesPerLine - 1) * self.xUnit
    else
      xRight = xRight * self.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 * self.baselineSkip - self.probDistance
    if drawCurrProb then
      tex.sprint(
        "\n\t\\draw [" ..
        self.probStyle(currPeak.prob[currPeak.base]) ..
        "] (" ..
        dimToString(xLeft) ..
        ", " ..
        dimToString(y) ..
        ") -- (" ..
        dimToString(xRight) ..
        ", " ..
        dimToString(y) ..
        ");"
```

```
      )
    end
    lastProbX = currPeak.probXRight
  end
end
```

5.5 pgfmolbio.domains.tex

```
\ProvidesFile{pgfmolbio.domains.tex}[2012/10/01 v0.2 Protein Domains]

```

If the domains module is requested by LuaTEX it loads the corresponding Lua module and generates a new SpecialKeys object, which will store all feature styles, disulfide keys and print functions (section 5.6.2).

```
\ifluatex
  \RequireLuaModule{pgfmolbio.domains}
  \directlua{pmbSpecialKeys = pgfmolbio.domains.SpecialKeys:new()}
\fi

```

5.5.1 Keys

\@pmb@dom@keydef

#1: ⟨*key*⟩ name
#2: default ⟨*value*⟩

\@pmb@dom@keydef declares a ⟨*key*⟩ in path /pgfmolbio/domains and assigns a default ⟨*value*⟩.

```
\def\@pmb@dom@keydef#1#2{%
  \pgfkeyssetvalue{/pgfmolbio/domains/#1}{#2}%
}

```

\pmbdomvalueof

#1: ⟨*key*⟩ name

\pmbdomvalueof retrieves the value of a ⟨*key*⟩ in path /pgfmolbio/domains. Note that the control word lacks an @ and is thus freely accessible within a LaTeX document (see section 3.4).

```
\def\pmbdomvalueof#1{%
  \pgfkeysvalueof{/pgfmolbio/domains/#1}%
}

```

Aided by these auxiliary macros, we define all keys of the domains module.

```
\@pmb@dom@keydef{name}{Protein}
\newif\ifpmb@dom@showname
\pgfmolbioset[domains]{%
  show name/.is if=pmb@dom@showname,
  show name
}
\@pmb@dom@keydef{description}{}

\@pmb@dom@keydef{x unit}{.5mm}
\@pmb@dom@keydef{y unit}{6mm}
\@pmb@dom@keydef{residues per line}{200}
\@pmb@dom@keydef{baseline skip}{3}
\@pmb@dom@keydef{residue numbering}{auto}
\@pmb@dom@keydef{residue range}{auto-auto}
\@pmb@dom@keydef{enlarge left}{0cm}
\@pmb@dom@keydef{enlarge right}{0cm}
\@pmb@dom@keydef{enlarge top}{1cm}
\@pmb@dom@keydef{enlarge bottom}{0cm}

\pgfmolbioset[domains]{%
  style/.code=\pgfmolbioset[domains]{current style/.style={#1}}
}

\@pmb@dom@keydef{domain font}{\footnotesize}

\@pmb@dom@keydef{level}{}
\@pmb@dom@keydef{disulfide base distance}{1}
\@pmb@dom@keydef{disulfide level distance}{.2}
\@pmb@dom@keydef{range font}{\sffamily\scriptsize}

\newif\ifpmb@dom@showruler
\pgfmolbioset[domains]{%
  show ruler/.is if=pmb@dom@showruler,
  show ruler
}
\@pmb@dom@keydef{ruler range}{auto-auto}
\@pmb@dom@keydef{default ruler step size}{50}
\@pmb@dom@keydef{ruler distance}{-.5}

\@pmb@dom@keydef{sequence}{}
\@pmb@dom@keydef{magnified sequence font}{\ttfamily\footnotesize}

```

```

\newif\ifpmb@dom@showsecstructure
\pgfmolbioset[domains]{%
  show secondary structure/.is if=pmb@dom@showsecstructure,
  show secondary structure=false
}
\@pmb@dom@keydef{secondary structure distance}{1}
\pgfmolbioset[domains]{%
  helix back border color/.code=\colorlet{helix back border color}{#1},
  helix back main color/.code=\colorlet{helix back main color}{#1},
  helix back middle color/.code=\colorlet{helix back middle color}{#1},
  helix front border color/.code=\colorlet{helix front border color}{#1},
  helix front main color/.code=\colorlet{helix front main color}{#1},
  helix front middle color/.code=\colorlet{helix front middle color}{#1},
  helix back border color=white!50!black,
  helix back main color=white!90!black,
  helix back middle color=white,
  helix front border color=red!50!black,
  helix front main color=red!90!black,
  helix front middle color=red!10!white
}

\@pmb@dom@keydef{sequence length}{}

\@pmb@dom@keydef{@layer}{}

```

5.5.2 Feature Shapes

\setfeatureshape

#1: Shape ⟨*name*⟩.
#2: Ti*k*Z ⟨*code*⟩.

Stores the ⟨*code*⟩ for a shape in the macro `\@pmb@dom@feature@`⟨*name*⟩`@shape`.

```
\newcommand\setfeatureshape[2]{%
  \expandafter\def\csname @pmb@dom@feature@#1@shape\endcsname{#2}%
}

```

\setfeatureshapealias

#1: New shape.
#2: Existing shape.

Links a new shape to an existing one.

```
\newcommand\setfeatureshapealias[2]{%
  \expandafter\def\csname @pmb@dom@feature@#1@shape\endcsname{%
    \@nameuse{@pmb@dom@feature@#2@shape}%
  }%
}

```

\setfeaturestylealias

#1: New style.
#2: Existing style.

This macro and the next one are only defined in LuaTEXDepending on whether \setfeaturestylealias occurs within a pmbdomains environment, it either sets the feature styles of the SpecialKeys object in the current Protein (pmbProtein.specialKeys) or of the global SpecialKeys object (pmbSpecialKeys).

\setfeaturealias

#1: New feature.
#2: Existing feature.

Calls \setfeatureshapealias and possibly \setfeaturestylealias.

```
\ifluatex
  \newcommand\setfeaturestylealias[2]{%
    \directlua{
      if pmbProtein then
        pmbProtein.specialKeys:aliasFeatureStyle("#1", "#2")
      else
        pmbSpecialKeys:aliasFeatureStyle("#1", "#2")
      end
    }%
  }
  \newcommand\setfeaturealias[2]{%
    \setfeatureshapealias{#1}{#2}%
    \setfeaturestylealias{#1}{#2}%
  }
\else
  \let\setfeaturealias\setfeatureshapealias%
\fi

```

\pmbdomdrawfeature

#1: The feature ⟨*type*⟩ that should be drawn.

If a feature ⟨*type*⟩ (i. e., the corresponding macro) is undefined, we issue a warning and draw feature `default`.

```
\newcommand\pmbdomdrawfeature[1]{%
  \@ifundefined{@pmb@dom@feature@#1@shape}{%
    \PackageWarning{pgfmolbio}%
      {Feature shape `#1' unknown, using `default'.}%
    \@pmb@dom@feature@default@shape%
  }{%
    \@nameuse{@pmb@dom@feature@#1@shape}%
  }%
}

```

Definitions of standard features and their aliases.

```
\setfeatureshape{default}{%
  \path [/pgfmolbio/domains/current style]
    (\xLeft, \yMid + .5 * \pmbdomvalueof{y unit}) rectangle
    (\xRight, \yMid - .5 * \pmbdomvalueof{y unit});
}

\setfeatureshape{domain}{
  \draw [/pgfmolbio/domains/current style, rounded corners=2pt]
    (\xLeft, \yMid + .5 * \pmbdomvalueof{y unit}) rectangle
    (\xRight, \yMid - .5 * \pmbdomvalueof{y unit});
  \node at (\xMid, \yMid)
    {\pmbdomvalueof{domain font}{\pmbdomvalueof{description}}};
}
\setfeaturealias{DOMAIN}{domain}

\setfeatureshape{signal peptide}{%
  \path [/pgfmolbio/domains/current style]
    (\xLeft, \yMid + \pmbdomvalueof{y unit} / 5) rectangle
    (\xRight, \yMid - \pmbdomvalueof{y unit} / 5);
}
\setfeaturealias{SIGNAL}{signal peptide}

\setfeatureshape{propeptide}{%
  \path [/pgfmolbio/domains/current style]
    (\xLeft, \yMid + .5 * \pmbdomvalueof{y unit}) rectangle
    (\xRight, \yMid - .5 * \pmbdomvalueof{y unit});
}
\setfeaturealias{PROPEP}{propeptide}

\setfeatureshape{carbohydrate}{%
```

```
  \draw [/pgfmolbio/domains/current style]
    (\xMid, \yMid) --
    (\xMid, \yMid + .7 * \pmbdomvalueof{y unit})
    node [above] {\tiny\strut\pmbdomvalueof{description}};
  \fill [/pgfmolbio/domains/current style]
    (\xMid, \yMid + .7 * \pmbdomvalueof{y unit}) circle [radius=1pt];
}
\setfeaturealias{CARBOHYD}{carbohydrate}

\setfeatureshape{other/main chain}{%
  \ifpmb@dom@showsecstructure%
    \pgfmathsetmacro\yUpper{%
      \yMid + \pmbdomvalueof{secondary structure distance}
        * \pmbdomvalueof{y unit}%
    }
    \draw [thin]
      (\xLeft, \yUpper pt) --
      (\xRight, \yUpper pt);%
  \fi%
  \path [/pgfmolbio/domains/current style]
    (\xLeft, \yMid) --
    (\xRight, \yMid);%
}

