GenomeDiagram: A User Guide v0.2

Leighton Pritchard
Scottish Crop Research Institute,
Invergowrie,
Dundee,
DD2 5DA,
UK

December 12, 2011

Contents

1 Introduction **2**

2 Installation **3**
2.1 Dependencies 3
2.2 Windows 3
2.3 Linux 3
2.4 Mac OS X 3

3 Quick Start: Creating a Simple Diagram **4**
3.1 Creating a FeatureSet 4
3.2 Creating a GraphSet 4
3.3 Creating a Track 5
3.4 Creating a Diagram 5
3.5 Drawing the image 5
3.6 Writing the image to file 5

4 Creating a Diagram - Long Version: Objects and Methods 6
4.1 GDDiagram . . . 6
4.1.1 Track methods of GDDiagram . . . 6
4.1.2 Diagram methods of GDDiagram . . . 7
4.2 GDTrack . . . 8
4.2.1 Set methods of GDTrack . . . 10
4.2.2 Other methods of GDTrack . . . 10
4.3 GDFeatureSet . . . 11
4.3.1 Feature methods of GDFeatureSet . . . 11
4.3.2 Other methods of GDFeatureSet . . . 11
4.4 GDGraphSet . . . 12
4.4.1 Feature methods of GDGraphSet . . . 12
4.4.2 Other methods of GDGraphSet . . . 12

5 Creating a Diagram - Long Version: Creating the Diagram 13
5.1 Two Ways to Build a Diagram . . . 13
5.1.1 Bottom-up . . . 13
5.1.2 Top-Down . . . 15
5.2 Drawing the diagram . . . 16

6 Help With Diagram Formatting 16
6.1 How Do I Make a Circular Diagram from a GenBank File? . . . 16
6.2 How Do I Make a Linear Diagram from a GenBank File? . . . 18
6.3 How Do I Change the Colours of Features? . . . 20
6.4 How Do I Change the Graph Format? . . . 22
6.5 How Do I Change the Graph Colours? . . . 22
6.6 How Do I Move Circular Tracks Out From the Centre of the Diagram? . . . 25
6.7 How Do I Change the Size of One Track Relative to Another? . . 29
6.8 How Do I Place One Graph on Top of Another? . . . 30

7 Acknowledgements 33

1 Introduction

GenomeDiagram is a Python module containing classes to aid the generation of publication-quality genome schematics in several vector and bitmap formats. The module can draw both linear and circular genome diagrams, focusing on a 'slice' of the full sequence if required. Information can be presented on a number of 'tracks' or 'levels' on the diagram; for example, the inner track of a circular diagram may be a scale, while outer tracks may describe a plot of GC skew and selected ORFs.

This user guide is an introductory account of how to use the Diagram class to generate schematics.

2 Installation

This section describes installation procedures for Windows, Mac (OS X) and Linux.

2.1 Dependencies

The GenomeDiagram module requires BioPython (http://www.biopython.org) and ReportLab (http://www.reportlab.com). Rendering of fonts in bitmaps may also require the installation of ˙renderPM (http://www.reportlab.com/rl˙addons) and Adobe Acrobat Reader and fonts (http://www.adobe.com).

2.2 Windows

The GenomeDiagram libraries are provided as a Windows installer. To install, double-click on the installer file `GenomeDiagram-n.n.win32.exe` (the numbers n.n depend on the version of the library you've downloaded). The installation wizard will walk you through the installation process.

2.3 Linux

The GenomeDiagram libraries are provided as a source distribution in the file `GenomeDiagram-n.n.tar.gz`. Copy this file to a temporary directory, and untar/unzip it:

```
tar -zxvf GenomeDiagram-0.1.tar.gz
```

This will unzip the source files into the directory GenomeDiagram-0.1. Change directory to this folder, and use

```
python setup.py install
```

to install the libraries.

2.4 Mac OS X

The Mac OS X installation works the same way as the Linux installation. Copy the GenomeDiagram-n.n.tar.gz file to a temporary directory and uncompress it with

```
tar -zxvf GenomeDiagram-0.1.tar.gz
```

in the shell. Change to the newly-created directory (`GenomeDiagram-0.1` here) and use

```
python setup.py install
```

to install the libraries.

If you have separate Python, Fink Python and MacPython installations, problems may arise. On my machine (G4 Powerbook, 10.2.6, MacPython 2.3), installing GenomeDiagram from the shell allowed me to use the package under Python in the shell and also from MacPython. Your mileage may vary.

3 Quick Start: Creating a Simple Diagram

The process of creating a diagram generally follows this simple pattern:

- create a FeatureSet for each separate set of features you want to display, and add Bio.SeqFeature objects to them
- create a GraphSet for each graph you want to display, and add graph data to them
- create a Track for each track you want on the diagram, and add GraphSets and FeatureSets to the tracks you require
- create a Diagram, and add the Tracks to it
- tell the Diagram to draw the image
- write the image to a file

The diagram may be altered - features, scales and plots added or removed, their colours, fonts and other display attributes changed - and multiple files may be written out from the same diagram. For the purposes of this document, however, only drawing single diagrams will be considered.

The following subsections describe the creation of a very simple graph, which doesn't take into account the many formatting options that are available. The ways in which more elaborate graphs may be constructed are described in more detail below.

3.1 Creating a FeatureSet

First import the **GenomeDiagram** package with

```
from GenomeDiagram import *
```

To create a handle to a new FeatureSet, use

```
gdfs = GDFeatureSet('name of the featureset')
```

The argument passed above is optional, to name the FeatureSet for ease of reference.

It doesn't matter how you obtain your set of Bio.SeqFeature objects - one way is via Biopython's Bio.GenBank.FeatureParser() - but once you have it you can add features one-by-one using the FeatureSet's add˙feature method

```
for feature in feature_list:  gdfs.add_feature(feature)
```

3.2 Creating a GraphSet

With the **GenomeDiagram** package imported, create a handle to a new GraphSet with

```
gdgs = GDGraphSet('name of the graphset')
```

As with the FeatureSet, the name argument is optional. To add a graph, use the GraphSet's new˙graph method

```
gdgs.new_graph(graphdata, name='Graph Name')
```

new˙graph expects a list of (position, value) tuples as its first argument, where position relates to the value's location on the sequence to be drawn. name is an optional string describing the graph, while style must be one of the graph format types. The argument passed to colour must be a reportlab.lib.colors.Color object.

3.3 Creating a Track

With the **GenomeDiagram** package imported, create a handle to a new Track with

```
gdt = GDTrack('name of track')
```

FeatureSets and GraphSets may be added to a track with the add˙set method.

```
gdt.add_set(gdgs)
gdt.add_set(gdfs)
```

3.4 Creating a Diagram

With the **GenomeDiagram** package imported, create a handle to a new Diagram with

```
gdd = GDDiagram('name of diagram')
```

Again the string argument is an optional descriptive name for the diagram

Tracks are added to the Diagram using the add˙track method

```
gdd.add_track(gdt, level)
```

where level is an integer denoting which level the track should be added at on the diagram.

3.5 Drawing the image

The features, graphs and other elements of the image are compiled using the draw method of the Diagram

```
gdd.draw()
```

Until it is explicitly written to file, the image is retained only in memory. Any changes made to the drawing settings however, must be checked in with the draw method

3.6 Writing the image to file

The compiled drawing is written to file with the passed filename using the Diagram's write method

```
gdd.write(filename, format)
```

The filename must be a valid file location, while the format should be one of the available image file format identifiers (see below).

4 Creating a Diagram - Long Version: Objects and Methods

In GenomeDiagram, the diagram structure is hierarchical: the Diagram itself contains, and is built from, Tracks which contain either FeatureSets or GraphSets. FeatureSets contain individual features, while GraphSets contain graph data. Tracks may be moved around the diagram independently of each other, and Sets may be moved between Tracks independently of each other.

4.1 GDDiagram

The keystone of **GenomeDiagram** is the GDDiagram object. This object provides the top-level interface for drawing the image, and also acts as the container for the data used to construct the diagram. This data is supplied in Tracks, each of which holds feature and/or graph information. Each track is displayed at a single level on the diagram; For a circular diagram, levels are numbered consecutively running outwards from the centre of the drawable area. For a linear diagram, they are numbered consecutively from the base of each sequence fragment.

GDDiagram has only one attribute, `name`, holding a short descriptive string describing the diagram as a whole. This string may be passed when instantiating an object.

GDDiagram's methods are divided into two types - those manipulating tracks, and those manipulating the diagram:

4.1.1 Track methods of GDDiagram

Specific tracks of the diagam may be retrieved from the diagram by subscripting the object with the level at which they appear, e.g. `gdd[3]` will return the track at level 3 of the GDDiagram instance `gdd`.