\setfeatureshape{other/name}{%
  \ifpmb@dom@showname%
    \node [/pgfmolbio/domains/current style]
      at (\xMid, \pmbdomvalueof{baseline skip} * \pmbdomvalueof{y unit})
      {\pmbdomvalueof{name} (\pmbdomvalueof{sequence length} residues)};
  \fi%
}

\setfeatureshape{disulfide}{%
  \pgfmathsetmacro\yUpper{%
    \yMid + (
      \pmbdomvalueof{disulfide base distance} +
      (\pmbdomvalueof{level} - 1) *
      \pmbdomvalueof{disulfide level distance}
    ) * \pmbdomvalueof{y unit}
  }
  \path [/pgfmolbio/domains/current style]
    (\xLeft, \yMid) --
    (\xLeft, \yUpper pt) --
    (\xRight, \yUpper pt) --
    (\xRight, \yMid);
}
\setfeaturealias{DISULFID}{disulfide}

\setfeatureshape{range}{%
```

```
  \pgfmathsetmacro\yUpper{%
    \yMid + (
      \pmbdomvalueof{disulfide base distance} +
      (\pmbdomvalueof{level} - 1) *
      \pmbdomvalueof{disulfide level distance}
    ) * \pmbdomvalueof{y unit}
  }
  \path [/pgfmolbio/domains/current style]
    (\xLeft, \yUpper pt) --
    (\xRight, \yUpper pt)
    node [pos=.5, above]
      {\pmbdomvalueof{range font}{\pmbdomvalueof{description}}};
}

\setfeatureshape{other/ruler}{%
  \draw [/pgfmolbio/domains/current style]
    (\xMid,
      \yMid + \pmbdomvalueof{ruler distance} *
              \pmbdomvalueof{y unit}) --
    (\xMid,
      \yMid + \pmbdomvalueof{ruler distance} *
              \pmbdomvalueof{y unit} - 1mm)
    node [below=-1mm] {\tiny\sffamily\strut\residueNumber};
}


\setfeatureshape{other/sequence}{%
  \node [/pgfmolbio/domains/current style]
    at (\xMid, \yMid) {\strut\currentResidue};
}

\newlength\pmb@magnifiedsequence@width

\setfeatureshape{other/magnified sequence above}{%
  \settowidth\pmb@magnifiedsequence@width{%
    \begin{pgfinterruptpicture}%
      \pmbdomvalueof{magnified sequence font}%
      \featureSequence%
      \end{pgfinterruptpicture}%
    }%
  \pgfmathsetmacro\xUpperLeft{\xMid - \pmb@magnifiedsequence@width / 2}
  \pgfmathsetmacro\xUpperRight{\xMid + \pmb@magnifiedsequence@width / 2}

  \draw [/pgfmolbio/domains/current style]
    (\xLeft, \yMid) --
    (\xLeft, \yMid + \pmbdomvalueof{y unit} / 6) --
    (\xUpperLeft pt, \yMid + \pmbdomvalueof{y unit} * 4/6) --
    (\xUpperLeft pt, \yMid + \pmbdomvalueof{y unit} * 5/6)
    (\xUpperRight pt, \yMid + \pmbdomvalueof{y unit} * 5/6) --
```

```
    (\xUpperRight pt, \yMid + \pmbdomvalueof{y unit} * 4/6) --
    (\xRight, \yMid + \pmbdomvalueof{y unit} / 6) --
    (\xRight, \yMid);
  \node [anchor=mid]
    at (\xMid, \yMid + \pmbdomvalueof{y unit})
    {\pmbdomvalueof{magnified sequence font}\featureSequence};
}

\setfeatureshape{other/magnified sequence below}{%
  \settowidth\pmb@magnifiedsequence@width{%
    \begin{pgfinterruptpicture}%
      \pmbdomvalueof{magnified sequence font}%
      \featureSequence%
      \end{pgfinterruptpicture}%
    }%
  \pgfmathsetmacro\xLowerLeft{\xMid - \pmb@magnifiedsequence@width / 2}
  \pgfmathsetmacro\xLowerRight{\xMid + \pmb@magnifiedsequence@width / 2}

  \draw [/pgfmolbio/domains/current style]
    (\xLeft, \yMid) --
    (\xLeft, \yMid - \pmbdomvalueof{y unit} / 6) --
    (\xLowerLeft pt, \yMid - \pmbdomvalueof{y unit}) --
    (\xLowerLeft pt, \yMid - \pmbdomvalueof{y unit} * 7/6)
    (\xLowerRight pt, \yMid - \pmbdomvalueof{y unit} * 7/6) --
    (\xLowerRight pt, \yMid - \pmbdomvalueof{y unit}) --
    (\xRight, \yMid - \pmbdomvalueof{y unit} / 6) --
    (\xRight, \yMid);
  \node [anchor=mid]
    at (\xMid, \yMid - \pmbdomvalueof{y unit} * 8/6)
    {\pmbdomvalueof{magnified sequence font}\featureSequence};
}


```

5.5.3 Secondary Structure Elements

\@pmb@dom@helixsegment

#1: Scale factor for TikZ's `svg` action.

Draws a full helix segment at the current canvas position. We use the (unusual) `svg` syntax since the helix segment was designed in Inkscape, and the `svg` commands were copied from the resulting vector graphics file.

```
\newcommand\@pmb@dom@helixsegment[1]{%
  svg [scale=#1] "%
    c  0.30427  0
```

```
      0.62523  0.59174
      0.79543  0.96646
   c  0.97673  2.15039
      1.34005  4.49858
      1.84538  6.6178
   c  0.56155  2.35498
      0.99602  4.514
      1.82948  6.72355
   c  0.11069  0.29346
      0.23841  0.69219
      0.56172  0.69219
   l -5        0
   c -0.27235  0.0237
     -0.55793 -0.51373
     -0.65225 -0.76773
   c -0.98048 -2.64055
     -1.40233 -5.46534
     -2.06809 -8.00784
   c -0.50047 -1.91127
     -0.94696 -3.73368
     -1.68631 -5.43929
   c -0.14066 -0.3245
     -0.34516 -0.78514
     -0.69997 -0.78514
   z"
}

```

\@pmb@dom@helixhalfsegment

#1: Scale factor for Ti*k*Z's `svg` action.

Draws a half helix segment.

```
\newcommand\@pmb@dom@helixhalfsegment[1]{%
  svg [scale=#1] "%
   c  0.50663  2.18926
      0.96294  4.51494
      1.78125  6.71875
   c  0.09432  0.254
      0.35265  0.80495
      0.625    0.78125
   l  5        0
   c -0.32331  0
     -0.45181 -0.42529
     -0.5625  -0.71875
   c -0.83346 -2.20955
     -1.2822  -4.36377
     -1.84375 -6.78125
```

```
    l -5        0
    z"
}

```

Shadings for helix segments.

```
\pgfdeclareverticalshading[%
    helix back border color,%
    helix back main color,%
    helix back middle color%
  ]{helix half upper back}{100bp}{
  color(0bp)=(helix back middle color);
  color(5bp)=(helix back middle color);
  color(45bp)=(helix back main color);
  color(75bp)=(helix back border color);
  color(100bp)=(helix back border color)
}

\pgfdeclareverticalshading[%
    helix back border color,%
    helix back main color,%
    helix back middle color%
  ]{helix half lower back}{100bp}{
  color(0bp)=(helix back border color);
  color(25bp)=(helix back border color);
  color(35bp)=(helix back main color);
  color(55bp)=(helix back middle color);
  color(95bp)=(helix back main color);
  color(100bp)=(helix back main color)
}

\pgfdeclareverticalshading[%
    helix back border color,%
    helix back main color,%
    helix back middle color%
  ]{helix full back}{100bp}{
  color(0bp)=(helix back border color);
  color(25bp)=(helix back border color);
  color(30bp)=(helix back main color);
  color(40bp)=(helix back middle color);
  color(60bp)=(helix back main color);
  color(75bp)=(helix back border color);
  color(100bp)=(helix back border color)
}

\pgfdeclareverticalshading[%
    helix front border color,%
    helix front main color,%
    helix front middle color%
```

```
  ]{helix half upper front}{100bp}{
  color(0bp)=(helix front main color);
  color(5bp)=(helix front main color);
  color(45bp)=(helix front middle color);
  color(65bp)=(helix front main color);
  color(75bp)=(helix front border color);
  color(100bp)=(helix front border color)
}

\pgfdeclareverticalshading[%
    helix front border color,%
    helix front main color,%
    helix front middle color%
  ]{helix full front}{100bp}{
  color(0bp)=(helix front border color);
  color(25bp)=(helix front border color);
  color(40bp)=(helix front main color);
  color(60bp)=(helix front middle color);
  color(70bp)=(helix front main color);
  color(75bp)=(helix front border color);
  color(100bp)=(helix front border color)
}

```

The following features print single helical turns. They are drawn with appropriate coordinates by `printHelixFeature` (section 5.6.1).

```
\setfeatureshape{helix/half upper back}{%
  \ifpmb@dom@showsecstructure%
    \pgfmathsetmacro\yShift{%
      \pmbdomvalueof{secondary structure distance} *
      \pmbdomvalueof{y unit}%
    }
    \draw [shading=helix half upper back]
      (\xLeft, \yMid + \yShift pt)
      \@pmb@dom@helixhalfsegment{\pmbdomvalueof{x unit} / 5};
  \fi%
}

\setfeatureshape{helix/half lower back}{%
  \ifpmb@dom@showsecstructure%
    \pgfmathsetmacro\yShift{%
      \pmbdomvalueof{secondary structure distance} *
      \pmbdomvalueof{y unit}%
    }
    \draw [shading=helix half lower back]
      (\xRight, \yMid + \yShift pt) [rotate=180]
      \@pmb@dom@helixhalfsegment{\pmbdomvalueof{x unit} / 5};
  \fi%
}
```

```

\setfeatureshape{helix/full back}{%
  \ifpmb@dom@showsecstructure%
    \pgfmathsetmacro\yShift{%
      \pmbdomvalueof{secondary structure distance} *
      \pmbdomvalueof{y unit}%
    }
    \draw [shading=helix full back]
      (\xMid, \yLower + \yShift pt)
      \@pmb@dom@helixsegment{\pmbdomvalueof{x unit} / 5};
  \fi%
}

\setfeatureshape{helix/half upper front}{%
  \ifpmb@dom@showsecstructure%
    \pgfmathsetmacro\yShift{%
      \pmbdomvalueof{secondary structure distance} *
      \pmbdomvalueof{y unit}%
    }
    \draw [shading=helix half upper front]
      (\xRight, \yMid + \yShift pt) [xscale=-1]
      \@pmb@dom@helixhalfsegment{\pmbdomvalueof{x unit} / 5};
  \fi%
}

\setfeatureshape{helix/full front}{%
  \ifpmb@dom@showsecstructure%
    \pgfmathsetmacro\yShift{%
      \pmbdomvalueof{secondary structure distance} *
      \pmbdomvalueof{y unit}%
    }
    \draw [shading=helix full front]
      (\xMid, \yLower + \yShift pt) [xscale=-1]
      \@pmb@dom@helixsegment{\pmbdomvalueof{x unit} / 5};
  \fi%
}

```

Definitions of the remaining secondary structure features.

```
\definecolor{strand left color}{RGB}{42,127,255}
\definecolor{strand right color}{RGB}{128,179,255}

\setfeatureshape{beta strand}{%
  \ifpmb@dom@showsecstructure%
    \pgfmathsetmacro\yShift{%
      \pmbdomvalueof{secondary structure distance} *
      \pmbdomvalueof{y unit}%
    }
    \draw [/pgfmolbio/domains/current style]
```

```
      (\xLeft, \yMid + \pmbdomvalueof{x unit} + \yShift pt) --
      (\xRight - 1.5 * \pmbdomvalueof{x unit},
        \yMid + \pmbdomvalueof{x unit} + \yShift pt) --
      (\xRight - 1.5 * \pmbdomvalueof{x unit},
        \yMid + 1.5 * \pmbdomvalueof{x unit} + \yShift pt) --
      (\xRight, \yMid + \yShift pt) --
      (\xRight - 1.5 * \pmbdomvalueof{x unit},
        \yMid - 1.5 * \pmbdomvalueof{x unit} + \yShift pt) --
      (\xRight - 1.5 * \pmbdomvalueof{x unit},
        \yMid - \pmbdomvalueof{x unit} + \yShift pt) --
      (\xLeft, \yMid - \pmbdomvalueof{x unit} + \yShift pt) --
      cycle;%
  \fi%
}
\setfeaturealias{STRAND}{beta strand}

\setfeatureshape{beta turn}{%
  \ifpmb@dom@showsecstructure%
    \pgfmathsetmacro\yShift{%
      \pmbdomvalueof{secondary structure distance} *
      \pmbdomvalueof{y unit}%
    }
    \pgfmathsetmacro\turnXradius{(\xRight - \xLeft) / 2}%
    \pgfmathsetmacro\turnYradius{\pmbdomvalueof{x unit} * 1.5}%
    \fill [white]
      (\xLeft, \yMid + 1mm + \yShift pt) rectangle
      (\xRight, \yMid - 1mm + \yShift pt);%
    \draw [/pgfmolbio/domains/current style]
      (\xLeft - .5pt, \yMid + \yShift pt) --
      (\xLeft, \yMid + \yShift pt) arc
        [start angle=180, end angle=0,
        x radius=\turnXradius pt, y radius=\turnYradius pt] --
      (\xRight + .5pt, \yMid + \yShift pt);%
  \fi%
}
\setfeaturealias{TURN}{beta turn}

\setfeatureshape{beta bridge}{%
  \ifpmb@dom@showsecstructure%
    \pgfmathsetmacro\yShift{%
      \pmbdomvalueof{secondary structure distance} *
      \pmbdomvalueof{y unit}%
    }
    \draw [/pgfmolbio/domains/current style]
      (\xLeft, \yMid + .25 * \pmbdomvalueof{x unit} + \yShift pt) --
      (\xRight - 1.5 * \pmbdomvalueof{x unit},
        \yMid + .25 * \pmbdomvalueof{x unit} + \yShift pt) --
      (\xRight - 1.5 * \pmbdomvalueof{x unit},
        \yMid + 1.5 * \pmbdomvalueof{x unit} + \yShift pt) --
```

```
      (\xRight, \yMid + \yShift pt) --
      (\xRight - 1.5 * \pmbdomvalueof{x unit},
        \yMid - 1.5 * \pmbdomvalueof{x unit} + \yShift pt) --
      (\xRight - 1.5 * \pmbdomvalueof{x unit},
        \yMid - .25 * \pmbdomvalueof{x unit} + \yShift pt) --
      (\xLeft, \yMid - .25 * \pmbdomvalueof{x unit} + \yShift pt) --
      cycle;%
  \fi%
}

\setfeatureshape{bend}{%
  \ifpmb@dom@showsecstructure%
    \pgfmathsetmacro\yShift{%
      \pmbdomvalueof{secondary structure distance} *
      \pmbdomvalueof{y unit}%
    }
    \fill [white]
      (\xLeft, \yMid + 1mm + \yShift pt) rectangle
      (\xRight, \yMid - 1mm + \yShift pt);%
    \draw [/pgfmolbio/domains/current style]
      (\xLeft - .5pt, \yMid + \yShift pt) --
      (\xLeft, \yMid + \yShift pt) --
      (\xMid, \yMid + .5 * \pmbdomvalueof{y unit} + \yShift pt) --
      (\xRight, \yMid + \yShift pt) --
      (\xRight + .5pt, \yMid + \yShift pt);%
  \fi%
}

```

This concludes the part of the package that is always loaded. The remaining code is only executed within LuaTEX.

```
\ifluatex\else\expandafter\endinput\fi


```

5.5.4 Adding Features

\pmb@dom@inputuniprot

#1: The ⟨*name*⟩ of a Uniprot file.

\pmb@dom@inputuniprot reads some attributes and all features from a Uniprot file (readUniprotFile, section 5.6.4). It then updates some keys of the domains module (getParameters, section 5.6.5) and then passes the value of residue numbering to the pmbProtein object.

```
\newcommand\pmb@dom@inputuniprot[1]{%
  \directlua{
    pmbProtein:readUniprotFile("#1")
    pmbProtein:getParameters()
    pmbProtein:setParameters{
      residueNumbering = "\pmbdomvalueof{residue numbering}"
    }
  }%
}

```

\pmb@dom@inputgff

#1: The ⟨*name*⟩ of a General Feature Format (gff) file.

This macro reads all features from a gff file (readGffFile, section 5.6.4). It then passes the value of residue numbering to pmbProtein.

```
\newcommand\pmb@dom@inputgff[1]{%
  \directlua{
    pmbProtein:readGffFile("#1")
    pmbProtein:setParameters{
      residueNumbering = "\pmbdomvalueof{residue numbering}"
    }
  }%
}

```

\pmb@dom@addfeature

#1: A ⟨*key-value list*⟩ that is locally applied to the feature.
#2: The feature ⟨*key*⟩.
#3: The ⟨*first*⟩ …
#4: and ⟨*last*⟩ residue covered by the feature.

This macro adds a feature to pmbProtein by calling its addFeature method. The ⟨*key-value list*⟩ should be stored without any expansion in the kvList field of addFeature's single argument table. To this end, we first store the ⟨*key-value list*⟩ in the token register \@pmb@toksa and then access its contents by the construction \directlua{[...]\the\@pmb@toksa[...]}. This code behaves similarly to \the inside an \edef, i. e. the contents of the token register are not further expanded.

```
\newcommand\pmb@dom@addfeature[4][]{%
  \begingroup%
  \pgfmolbioset[domains]{#1}%
  \@pmb@toksa{#1}%
```

```
  \directlua{
    pmbProtein:addFeature{
      key = "#2",
      start = "#3",
      stop = "#4",
      kvList = "\luaescapestring{\the\@pmb@toksa}",
      level = tonumber("\pmbdomvalueof{level}"),
      layer = tonumber("\pmbdomvalueof{@layer}")
    }
  }%
  \endgroup%
}

```

5.5.5 The Main Environment

pmbdomains

#1: A ⟨*key-value list*⟩ that configures the domain diagram.
#2: The ⟨*sequence length*⟩.

If pmbdomains appears outside of a tikzpicture, we implicitly start this environment, otherwise we begin a new group. "Within a tikzpicture" means that \useasboundingbox is defined. The ⟨*key-value list*⟩ is processed.

```
\newif\ifpmb@dom@tikzpicture

\newenvironment{pmbdomains}[2][]{%
  \@ifundefined{useasboundingbox}%
    {\pmb@dom@tikzpicturefalse\begin{tikzpicture}}%
    {\pmb@dom@tikzpicturetrue}%
  \pgfmolbioset[domains]{sequence length=#2, #1}%
```

The macros \inputuniprot, \inputgff and \addfeature only point to their respective internal macros (section 5.5.4) within pmbdomains.