- new˙track(track˙level) Creates a new GDTrack object at the passed *track˙level* on the diagram, and returns it to the user for the addition of FeatureSets and/or GraphSets. If the passed level is already occupied, outer tracks are shunted up a level.

- add˙track(track, track˙level) Adds an existing GDTrack object (*track*to the diagram, to be displayed at the level indicated by the integer *track˙level*. If the passed *track˙level* is already occupied, the outer tracks of the diagram are pushed outwards by a level.

- move˙track(from˙level, to˙level) Moves an existing GDTrack object from one level (*from˙level*) to another (*to˙level*) on the same diagram.

- get˙tracks() Returns a list of the GDTrack objects contained in the diagram.

- get˙levels() Returns a list of the levels in the diagram that are already occupied by GDTrack objects.
- get˙drawn˙levels() Returns a list of the levels in the diagram that are occupied by GDTrack objects, and will be shown (have their hide attribute set to 0).
- del˙track(track˙level) Removes the GDTrack object found at the level indicated by *track˙level* from the GDDiagram object.
- renumber˙tracks(low=1) Renumbers all GDTrack objects in the GDDiagram object consecutively from a passed integer, *low* (default=1).
- range() Returns, as an integer tuple, the highest and lowest base positions indicated in any of the GDTrack child objects contained within the GDDiagram object.

4.1.2 Diagram methods of GDDiagram

- draw(format='circular', pagesize='A3', orientation='landscape', x=0.05, y=0.05, xl=None, xr=None, yt=None, yb=None, start=None, end=None, tracklines=0, fragments=10, fragment˙size=0.9, track˙size=0.75, circular=1) Constructs the diagram to be written out. Numerous formatting options are available here:
  - *format* is either 'circular' or 'linear', for the choice of circular or linear diagram (default='circular').
  - *pagesize* should be either a tuple of floats as (page width, page height) in pixels, or a string denoting a recognised ISO page size, such as 'A4', 'LEGAL', 'LETTER', etc. (default='A3').
  - *orientation* is either 'landscape' or 'portrait' and refers to the orientation of the page on which the diagram sits, not the format of the image itself (default='landscape').
  - *x* is a float indicating the size of the X (vertical) margins as a proportion of the whole page (default=0.05).
  - *y* is a float indicating the size of the Y (horizontal) margins as a proportion of the whole page (default=0.05)
  - *xl* is a float indicating the size of the left X margin, as a proportion of the whole page. If specified, this overrides the parameter *x*.
  - *xr* is a float indicating the size of the right X margin, as a proportion of the whole page. If specified, this overrides the parameter *x*.
  - *yt* is a float indicating the size of the top Y margin, as a proportion of the whole page. If specified, this overrides the parameter *y*.
  - *yb* is a float indicating the size of the bottom Y margin, as a proportion of the whole page. If specified, this overrides the parameter *y*.

- *start* is an int indicating the base position from which to begin drawing the diagram.
- *end* is in int indicating the base position at which to stop drawing the diagram.
- *track˙size* is a float specifying what proportion of the vertical space available to each track should be taken up by the drawing of the track (default=0.75).
- *tracklines* is a Boolean indicating whether a set of lines delineating each track should be superimposed on the diagram (default=0).
- *fragments* is an integer specifying how many sections the sequence should be divided into on a linear diagram. This is necessary for clarity, to avoid unreadable compression of diagram information in the X direction (default=10).
- *fragment˙size* is a float specifying what proportion of the vertical space available to each fragment should be taken up by the drawing of the fragmwriteent (default=0.9).
- *circular* is a Boolean value describing whether the sequence to be drawn is circular or not - this is only directly relevant to circular diagrams.

The parameters listed above are also attributes of the object which may be accessed directly.

Once drawn, the diagram remains 'virtual' in memory until written out to file. Once the draw() method is called, if the diagram is modified, the draw() method must be called again before the changes are applied.

- write(filename='test1.ps', output='PS') Writes the diagram out as an image to the filename passed as *filename*, in the format specified by *output*. Both raster (BMP, JPG, PNG etc.) and vector (EPS, PDF) formats are supported via **ReportLab** and **RasterPM**.

4.2 GDTrack

The GDTrack object is the largest scale of granularity for the diagram. It contains sets of features and/or graphs, and general formatting information for the track as a whole. Tracks may also incorporate a 'greytrack', which comprises a shaded track background and a superimposed label (the *name* attribute) foreground, and/or a scale. The track scale comprises a line running through the centre of the track, and ticks that run perpendicular to this. There are two types of tick, nominally long and short, and each may be labelled and manipulated separately. GDTrack provides methods for manipulating graph and feature sets, while track attributes may be accessed directly. These attributes may also be set on instantiation, and are listed below:

- *name* is a short descriptive string

- *height* is a float denoting the height of the track, relative to other tracks on the diagram (default=1).
- *hide* is a Boolean specifying whether the track should be drawn or not (default=0). Only tracks with hide=0 are listed by the GDDiagram method *get˙drawn˙levels()*.
- *greytrack* is a Boolean specifying whether the track should include a grey background (useful for delineating many closely-spaced tracks), and a set of foreground labels (default=0).
- *greytrack˙labels* is an integer specifying the number of foreground labels that should be included on the track (default=5).
- *greytrack˙fontsize* is an integer specifying the size of font to be used on the foreground labels (default=8).
- *greytrack˙font* is a string specifying the name of the font to be used for the foreground labels. Not all machines will provide the same font selection (default='Helvetica').
- *greytrack˙font˙rotation* is an integer specifying the angle in degrees through which to rotate the foreground labels, which are, by default, radial in circular diagrams and collinear with the track for linear diagrams (deafult=0).
- *greytrack˙font˙colour* is a ReportLab colors.Color object defining the colour of the foreground labels (default=colors.Color(0.6,0.6,0.6)).
- *scale* is a Boolean defining whether the track will carry a scale (default=1).
- *scale˙colour* is a ReportLab colors.Color object defining the colour of the scale (default=colors.black).
- *scale˙font* is a string specifying the font to use on the scale; not all machines provide the same selection (default='Helvetica').
- *scale˙fontsize* is an integer specifying the size of font to use for the scale (default=6).
- *scale˙fontangle* is an integer specifying the angle, in degrees, through which to rotate the scale labels relative, on circular diagrams, to the tangent to the scale at that point or, on linear diagrams, to the scale itself (default=45).
- *scale˙ticks* is a Boolean denoting whether any ticks will be shown on the scale (default=1).
- *scale˙largeticks* is a float describing the height of the large tick set as a proportion of half the track height. Positive values run 'upwards' (linear) or away from the centre of the diagram (circular), while negative values run in the opposite direction (default=0.5).

- *scale˙smallticks* is a float describing the height of the small tick set as a proportion of half the track height. Positive values run 'upwards' (linear) or away from the centre of the diagram (circular), while negative values run in the opposite direction (default=0.3).
- *scale˙largetick˙interval* is an integer specifying the interval between large ticks as a number of bases (default=1000000).
- *scale˙smalltick˙interval* is an integer specifying the interval between smallticks as a number of bases (default=10000).
- *scale˙largetick˙labels* is a Boolean describing whether labels marking tick position will be placed over every large tick (default=1).
- *scale˙smalltick˙labels* is a Boolean describing whether labels marking tick position will be places over every small tick (default=0).

The attributes described above are gross attributes of the track presentation as a whole. The track's additional role is to contain sets of features and graphs, which are manipulated by the following methods:

4.2.1 Set methods of GDTrack

Individual sets contained within a GDTrack object may be retrieved by subscripting with their unique ID, e.g. `gdt[6]` would return the set with unique ID 6.

- new˙set(type='feature') Creates a new GDFeatureSet (type='feature') or GDGraphSet (type='graph') object, adds it to the track, then returns it to the user so that features may be added.
- add˙set(set) Adds a preexisting GDFeatureSet or GDGraphSet, passed as *set* to the track.
- get˙sets() Returns a list of the graph and feature sets contained in the track.
- get˙ids() Returns a list of the unique IDs for all sets in the track.
- del˙set(id) Removes the set with the passed unique ID from the track.
- range() Returns a tuple of the lowest and highest bases represented by features and/or graphs on the track.

4.2.2 Other methods of GDTrack

There is one further method provided by GDTrack, which returns an account of the contents of the track.

- to˙string(verbose=0) Returns a formatted string containing an account of the GDTrack object's contents. *verbose* is a Boolean specifying whether the long or short form of this string is returned.

4.3 GDFeatureSet

The GDFeatureSet object is a container for GDFeature objects, and provides methods for manipulating them. Formatting information is not held at the GDFeatureSet level, but at the GDFeature level. GDFeatureSet has only two attributes, its unique ID, *id* for reference via GDTrack objects, and *name*, containing a short descriptive string. Both attributes may be set at instantiation.