```
  \let\inputuniprot\pmb@dom@inputuniprot%
  \let\inputgff\pmb@dom@inputgff%
  \let\addfeature\pmb@dom@addfeature%
```

pmbProtein is a new Protein object whose specialKeys attribute is initialized with the values from the package-wide SpecialKeys object. Since pmbProtein must know the sequence length and residue numbering before the environment's body is processed, we call setParameters twice to ensure that sequenceLength is set prior to residueNumbering.

```
  \directlua{
    pmbProtein = pgfmolbio.domains.Protein:new()
```

```
    pmbProtein.specialKeys =
      pgfmolbio.domains.SpecialKeys:new(pmbSpecialKeys)
    pmbProtein:setParameters{
      sequenceLength = "\pmbdomvalueof{sequence length}"
    }
    pmbProtein:setParameters{
      residueNumbering = "\pmbdomvalueof{residue numbering}"
    }
  }%
}{%
```

At the end of pmbdomains's body, pmbProtein stores all features that have been defined there. We add one more feature, other/main chain, which spans the whole protein and occupies the lowermost layer (this is the only instance where we need the @layer key).

```
  \pmb@dom@addfeature[@layer=1]{other/main chain}%
    {(1)}{(\pmbdomvalueof{sequence length})}%
```

The following syntactical gem ensures that the token register \@pmb@toksa contains the value of the name key without expansion of any macros within the value.

```
  \@pmb@toksa=%
    \expandafter\expandafter\expandafter\expandafter%
    \expandafter\expandafter\expandafter{%
      \pgfkeysvalueof{/pgfmolbio/domains/name}%
    }%
```

Set the remaining attributes of pmbProtein.

```
  \directlua{
    pmbProtein:setParameters{
      residueRange = "\pmbdomvalueof{residue range}",
      defaultRulerStepSize = "\pmbdomvalueof{default ruler step size}"
    }
    pmbProtein:setParameters{
      name = "\luaescapestring{\the\@pmb@toksa}",
      xUnit = "\pmbdomvalueof{x unit}",
      yUnit = "\pmbdomvalueof{y unit}",
      residuesPerLine = "\pmbdomvalueof{residues per line}",
      baselineSkip = "\pmbdomvalueof{baseline skip}",
      showRuler = "\ifpmb@dom@showruler true\else false\fi",
      rulerRange = "\pmbdomvalueof{ruler range}",
      sequence = "\pmbdomvalueof{sequence}"
    }
```

Calculate he appropriate levels of disulfide-like features (section 5.6.7). pgfmolbio.setCoordinateFormat sets the coordinate output format (section 5.2).

```
    pmbProtein:calculateDisulfideLevels()
```

```
    pgfmolbio.setCoordinateFormat(
      "\pgfkeysvalueof{/pgfmolbio/coordinate unit}",
      "\pgfkeysvalueof{/pgfmolbio/coordinate format string}"
    )
```

If the convert module is loaded, we open the appropriate output file. If we wish to output final TikZ code, we change `tex.sprint` so that the function writes to this file and then call `printTikzDomains`. Otherwise, we write a string representation of `pmbProtein` to the file (section 5.6.9). Without the convert module, `printTikzDomains` simply returns the drawing commands for the chromatogram to the TEX input stream (section 5.6.8).

```
    \ifpmb@loadmodule@convert
      local filename =
        "\pgfkeysvalueof{/pgfmolbio/convert/output file name}"
      if filename == "(auto)" then
        filename = "pmbconverted" .. pgfmolbio.outputFileId
      end
      filename = filename ..
        ".\pgfkeysvalueof{/pgfmolbio/convert/output file extension}"
      outputFile, ioError = io.open(filename, "w")
      if ioError then
        tex.error(ioError)
      end
      \ifpmb@con@outputtikzcode
        tex.sprint = function(a) outputFile:write(a) end
        pmbProtein:getParameters()
        tex.sprint("\string\n\string\\begin{tikzpicture}")
        pmbProtein:printTikzDomains()
        tex.sprint("\string\n\string\\end{tikzpicture}")
      \else
        \ifpmb@con@includedescription
          pmbProtein.includeDescription = true
        \fi
        outputFile:write(tostring(pmbProtein))
      \fi
      outputFile:close()
      pgfmolbio.outputFileId = pgfmolbio.outputFileId + 1
    \else
      pmbProtein:printTikzDomains()
    \fi
    pmbProtein = nil
  }%
```

At the end of `pmbdomains`, we close an implicitly added `tikzpicture`.

```
  \ifpmb@dom@tikzpicture\else\end{tikzpicture}\fi%
}
```

5.5.6 Feature Styles

\setdisulfidefeatures

#1: A list of ⟨*features*⟩.

Clears the list of disulfide-like features and adds the ⟨*features*⟩ to the empty list. Disulfide-like features are arranged in non-overlapping layers (section 3.6). Depending on whether this macro appears inside a pmbdomains environment or not, the appropriate methods of either pmbProtein.specialKeys or pmbSpecialKeys are called, respectively.

```
\newcommand\setdisulfidefeatures[1]{%
  \directlua{
    if pmbProtein then
      pmbProtein.specialKeys:clearKeys("disulfideKeys")
      pmbProtein.specialKeys:setKeys("disulfideKeys", "#1", true)
    else
      pmbSpecialKeys:clearKeys("disulfideKeys")
      pmbSpecialKeys:setKeys("disulfideKeys", "#1", true)
    end
  }%
}

```

\adddisulfidefeatures

#1: A list of ⟨*features*⟩.

Adds the ⟨*features*⟩ to the list of disulfide-like features without overwriting the current list.

```
\newcommand\adddisulfidefeatures[1]{%
  \directlua{
    if pmbProtein then
      pmbProtein.specialKeys:setKeys("disulfideKeys", "#1", true)
    else
      pmbSpecialKeys:setKeys("disulfideKeys", "#1", true)
    end
  }%
}

```

\removedisulfidefeatures

#1: A list of ⟨*features*⟩.

Removes the ⟨*features*⟩ from the list of disulfide-like features.

```
\newcommand\removedisulfidefeatures[1]{%
  \directlua{
    if pmbProtein then
      pmbProtein.specialKeys:setKeys("disulfideKeys", "#1", nil)
    else
      pmbSpecialKeys:setKeys("disulfideKeys", "#1", nil)
    end
  }%
}

```

Declare the default disulfide-like features.

```
\setdisulfidefeatures{DISULFID, disulfide, range}

```

\setfeatureprintfunction

#1: A ⟨*list*⟩ of features.
#2: Name of a Lua ⟨*function*⟩.

Assigns a feature print ⟨*function*⟩ to each feature in the ⟨*list*⟩. Feature print functions are preprocessors which, for instance, calculate coordinates for features (section 3.8).

```
\newcommand\setfeatureprintfunction[2]{%
  \directlua{
    if pmbProtein then
      pmbProtein.specialKeys:setKeys("printFunctions", "#1", #2)
    else
      pmbSpecialKeys:setKeys("printFunctions", "#1", #2)
    end
  }%
}

```

\removefeatureprintfunction

#1: A ⟨*list*⟩ of features.

Removes any feature print function from the features in the ⟨*list*⟩.

```
\newcommand\removefeatureprintfunction[1]{%
  \directlua{
    if pmbProtein then
      pmbProtein.specialKeys:setKeys("printFunctions", "#1", nil)
    else
      pmbSpecialKeys:setKeys("printFunctions", "#1", nil)
    end
```

```
  }%
}

```

Assign default feature print functions.

```
\setfeatureprintfunction{other/sequence}%
  {pgfmolbio.domains.printSequenceFeature}
\setfeatureprintfunction{alpha helix, pi helix, 310 helix, HELIX}%
  {pgfmolbio.domains.printHelixFeature}

```

\setfeaturestyle

#1: A ⟨*feature*⟩ name.
#2: A ⟨*style list*⟩.

Sets the style of a ⟨*feature*⟩ to the style described in the ⟨*style list*⟩. Note that the contents of ⟨*style list*⟩ are passed to the Lua function without expansion (via the token register \@pmb@toksa).

```
\newcommand\setfeaturestyle[2]{%
  \@pmb@toksa{#2}%
  \directlua{
    if pmbProtein then
      pmbProtein.specialKeys:setFeatureStyle(
        "#1", "\luaescapestring{\the\@pmb@toksa}"
      )
    else
      pmbSpecialKeys:setFeatureStyle(
        "#1", "\luaescapestring{\the\@pmb@toksa}"
      )
    end
  }%
}

```

Declare default feature styles.

```
\setfeaturestyle{default}{draw}
\setfeaturestyle{domain}%
  {fill=Chartreuse,fill=LightSkyBlue,fill=LightPink,fill=Gold!50}
\setfeaturestyle{signal peptide}{fill=black}
\setfeaturestyle{propeptide}%
  {*1{fill=Gold, opacity=.5, rounded corners=4pt}}
\setfeaturestyle{carbohydrate}{red}
\setfeaturestyle{other/main chain}{*1{draw, line width=2pt, black!25}}
\setfeaturestyle{other/name}{font=\sffamily}
\setfeaturestyle{disulfide}{draw=olive}
```

```
\setfeaturestyle{range}{*1{draw,decorate,decoration=brace}}
\setfeaturestyle{other/ruler}{black, black!50}
\setfeaturestyle{other/sequence}{*1{font=\ttfamily\tiny}}%
\setfeaturestyle{other/magnified sequence above}%
  {*1{draw=black!50, densely dashed}}
\setfeaturestylealias{other/magnified sequence below}%
  {other/magnified sequence above}
\setfeaturestyle{alpha helix}{%
  *1{helix front border color=red!50!black,%
  helix front main color=red!90!black,%
  helix front middle color=red!10!white}%
}
\setfeaturestylealias{HELIX}{alpha helix}
\setfeaturestyle{pi helix}{%
  *1{helix front border color=yellow!50!black,%
  helix front main color=yellow!70!red,%
  helix front middle color=yellow!10!white}%
}
\setfeaturestyle{310 helix}{%
  *1{helix front border color=magenta!50!black,%
  helix front main color=magenta!90!black,%
  helix front middle color=magenta!10!white}%
}
\setfeaturestyle{beta strand}{%
  *1{left color=strand left color, right color=strand right color}%
}
\setfeaturestyle{beta turn}{*1{draw=violet, thick}}
\setfeaturestyle{beta bridge}{*1{fill=MediumBlue}}
\setfeaturestyle{bend}{*1{draw=magenta, thick}}
```

5.6 pgfmolbio.domains.lua

```
if luatexbase then
  luatexbase.provides_module({
    name          = "pgfmolbio.domains",
    version       = "0.21a",
    date          = "2014/06/17",
    description   = "Domain graphs",
    author        = "Wolfgang Esser-Skala",
    copyright     = "Wolfgang Esser-Skala",
    license       = "LPPL",
  })
end

```

These local functions point to functions in `pgfmolbio.lua` (section 5.2).

```
local stringToDim = pgfmolbio.stringToDim
local dimToString = pgfmolbio.dimToString
local packageError = pgfmolbio.packageError
local packageWarning = pgfmolbio.packageWarning
local getRange = pgfmolbio.getRange

```

5.6.1 Predefined Feature Print Functions

`printSequenceFeature` prints the letters of a sequence between the x-coordinates `xLeft` and `xRight`.

```
function printSequenceFeature(feature, xLeft, xRight, yMid, xUnit, yUnit)
  xLeft = xLeft + 0.5
  for currResidue in feature.sequence:gmatch(".") do
    tex.sprint("\n\t\t\\def\\xMid{" .. dimToString(xLeft * xUnit) .. "}")
    tex.sprint("\n\t\t\\def\\yMid{" .. dimToString(yMid * yUnit) .. "}")
    tex.sprint("\n\t\t\\def\\currentResidue{" .. currResidue .. "}")
    tex.sprint("\n\t\t\\pmbdomdrawfeature{other/sequence}")
    xLeft = xLeft + 1
  end
end

```

`printHelixFeature` prints a helix feature between the x-coordinates `xLeft` and `xRight`.

```
function printHelixFeature(feature, xLeft, xRight, yMid, xUnit, yUnit)
  local residuesLeft, currX
  tex.sprint("\n\t\t\\pgfmolbioset[domains]{current style}")

```

Firstly, three different background parts are drawn: one `half upper back` at the left, zero or more `full back` in the middle and possibly one `half lower back` at the right.

```
  residuesLeft = feature.stop - feature.start + 1
  currX = xLeft
  tex.sprint("\n\t\t\\def\\xLeft{" .. dimToString(currX * xUnit) .. "}")
  tex.sprint("\n\t\t\\def\\yMid{" .. dimToString(yMid * yUnit) .. "}")
  tex.sprint("\n\t\t\\pmbdomdrawfeature{helix/half upper back}")
  residuesLeft = residuesLeft - 2
  currX = currX + 2.5

  while residuesLeft > 0 do
    if residuesLeft == 1 then
      tex.sprint(
```

```
        "\n\t\t\\def\\xRight{" ..
        dimToString((currX + 0.5) * xUnit) ..
        "}"
      )
      tex.sprint("\n\t\t\\def\\yMid{" .. dimToString(yMid * yUnit) .. "}")
      tex.sprint("\n\t\t\\pmbdomdrawfeature{helix/half lower back}")
    else
      tex.sprint("\n\t\t\\def\\xMid{" .. dimToString(currX * xUnit) .. "}")
      tex.sprint(
        "\n\t\t\\def\\yLower{" ..
        dimToString(yMid * yUnit - 1.5 * xUnit) ..
        "}"
      )
      tex.sprint("\n\t\t\\pmbdomdrawfeature{helix/full back}")
    end
    residuesLeft = residuesLeft - 2
    currX = currX + 2
  end

```

Secondly, two different foreground parts are drawn: at least one `full front` at the left and in the middle, and possibly one `half upper front` at the right.

```
  residuesLeft = feature.stop - feature.start
  currX = xLeft + 1.5
  while residuesLeft > 0 do
    if residuesLeft == 1 then
      tex.sprint(
        "\n\t\t\\def\\xRight{" ..
        dimToString((currX + 0.5) * xUnit) ..
        "}"
      )
      tex.sprint("\n\t\t\\def\\yMid{" .. dimToString(yMid * yUnit) .. "}")
      tex.sprint("\n\t\t\\pmbdomdrawfeature{helix/half upper front}")
    else
      tex.sprint("\n\t\t\\def\\xMid{" .. dimToString(currX * xUnit) .. "}")
      tex.sprint(
        "\n\t\t\\def\\yLower{" ..
        dimToString(yMid * yUnit - 1.5 * xUnit) ..
        "}"
      )
      tex.sprint("\n\t\t\\pmbdomdrawfeature{helix/full front}")
    end
    residuesLeft = residuesLeft - 2
    currX = currX + 2
  end
end

```

5.6.2 The SpecialKeys Class

The SpecialKeys class contains three member variables: disulfideKeys (a list of keys that indicate disulfide-like features, like disulfide), featureStyles (a list of feature styles) and printFunctions (a list of keys associated with a feature print function, like alpha helix). Furthermore, it provides methods to manipulate these fields.

The constructor SpecialKeys:new generates a new SpecialKeys object and initializes it with values from parms.

```
SpecialKeys = {}

function SpecialKeys:new(parms)
  parms = parms or {}
  local newSpecialKeys = {
    disulfideKeys = {},
    featureStyles = {},
    printFunctions = {}
  }

  for keyList, listContents in pairs(parms) do
    for key, value in pairs(listContents) do
      newSpecialKeys[keyList][key] = value
    end
  end

  setmetatable(newSpecialKeys, self)
  self.__index = self
  return newSpecialKeys
end

```

SpecialKeys:setKeys sets a value for a key in the keylist. Possible values for keyList are "disulfideKeys", "featureStyles" or "printFunctions".

```
function SpecialKeys:setKeys(keylist, keys, value)
  for key in keys:gmatch("([^,]+)") do
    key = key:trim()
    self[keylist][key] = value
  end
end

```

SpecialKeys:setFeatureStyle parses the style list style and associates it with a certain key. In Lua, a style list is an array of tables. Each table contains the fields cycles and style. cycles determines how often the style (a string suitable for the mandatory argument of \pgfmolbioset) is to be used. In addition, an optional

field `alias` contains a reference to another key, if the current key is an alias of it (see below).

```
function SpecialKeys:setFeatureStyle(key, style)
  local newStyleList, styleCycles, styleContents

  newStyleList = {}
  while style ~= "" do
    styleCycles = 1
    if style:sub(1,1) == "{" then
      styleContents = style:match("%b{}")
      style = style:match("%b{}(.*)")
    elseif style:sub(1,1) == "*" then
      styleCycles, styleContents = style:match("%*(%d*)(%b{})")
      if styleCycles == "" then styleCycles = 1 end
      style = style:match("%*%d*%b{}(.*)")
    elseif style:sub(1,1) == "," or style:sub(1,1) == " " then
      style = style:match("[,%s]+(.*)")
      styleCycles, styleContents = nil, nil
    else
      styleContents = style:match("([^,]+),")
      if not styleContents then
        styleContents = style
        style = ""
      else
        style = style:match("[^,]+,(.*)")
      end
    end
    if styleCycles then
      table.insert(
        newStyleList,
        {cycles = styleCycles, style = styleContents}
      )
    end
  end
  self.featureStyles[key] = newStyleList
end

```

`SpecialKeys:aliasFeatureStyle` sets the `alias` field of a style list so that feature `newKey` uses the same feature style as feature `oldKey`.

```
function SpecialKeys:aliasFeatureStyle(newKey, oldKey)
  self.featureStyles[newKey] = {alias = oldKey}
end

```

`SpecialKeys:getBaseKey` returns either the name of `key` itself or of its parent key if `key` is an alias.

```lua
function SpecialKeys:getBaseKey(key)
  if self.featureStyles[key] then
    if self.featureStyles[key].alias then
      return self.featureStyles[key].alias
    end
  end
  return key
end

```

`SpecialKeys:clearKeys` clears a `keylist`.

```lua
function SpecialKeys:clearKeys(keylist)
  self[keylist] = {}
end

```

`SpecialKeys:selectStyleFromList` returns the `styleID`-th style from the style list associated with `key`. Firstly, the correct style list is selected.

```lua
function SpecialKeys:selectStyleFromList(key, styleID)
  local styleList

  if not self.featureStyles[key] then
    packageWarning(
      "Feature style `" ..
      key ..
      "' unknown, using `default'."
      )
    styleList = self.featureStyles.default
  elseif self.featureStyles[key].alias then
    styleList = self.featureStyles[self.featureStyles[key].alias]
  else
    styleList = self.featureStyles[key]
  end

```

Secondly, the method choses the appropriate style in the list.