4.3.1 Feature methods of GDFeatureSet

As with GDDiagram and GDTrack objects, features contained within GDFeatureSets may be retrieved by subscripting the feature set object with the unique ID of the feature, e.g. `gdfs[954]` returns the feature with unique ID 954. The number of features in a feature set may be found by using the *len* operator, i.e. `len(gdfs)`.

- add˙feature(feature, colour=colors.lightgreen) This method adds a Bio.SeqFeature object (from the BioPython package) to the feature set as the *feature* argument, with an optional rendering ReportLab colors.Color object *colour* argument (default=colors.lightgreen). The *add˙feature* method will process a Bio.SeqFeature 'colour' qualifier containing an integer corresponding to a member of the Artemis colour scheme.
- get˙features() Returns a list of GDFeature objects contained in the feature set.
- get˙ids() Returns a list of unique identifiers for the features contained in the set.
- set˙all˙features(attr, value) For all features in the feature set, assigns the passed *value* to the attribute *attr* (passed as a string of the attribute's name).
- del˙feature(id) Deletes the feature with the passed unique ID from the feature set.
- range() Returns a tuple of the highest and lowest bases covered by the features in the feature set.

4.3.2 Other methods of GDFeatureSet

The GDFeatureSet object also provides:

- to˙string(verbose=0) Returns a formatted description of the contents of the feature set in either long (verbose=1) or short(default, verbose=0) form.

4.4 GDGraphSet

The GDGraphSet object is a container for GDGraphData objects, and provides methods for manipulating them. Formatting information is not held at the GDGraphSet level, but at the GDGraphData level. GDGraphSet, like GDFeatureSet, with which it shares many properties, has only two attributes, its unique ID, *id* for reference via GDTrack objects, and *name*, containing a short descriptive string. Both attributes may be set at instantiation.

4.4.1 Feature methods of GDGraphSet

GDGraphSet objects contain GDGraphData objects, which in turn contain a series of values for each position on the sequence. As with GDFeatureSet objects, these values may be retrieved by subscripting the GDGraphSet object with the position, e.g. `gdgs[397]` returns the data value at position 397. The number of datapoints in a graph set may be found by using the *len* operator, i.e. `len(gdgs)`.

- new˙graph(data, name, style='bar', colour=colors.lightgreen, altcolour=colors.darkseagreen) This method adds a new dataset to the GDGraphSet object. *data* should be a list of (position, value) tuples, while *name* should be a short descriptive string. The graph drawing format is specified by the *style* argument; this may be one of the strings 'bar', 'heat' or 'line'. The *colour* argument specifies the colour of the line in the 'line' format graph, and the colour of 'high' values (greater than the median value) in 'bar' and 'heat' graphs. The *altcolour* argument denotes the colour to be used for values below the median in 'bar' and 'heat' graphs. Both colour arguments expect a colors.Color object.
- get˙graphs() Returns a list of GDGraphData objects contained in the graph set.
- get˙ids() Returns a list of unique identifiers for the graphs contained in the set.
- del˙graph(id) Deletes the graph with the passed unique ID from the feature set.
- range() Returns a tuple of the highest and lowest bases covered by the graphs in the feature set.

4.4.2 Other methods of GDGraphSet

The GDGraphSet object also provides:

- to˙string(verbose=0) Returns a formatted description of the contents of the graph set in either long (verbose=1) or short(default, verbose=0) form.

5 Creating a Diagram - Long Version: Creating the Diagram

5.1 Two Ways to Build a Diagram

There are two approaches to constructing a sequence diagram using the Genome-Diagram library. One (outlined in section 3) involves constructing the diagram from the bottom up, first filling a GDFeatureSet with features and adding it to a GDTrack, and so on. The second is a top-down approach, first creating a diagram, and successively obtaining new tracks and feature- and graph sets. Both methods are described below.

5.1.1 Bottom-up

For this approach, we obtain our features, then bundle them into one or more feature sets; graphs are also bundled into one or more graph sets. These sets are then apportioned amongst GDTrack objects, which are then added to the GDDiagram itself.