```lua
  while true do
    for _, v in ipairs(styleList) do
      styleID = styleID - v.cycles
      if styleID < 1 then
        return v.style
      end
    end
  end
end

```

5.6.3 The `Protein` Class

The `Protein` class represents a domain diagram in Lua. Its member variables largely correspond to the keys of the domains module. In detail:

- `sequenceLength`: A value of `-1` indicates that the sequence length has not been properly set.
- `ft` is the feature table, i. e. an array of tables with the following fields:
  - `key`: A string that equals the feature key.
  - `start`: The start …
  - `stop`: … and the end residue of the feature, both in *absolute* numbering. (For the difference between absolute and relative numbering, see section 3.3.)
  - `kvList`: A string containing comma-separated key-value pairs, which is passed to `\pgfmolbioset` immediately before the feature is drawn.
  - `level`: The level of the feature (only relevant for disulfide-like features).
- `residueNumbering`: An array of strings. The indices are absolute residue numbers, while the fields represent the corresponding relative residue numbers.
- `revResidueNumbering`: The inverse of `residueNumbering` (i. e., a table of numbers).
- `rulerRange`: An array of tables. Each table represents one mark of the ruler and has the fields `pos` (position in absolute residue numbers) and `number` (relative number of the marked residue).
- `currentStyle`: A table whose field names equal feature keys. Each field denotes the index of the style that was last selected from that feature's style list.
- `includeDescription`: This boolean field remains uninitialized. Instead, it is directly set in `pgfmolbio.domains.tex` if the convert module is loaded and the user requests a string representation of a `Protein` object (section 5.6.9).

The constructor `Protein:new` initializes the member variables with default values.

```
Protein = {}

function Protein:new()
  local newProtein = {
    name = "",
    sequenceLength = -1,
    ft = {},
    sequence = "",
    xUnit = stringToDim("0.5mm"),
```

```
    yUnit = stringToDim("6mm"),
    residuesPerLine = 250,
    residueRangeMin = 1,
    residueRangeMax = 100,
    residueNumbering = {},
    revResidueNumbering = {},
    baselineSkip = 3,
    rulerRange = {},
    defaultRulerStepSize = 50,
    showRuler = true,
    currentStyle = {},
    specialKeys = SpecialKeys:new()
  }
  setmetatable(newProtein, self)
  self.__index = self
  return newProtein
end

```

`Protein:toAbsoluteResidueNumber` converts a string that either contains an absolute or relative residue number to an absolute residue number.

```
function Protein:toAbsoluteResidueNumber(value)
  local result = value:match("%b()")
  if result then
    result = tonumber(result:sub(2, -2))
  else
    result = self.revResidueNumbering[(value:gsub("[<>%?]", ""))]
  end
  if not result then
    packageError("Bad or missing start/end point value: " .. value)
  end
  return result
end

```

5.6.4 Uniprot and GFF Files

`Protein:readUniprotFile` reads the relevant parts of Uniprot file `filename`[2].

```
function Protein:readUniprotFile(filename)
  local uniprotFile, errorMsg = io.open(filename, "r")
  if not uniprotFile then packageError(errorMsg) end

```

[2]For a detailed description of this format, see `http://web.expasy.org/docs/userman.html`.

Each line in a Uniprot file starts with a line code consisting of two letters. This code determines the syntax of the remainder of the line.

```
6.239 local sequence = {}
6.240 local inSequence = false
6.241 local featureTable = {}
6.242
6.243 for currLine in uniprotFile:lines() do
6.244   local lineCode = currLine:sub(1, 2)
6.245   local lineContents = currLine:sub(3)
```

The `ID` line is the first line in a Uniprot file. It provides two relevant properties of the protein, namely its name and ints sequence length. For example, in the file `SampleUniprot.txt` (see section 3.10), the `ID` line reads

```
ID   TestProtein  Reviewed;         200 AA.
```

which declares a protein with 200 residues called TestProtein.

```
6.246   if lineCode == "ID" then
6.247     local name, sequenceLength =
6.248       lineContents:match("%s*(%S+)%s*%a+;%s*(%d+)%s*AA%.")
6.249     self.name = name
6.250     self.sequenceLength = tonumber(sequenceLength)
6.251     self.residueRangeMax = self.sequenceLength
```

`FT` lines describe features of the protein (domains, disulfides, sugars etc.). The first line of a feature always contains its key (columns 6–13) and endpoints (columns 15–20 and 22–27, respectively). The description (columns 35–75) may span several lines, in which case the key columns of consecutive lines are empty. For instance,

```
FT   DOMAIN        10      40        Domain 1
```

declares a `DOMAIN` feature between residues 10 and 40 with description "Domain 1".

```
6.252   elseif lineCode == "FT" then
6.253     local key = currLine:sub(6, 13):trim()
6.254     local start, stop, description =
6.255       currLine:sub(15, 20), currLine:sub(22, 27), currLine:sub(35, 75)
6.256     if key ~= "" then
6.257       table.insert(featureTable, {
6.258         key = key,
6.259         start = "(" .. start .. ")",
6.260         stop = "(" .. stop .. ")",
6.261         description = description,
6.262         style = "",
6.263         kvList = ""
6.264       })
6.265     else
6.266       featureTable[#featureTable].description =
6.267         featureTable[#featureTable].description .. description
6.268     end
```

The SQ line starts the sequence block. Each of the following sequence data lines lacks a line code and shows the amino acid sequence in one letter code, e. g.

```
SQ   SEQUENCE   200 AA;   22041 MW;  00A52FE2EC5431D9 CRC64;
     MGSKRSVPSR HRSLTTYEVM FAVLFVILVA LCAGLIAVSW LSIQ […]
```

```
    elseif lineCode == "SQ" then
      inSequence = true
    elseif lineCode == "  " and inSequence then
      table.insert(sequence, (lineContents:gsub("%s+", "")))
```

The \\ line terminates the Uniprot file.

```
    elseif lineCode == "\\\\" then
      break
    end
  end
```

After closing the file, features are converted to the proper format (section 5.6.3).

```
  uniprotFile:close()
  if next(sequence) then self.sequence = table.concat(sequence) end
  for _, v in ipairs(featureTable) do self:addFeature(v) end
end

```

Protein:readGffFile reads the relevant parts of General Feature Format file filename[3].

```
function Protein:readGffFile(filename)
  local gffFile, errorMsg = io.open(filename, "r")
  local lineContents, fields, lineNumber

  if not gffFile then packageError(errorMsg) end
```

Each line in a gff file describes a feature and consists of up to 9 tabulator-separated fields, of which only fields 3 (key), 4 (start) and 5 (end) are required for the domains module. Everything following the comment sign (#) on a line is ignored.

```
  lineNumber = 1
  for currLine in gffFile:lines() do
    lineContents = currLine:gsub("#.*$", "")
    fields = {}
    if lineContents ~= "" then
      for currField in lineContents:gmatch("([^\t]+)") do
        table.insert(fields, currField)
      end
```

[3]For a detailed description of this format, see http://http://www.sanger.ac.uk/resources/software/gff/spec.html.

```
      if not fields[5] then
        packageError("Bad line (" .. lineNumber .. ") in gff file '" ..
          filename .. "':\n" .. currLine)
        break
      end
      self:addFeature{
        key = fields[3],
        start = "(" .. fields[4] .. ")",
        stop = "(" .. fields[5] .. ")",
        description = fields[9] or "",
        style = "",
        kvList = ""
      }
    end
    lineNumber = lineNumber + 1
  end
  gffFile:close()
end

```

5.6.5 Getter and Setter Methods

`Protein:getParameters` informs TEX of the protein name, sequence and sequence length. This method is called after reading a Uniprot file (section 5.5.4).

```
function Protein:getParameters()
  tex.sprint(
    "\\pgfmolbioset[domains]{name={" ..
    self.name ..
    "},sequence={" ..
    self.sequence ..
    "},sequence length=" ..
    self.sequenceLength ..
    "}"
  )
end

```

`Protein:setParameters` passes options from the domains module to the Lua script. Each field of the table `keyHash` is named after a `Protein` attribute and represents a function that receives one string parameter (the value of a LATEX key).

```
function Protein:setParameters(newParms)
  local keyHash = {
```

`keyHash.sequenceLength` checks for an invalid sequence length.

```
6.328    sequenceLength = function(v)
6.329      v = tonumber(v)
6.330      if not v then return self.sequenceLength end
6.331      if v < 1 then
6.332        packageError("Sequence length must be larger than zero.")
6.333      end
6.334      return v
6.335    end,
```

`keyHash.residueNumbering` generates the residue numbering array and its inverse (described in section 5.6.3).

```
6.336    residueNumbering = function(v)
6.337      local ranges = {}
6.338      local start, startNumber, startLetter, stop
6.339      self.revResidueNumbering = {}
6.340      if v:trim() == "auto" then
6.341        for i = 1, self.sequenceLength do
6.342          table.insert(ranges, tostring(i))
6.343        end
6.344      else --example list: `1-4,5,6A-D'
6.345        for _, value in ipairs(v:explode(",+")) do
6.346          value = value:trim()
6.347          start, stop = value:match("(%w*)%s*%-%s*(%w*)$")
6.348          if not start then
6.349            start = value:match("(%w*)")
6.350          end
6.351          if not start or start == "" then --invalid range
6.352            packageError("Unknown residue numbering range: " .. value)
6.353          end
6.354          if stop then
6.355            if tonumber(start) and tonumber(stop) then
6.356              --process range `1-4'
6.357              for currNumber = tonumber(start), tonumber(stop) do
6.358                table.insert(ranges, tostring(currNumber))
6.359              end
6.360            else --process range `6A-D'
6.361              startNumber, startLetter = start:match("(%d*)(%a)")
6.362              stop = stop:match("(%a)")
6.363              for currLetter = startLetter:byte(), stop:byte() do
6.364                table.insert(ranges,
6.365                  startNumber .. string.char(currLetter))
6.366              end
6.367            end
6.368          else --process range `5'
6.369            table.insert(ranges, start)
6.370          end
6.371        end
```

```
      end
      for i, value in ipairs(ranges) do
        if self.revResidueNumbering[value] then
          packageError("The range value " .. value ..
            " appears more than once.")
        else
          self.revResidueNumbering[value] = i
        end
      end
      return ranges
    end,
```

`keyHash.residueRange` sets the residue range, treating possible errors.

```
    residueRange = function(v)
      local num
      local residueRangeMin, residueRangeMax =
        getRange(v:trim(), "^([%w%(%)]+)%s*%-", "%-%s*([%w%(%)]+)$")
      if residueRangeMin == "auto" then
        self.residueRangeMin = 1
      else
        num = residueRangeMin:match("%b()")
        if num then
          self.residueRangeMin = tonumber(num:sub(2, -2))
        elseif self.revResidueNumbering[residueRangeMin] then
          self.residueRangeMin = self.revResidueNumbering[residueRangeMin]
        else
          packageError("Invalid residue range: " .. residueRangeMin)
        end
      end

      if residueRangeMax == "auto" then
        self.residueRangeMax = self.sequenceLength
      else
        num = residueRangeMax:match("%b()")
        if num then
          self.residueRangeMax = tonumber(num:sub(2, -2))
        elseif self.revResidueNumbering[residueRangeMax] then
          self.residueRangeMax = self.revResidueNumbering[residueRangeMax]
        else
          packageError("Invalid residue range: " .. residueRangeMax)
        end
      end

      if self.residueRangeMin >= self.residueRangeMax then
        packageError("Residue range is smaller than 1.")
      end
    end,
```

The following fields map to functions already defined.

```
    defaultRulerStepSize = tonumber,
    name = tostring,
    sequence = tostring,
    xUnit = stringToDim,
    yUnit = stringToDim,
    residuesPerLine = tonumber,
    baselineSkip = tonumber,
```

`keyHash.rulerRange` sets the ruler range, treating possible errors and inconsistencies (for example, if the upper ruler range exceeds the upper residue range).

```
    rulerRange = function(v)
      local num
      local ranges = {}
      local rulerRangeMin, rulerRangeMax, rulerRangeStep
      for _, value in ipairs(v:explode(",+")) do
        rulerRangeMin, rulerRangeMax, rulerRangeStep =
          getRange(value:trim(), "^([%w%(%)]+)",
            "%-%s*([%w%(%)]+)", "step%s*(%d+)$")

        if rulerRangeMin == "auto" then
          rulerRangeMin = self.residueRangeMin
        else
          num = rulerRangeMin:match("%b()")
          if num then
            rulerRangeMin = tonumber(num:sub(2, -2))
          elseif self.revResidueNumbering[rulerRangeMin] then
            rulerRangeMin = self.revResidueNumbering[rulerRangeMin]
          else
            packageError("Invalid lower ruler range: " .. rulerRangeMin)
          end
        end

        if rulerRangeMax then
          if rulerRangeMax == "auto" then
            rulerRangeMax = self.residueRangeMax
          else
            num = rulerRangeMax:match("%b()")
            if num then
              rulerRangeMax = tonumber(num:sub(2, -2))
            elseif self.revResidueNumbering[rulerRangeMax] then
              rulerRangeMax = self.revResidueNumbering[rulerRangeMax]
            else
              packageError("Invalid upper ruler range: " .. rulerRangeMax)
            end
          end

```

```
        if rulerRangeMin >= rulerRangeMax then
          packageError("Ruler range is smaller than 1.")
        end
        if rulerRangeMin < self.residueRangeMin then
          rulerRangeMin = self.residueRangeMin
          packageWarning(
            "Lower ruler range is smaller than" ..
            "lower residue range. It was adjusted to " ..
            rulerRangeMin .. "."
          )
        end
        if rulerRangeMax > self.residueRangeMax then
          rulerRangeMax = self.residueRangeMax
          packageWarning(
            "Upper ruler range exceeds" ..
            "upper residue range. It was adjusted to " ..
            rulerRangeMax .. "."
          )
        end
      else
        rulerRangeMax = rulerRangeMin
      end
      rulerRangeStep = tonumber(rulerRangeStep)
        or self.defaultRulerStepSize

      for i = rulerRangeMin, rulerRangeMax, rulerRangeStep do
        table.insert(
          ranges,
          {pos = i, number = self.residueNumbering[i]}
        )
      end
    end
    return ranges
  end,
```

`keyHash.showRuler` determines if the ruler is visible.

```
  showRuler = function(v)
    if v == "true" then return true else return false end
  end
}
```

We iterate over all fields in the argument of `setParameters`. If a field of the same name exists in `keyHash`, we call this field with the value of the corresponding field in `newParms` as parameter.

```
for key, value in pairs(newParms) do
  if keyHash[key] then
    self[key] = keyHash[key](value)
```

```
      if pgfmolbio.errorCatched then return end
    end
  end
end

```

5.6.6 Adding Feature

`Protein:addFeature` converts raw feature information to the format of `ft` fields (described in section 5.6.3). Firstly, the method determines the index of the style that should be used for the current feature.

```
function Protein:addFeature(newFeature)
  local baseKey, ftEntry

  baseKey = self.specialKeys:getBaseKey(newFeature.key)
  if self.currentStyle[baseKey] then
    self.currentStyle[baseKey] = self.currentStyle[baseKey] + 1
  else
    self.currentStyle[baseKey] = 1
  end

```

Then, a new field for the feature table is set up.

```
  ftEntry = {
    key = newFeature.key,
    start = self:toAbsoluteResidueNumber(newFeature.start),
    stop = self:toAbsoluteResidueNumber(newFeature.stop),
    kvList = "style={" ..
      self.specialKeys:selectStyleFromList(baseKey,
        self.currentStyle[baseKey]) .. "}",
    level = newFeature.level or nil
  }
```

Finally, the key-value list `kvList` is modified (if applicable) and the new field is inserted into `ft`.

```
  if newFeature.kvList ~= "" then
    ftEntry.kvList = ftEntry.kvList .. "," .. newFeature.kvList
  end
  if newFeature.description then
    ftEntry.kvList = ftEntry.kvList ..
      ",description={" .. newFeature.description .. "}"
    ftEntry.description = newFeature.description
  end
  table.insert(self.ft, newFeature.layer or #self.ft + 1, ftEntry)
end
```

5.6.7 Calculate Disulfide Levels

`Protein:calculateDisulfideLevels` arranges disulfide-like features in non-overlapping levels.

```
function Protein:calculateDisulfideLevels()
  if pgfmolbio.errorCatched then return end
  local disulfideGrid, currLevel, levelFree
  disulfideGrid = {}

  for i, v in ipairs(self.ft) do
    if self.specialKeys.disulfideKeys[v.key] then
```

If the `level` field of a disulfide-like feature is already specified, it overrides the automatic mechanism of level determination. This may lead to clashes.

```
      if v.level then
        if not disulfideGrid[v.level] then
          disulfideGrid[v.level] = {}
        end
        for currPos = v.start, v.stop do
          disulfideGrid[v.level][currPos] = true
        end
```

Otherwise, the algorithm looks for the first free level (starting at level 1), i. e. the first level the feature may occupy without clashing with another one. (1) If the level currently checked already exists, it has been created by a previous disulfide-like feature. In this case, it is considered free if the previous feature does not overlap with the current one.

```
      else
        currLevel = 1
        repeat
          levelFree = true
          if disulfideGrid[currLevel] then
            for currPos = v.start, v.stop do
              levelFree = levelFree
                and not disulfideGrid[currLevel][currPos]
            end
            if levelFree then
              self.ft[i].level = currLevel
              for currPos = v.start, v.stop do
                disulfideGrid[currLevel][currPos] = true
              end
            end
```

(2) If the level currently checked does not exist, it must be free.

```
          else
            self.ft[i].level = currLevel
            disulfideGrid[currLevel] = {}
            for currPos = v.start, v.stop do
              disulfideGrid[currLevel][currPos] = true
            end
            levelFree = true
          end
          currLevel = currLevel + 1
        until levelFree == true
      end
    end
  end
end

```

5.6.8 Print Domains

`Protein:printTikzDomains` is the heart of the Lua script, since it converts a `Protein` object to TEX code.

```
function Protein:printTikzDomains()
  if pgfmolbio.errorCatched then return end
  local xLeft, xMid, xRight, yMid, xLeftClip, xRightClip,
    currLine, residuesLeft, currStyle

```

(1) Features (excluding `other/ruler` and `other/name`) For each feature in the feature table, we first calculate its coordinates (`xLeft`, `xMid`, `xRight` and `yMid`) and clipped areas (`xLeftClip`, `xRightClip`).