- Import modules

  ```
  from Bio import GenBank
  from GenomeDiagram import GDDiagram, GDFeatureSet, GDGraphSet, GDTrack, GDUtilities
  ```

  For this example we need the Bio.GenBank module only to obtain a set of Bio.SeqFeature objects. Any way you choose to obtain Bio.SeqFeature objects will do. The GD* classes are the building blocks for making the diagram, and are necessary for the bottom-up method of constructing the diagram.

- Obtain a set of Bio.SeqFeature objects

  ```
  parser = GenBank.FeatureParser()
  fhandle = open('NC_005213.gbk', 'r')
  genbank_entry = parser.parse(fhandle)
  ```

  For this example we use the GenBank.FeatureParser object to parse the [i]Nanoarchaeum equitans[/i] sequence contained in the GenBank file NC˙005213.gbk. This method returns an object containing multiple Bio.SeqFeature objects, which may be looped over, as seen below.

- Assign features to a feature set

  ```
  gdfs = GDFeatureSet(name='CDS features')
       for feature in genbank_entry.features:
  ```

```
if feature.type == 'CDS':
    gdfs.add_feature(feature)
```

Here we first create our feature set, assigning it to the variable `gdfs`, and giving it the name 'CDS features'. Then we loop over the set of features in the Genbank Entry, and add each one to the feature set.

- Assign graph data to a graph set

  ```
  gdgs = GDGraphSet('GC Content')
  graphdata = GDUtilities.gc_content(genbank_entry.seq, 100)
  gdgs.new_graph(graphdata, 'GC content', style='line')
  ```

  Here we create our graph set, assigning it to the variable `gdgs` and ascribing it the name 'GC Content'. We obtain our graph data by using the GDUtilities function `gc_content` with 100 base windows, though any method that returns a list of (position, value) tuples will do. Lastly, we add this graph data to the graph set using the GDGraphSet method `new_graph`, with the name 'GC Content', and the style set to 'line', for a line graph.

- Creating tracks and adding sets to them

  ```
  gdt1 = GDTrack('CDS features', greytrack=1)
  gdt2 = GDTrack('GC Content', greytrack=1)
  gdt1.add_set(gdfs)
  gdt2.add_set(gdgs)
  ```

  Two tracks are created, and assigned to the variables gdt1 and gdt2, respectively labelled 'CDS features' and 'GC content'. The `greytrack` parameter adds a grey background and a foreground label to each track. The `add_set` method is used to add the feature set `gdfs` to `gdt1` and the graph set `gdgs` to `gdt2`.

- Adding tracks to the diagram

  ```
  gdd = GDDiagram('NC_005213.gbk')
  gdd.add_track(gdt1, 2)
  gdd.add_track(gdt2, 4)
  ```

  The GDDiagram object is assigned to the variable gdd, and given the name 'NC˙005213.gbk'. Its `add_track` method is then used to add tracks `gdt1` and `gdt2` to track numbers 2 and 4 respectively. The diagram is now ready for drawing

5.1.2 Top-Down

- Import modules

  ```
  from Bio import GenBank
  from GenomeDiagram import GDDiagram, GDUtilities.gc_content
  ```

  When working top-down, the only required GenomeDiagram import is of `GDDiagram`. The Bio.GenBank and GDUtilities imports are here only to allow the example to work.

- Creating the diagram

  ```
  gdd = GDDiagram('NC_005213.gbk')
  ```

  The diagram is created first, and given the name 'NC˙005213.gbk'.

- Adding tracks to the diagram

  ```
  gdt1 = gdd.new_track(2, greytrack=1, name='CDS features')
  gdt2 = gdd.new_track(4, greytrack=1, name='GC content')
  ```

  Tracks are obtained by calling the `new_track` method of the GDDiagram object. Parameters for the track can be specified following the level at which the track is to be added, but must be specified by keyword.

- Adding graph and feature sets to the diagram

  ```
  gdfs = gdt1.new_set('feature')
  gdgs = gdt2.new_set('graph')
  ```

  Feature and graph sets are obtained by calling the `new_set` method of the GDTrack object. The type of set required is passed as the first parameter, and parameters for each set can be specified following the type, but must be specified by keyword.

- Adding features and graphs to the diagram

  ```
  parser = GenBank.FeatureParser()
  fhandle = open('/data/genomes/Bacteria/Nanoarchaeum_equitans/NC_005213.gbk', 'r')
  genbank_entry = parser.parse(fhandle)

  graphdata = GDUtilities.gc_content(genbank_entry.seq, 100)
  gdgs.new_graph(graphdata, 'GC content', style='line')
  ```

```
for feature in genbank_entry.features:
    if feature.type == 'CDS':
        gdfs.add_feature(feature)
```

We again use the Bio.GenBank.FeatureParser to read in a GenBank format file and obtain a set of Bio.SeqFeature objects, and the GDUtilities.gc˙content function to generate the graph data. Neither of these is the only method of obtaining this information, and any way obtaining a set of Bio.SeqFeature objects or a list of (position, value) tuples would do.

The method of adding feature and graph data to the feature and graph sets is identical to the bottom-up method. Adding features is achieved by looping over the contents of the genbank˙entry object, applying the GDFeatureSet's `add_feature` method. Adding graph data is done using the GDGraphSet's `new_graph` method. The diagram is again ready for drawing.

5.2 Drawing the diagram

Though there is more than one way to build a diagram, there is only one way to draw it in the GenomeDiagram library. The GDDiagram's `draw` method is used to make up the drawing, and the `write` method to write the resulting image to a file.

```
gdd.draw(format='linear', orientation='landscape',
         tracklines=0, pagesize='A5', fragments=5, circular=0)
gdd.write('NC_005213.ps', 'PS')
```

The resulting image is identical, whichever build method is used, and can be seen in figure 1.

6 Help With Diagram Formatting

This section contains some advice on how to use the GenomeDiagram package to achieve specific formatting effects. It is far from exhaustive, but contains examples and working code that may be useful in exploring further. Examples are presented in a 'How Do I...?' format:

6.1 How Do I Make a Circular Diagram from a GenBank File?

To build a basic circular diagram, first you need to obtain features from the GenBank file - this can be done using BioPython:

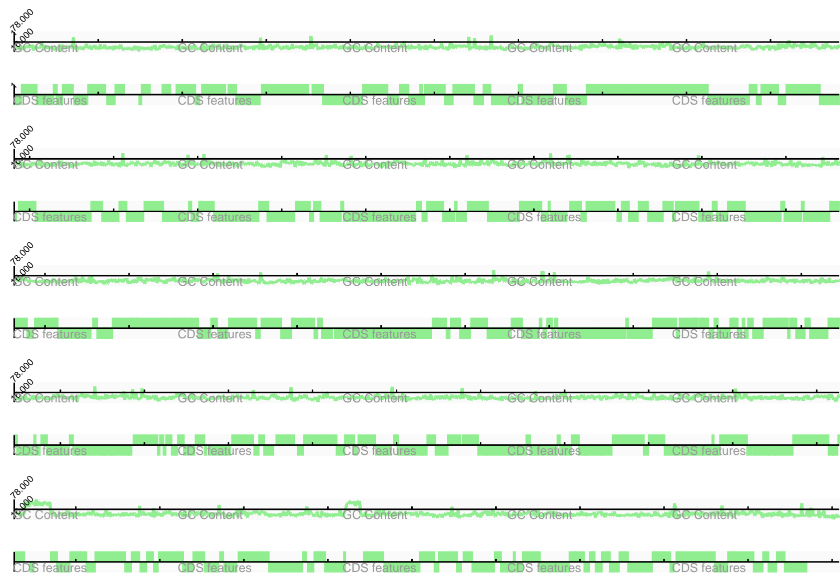

Figure 1: Output from section 5.2. The same result is achieved whether a top-down or bottom-up approach to constructing the diagram is used

```
figure from Bio import GenBank
parser = GenBank.FeatureParser()
fhandle = open('/data/genomes/Bacteria/Nanoarchaeum_equitans/NC_005213.gbk', 'r')
genbank_entry = parser.parse(fhandle)
fhandle.close()
```

Then you can build a diagram as described above (section 5):

```
from GenomeDiagram import GDDiagram, GDUtilities
gdd = GDDiagram('NC_005213.gbk')
gdt1 = gdd.new_track(1, greytrack=1, name='CDS features')
gdt2 = gdd.new_track(2, greytrack=1, name='GC content')
gdfs = gdt1.new_set('feature')
gdgs = gdt2.new_set('graph')
graphdata = GDUtilities.gc_content(genbank_entry.seq, 100)
gdgs.new_graph(graphdata, 'GC content', style='line')
for feature in genbank_entry.features:
    if feature.type == 'CDS':
        gdfs.add_feature(feature)
gdd.draw(format='circular', orientation='landscape',
         tracklines=0, pagesize='A5', fragments=5, circular=1)
gdd.write('NC_005213.ps', 'PS')
```