```
  for _, currFeature in ipairs(self.ft) do
    currLine = 0
    xLeft = currFeature.start - self.residueRangeMin -
      currLine * self.residuesPerLine + 1
    while xLeft > self.residuesPerLine do
      xLeft = xLeft - self.residuesPerLine
      currLine = currLine + 1
    end
    xLeft = xLeft - 1
    xRight = currFeature.stop - self.residueRangeMin -
      currLine * self.residuesPerLine + 1
    residuesLeft = self.residueRangeMax - self.residueRangeMin -
      currLine * self.residuesPerLine + 1
    xLeftClip = stringToDim("-5cm")
```

```
    xRightClip = self.residuesPerLine * self.xUnit

    if currFeature.start <= self.residueRangeMax
        and currFeature.stop >= self.residueRangeMin then
      repeat
        if residuesLeft <= self.residuesPerLine then
          if residuesLeft < xRight then
            xRightClip = residuesLeft * self.xUnit
          else
            xRightClip = xRight * self.xUnit + stringToDim("5cm")
          end
        else
          if xRight <= self.residuesPerLine then
            xRightClip = xRight * self.xUnit + stringToDim("5cm")
          end
        end
        if xLeft < 0 then xLeftClip = stringToDim("0cm") end

        xMid = (xLeft + xRight) / 2
        yMid = -currLine * self.baselineSkip
```

The current feature is extended by any level and sequence information present.

```
        if currFeature.level then
          currFeature.kvList = currFeature.kvList ..
            ",level=" .. currFeature.level
        end
        currFeature.sequence =
          self.sequence:sub(currFeature.start, currFeature.stop)

```

Each feature appears within its own `scope`. A `pgfinterruptboundingbox` ensures that the bounding box of the picture ignores the feature, since the `\clip` macro would enlarge it too much. Auxiliary macros for `\setfeatureshape` are defined (section 3.4).

```
        tex.sprint("\n\t\\begin{scope}\\begin{pgfinterruptboundingbox}")
        tex.sprint("\n\t\t\\def\\xLeft{" ..
          dimToString(xLeft * self.xUnit) .. "}")
        tex.sprint("\n\t\t\\def\\xMid{" ..
          dimToString(xMid * self.xUnit) .. "}")
        tex.sprint("\n\t\t\\def\\xRight{" ..
          dimToString(xRight * self.xUnit) .. "}")
        tex.sprint("\n\t\t\\def\\yMid{" ..
          dimToString(yMid * self.yUnit) .. "}")
        tex.sprint("\n\t\t\\def\\featureSequence{" ..
          currFeature.sequence .. "}")
        tex.sprint(
          "\n\t\t\\clip (" ..
```

```
          dimToString(xLeftClip) ..
          ", \\yMid + " ..
          dimToString(stringToDim("10cm")) ..
          ") rectangle (" ..
          dimToString(xRightClip) ..
          ", \\yMid - " ..
          dimToString(stringToDim("10cm")) ..
          ");"
        )
        tex.sprint(
          "\n\t\t\\pgfmolbioset[domains]{" ..
          currFeature.kvList ..
          "}"
        )
```

We invoke either the print function associated with the current feature or directly call `\pmbdomdrawfeature`. Afterwards, we close both surrounding environments.

```
        if self.specialKeys.printFunctions[currFeature.key] then
          self.specialKeys.printFunctions[currFeature.key](
            currFeature, xLeft, xRight, yMid, self.xUnit, self.yUnit)
        else
          tex.sprint("\n\t\t\\pmbdomdrawfeature{" ..
            currFeature.key .. "}")
        end
        tex.sprint("\n\t\\end{pgfinterruptboundingbox}\\end{scope}")

```

Calculate coordinates for the next line of the feature.

```
        currLine = currLine + 1
        xLeft = xLeft - self.residuesPerLine
        xRight = xRight - self.residuesPerLine
        residuesLeft = residuesLeft - self.residuesPerLine
      until xRight < 1 or residuesLeft < 1
    end
  end

```

(2) Feature `other/ruler` The ruler requires special treatment, buth the algorithm is actually simple: For each marker, calculate its coordinates, select its style and print it.

```
  if self.showRuler then
    currStyle = 1
    tex.sprint("\n\t\\begin{scope}")
    for _, currRuler in ipairs(self.rulerRange) do
```

```
      currLine = 0
      xMid = currRuler.pos - self.residueRangeMin -
        currLine * self.residuesPerLine + 1
      while xMid > self.residuesPerLine do
        xMid = xMid - self.residuesPerLine
        currLine = currLine + 1
      end
      xMid = xMid - 0.5
      yMid = -currLine * self.baselineSkip
      tex.sprint(
        "\n\t\t\\pgfmolbioset[domains]{current style/.style={" ..
        self.specialKeys:selectStyleFromList("other/ruler", currStyle) ..
        "}}"
      )
      tex.sprint("\n\t\t\t\\def\\xMid{" ..
        dimToString(xMid * self.xUnit) .. "}")
      tex.sprint("\n\t\t\t\\let\\xLeft\\xMid\\let\\xRight\\xMid")
      tex.sprint("\n\t\t\t\\def\\yMid{" ..
        dimToString(yMid * self.yUnit) .. "}")
      tex.sprint("\n\t\t\t\\def\\residueNumber{" ..
        currRuler.number .. "}")
      tex.sprint("\n\t\t\t\\pmbdomdrawfeature{other/ruler}")
      currStyle = currStyle + 1
    end
    tex.sprint("\n\t\\end{scope}")
  end

```

(3) Feature `other/name` Similarly, we calculate the coordinates of the name and print it.

```
  xMid =
    math.min(
      self.residuesPerLine,
      self.residueRangeMax - self.residueRangeMin + 1
    ) / 2
  tex.sprint("\n\t\\begin{scope}")
  tex.sprint(
    "\n\t\t\\pgfmolbioset[domains]{current style/.style={" ..
    self.specialKeys:selectStyleFromList("other/name", 1) ..
    "}}"
  )
  tex.sprint("\n\t\t\\def\\xLeft{0mm}")
  tex.sprint("\n\t\t\\def\\xMid{" .. dimToString(xMid * self.xUnit) .. "}")
  tex.sprint("\n\t\t\\def\\xRight{" ..
    dimToString(self.residuesPerLine * self.xUnit) .. "}")
  tex.sprint("\n\t\t\\def\\yMid{0mm}")
  tex.sprint("\n\t\t\\pmbdomdrawfeature{other/name}")
```

```
  tex.sprint("\n\t\\end{scope}")

```

(4) Set bounding box The bounding box is determined manually in order to prevent excessive enlargement due to clipping. The top left corner of the bounding box is the coordinate (`enlarge left`, `enlarge top`).

```
  tex.sprint(
    "\n\t\\pmbprotocolsizes{" ..
    "\\pmbdomvalueof{enlarge left}}{\\pmbdomvalueof{enlarge top}}"
  )
```

The x-coordinate of its right border is the largest line width plus the value of `enlarge right`. The y-coordinate of its bottom border is that of the lowermost line plus the value of `enlarge bottom`.

```
  currLine =
    math.ceil(
      (self.residueRangeMax - self.residueRangeMin + 1) /
        self.residuesPerLine
    ) - 1
  xRight =
    math.min(
      self.residuesPerLine,
      self.residueRangeMax - self.residueRangeMin + 1
    )
  tex.sprint(
    "\n\t\\pmbprotocolsizes{" ..
    dimToString(xRight * self.xUnit) ..
    " + \\pmbdomvalueof{enlarge right}}{" ..
    dimToString(-currLine * self.baselineSkip * self.yUnit) ..
    " + \\pmbdomvalueof{enlarge bottom}}"
  )
end

```

5.6.9 Converting a `Protein` to a String

`Protein:__tostring` is required by the convert module and returns a `pmbdomains` environment that contains all the information stored in the `Protein` object (section 4.3). Firstly, we start the environment.

```
function Protein:__tostring()
  local result = {}
  local currLine

```

```
  currLine = "\\begin{pmbdomains}\n\t\t[name={" ..
    self.name ..
    "}"
  if self.sequence ~= "" then
    currLine = currLine ..
      ",\n\t\tsequence=" ..
      self.sequence
  end
  currLine = currLine ..
    "]{" ..
    self.sequenceLength ..
    "}"
  table.insert(result, currLine)

```

Afterwards, each feature in the feature table is converted to an `\addfeature` macro. Note the use of the `includeDescription` field (described in section 5.6.3).

```
  for i, v in ipairs(self.ft) do
    if v.key ~= "other/main chain" then
      currLine = "\t\\addfeature"
      if self.includeDescription and v.description then
        currLine =
          currLine ..
          "[description={" ..
          v.description ..
          "}]"
      end
      currLine =
        currLine ..
        "{" ..
        v.key ..
        "}{" ..
        v.start ..
        "}{" ..
        v.stop ..
        "}"
      table.insert(result, currLine)
    end
  end
```

Finally, we close the environment.

```
  table.insert(result,
    "\\end{pmbdomains}"
  )
  return table.concat(result, "\n")
end
```

5.7 pgfmolbio.convert.tex

The code for the convert module is short: We only need to declare four options and set \pdfdraftmode to 1 in order to prevent pdfTEX from producing any pdf output.

```
\pdfdraftmode1

\pgfkeyssetvalue{/pgfmolbio/convert/output file name}{(auto)}
\pgfkeyssetvalue{/pgfmolbio/convert/output file extension}{tex}

\pgfmolbioset[convert]{%
  output code/.is choice,
  output code/tikz/.code=\pmb@con@outputtikzcodetrue,
  output code/pgfmolbio/.code=\pmb@con@outputtikzcodefalse,
  output code=tikz
}

\pgfmolbioset[convert]{%
  include description/.is if=pmb@con@includedescription,
  include description
}
```