6.2 How Do I Make a Linear Diagram from a GenBank File?

To construct a basic linear diagram, follow the template in section 6.1, passing the value 'linear' as the argument to *format* in gdd.draw:

```
from Bio import GenBank
parser = GenBank.FeatureParser()
fhandle = open('/data/genomes/Bacteria/Nanoarchaeum_equitans/NC_005213.gbk', 'r')
genbank_entry = parser.parse(fhandle)
fhandle.close()
from GenomeDiagram import GDDiagram, GDUtilities
gdd = GDDiagram('NC_005213.gbk')
gdt1 = gdd.new_track(1, greytrack=1, name='CDS features')
gdt2 = gdd.new_track(2, greytrack=1, name='GC content')
gdfs = gdt1.new_set('feature')
gdgs = gdt2.new_set('graph')
graphdata = GDUtilities.gc_content(genbank_entry.seq, 100)
gdgs.new_graph(graphdata, 'GC content', style='line')
for feature in genbank_entry.features:
    if feature.type == 'CDS':
```

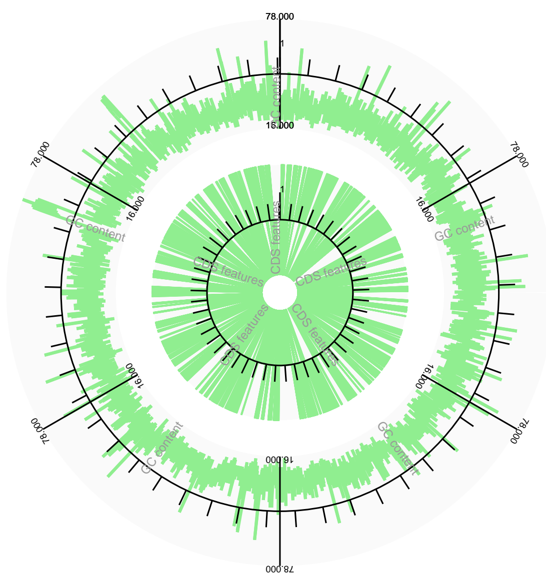

Figure 2: Output from section 6.1, a basic circular diagram.

```
        gdfs.add_feature(feature)
gdd.draw(format='linear', orientation='landscape',
         tracklines=0, pagesize='A5', fragments=5, circular=1)
gdd.write('NC_005213.ps', 'PS')
```

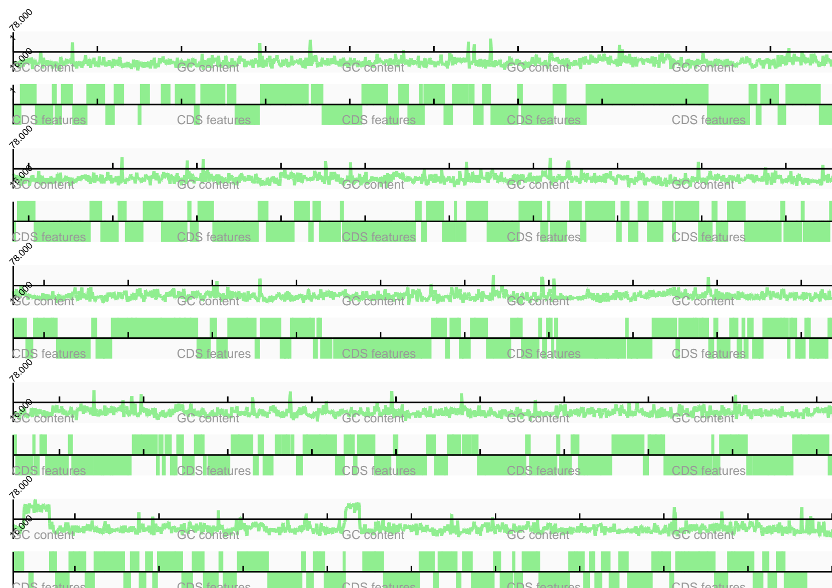

Figure 3: Output from section 6.2, a basic linear diagram.

6.3 How Do I Change the Colours of Features?

There is more than one way to do this. You could explicitly pass a colour as an argument to the `add_feature` method: `gdfs.add_feature(feature, colour=colors.red)` or you could colour all features simulataneously with the `set_all_features` method: `gdfs.set_all_features('colour', colors.red)` . Alternatively, you could examine individual features, retrieving them as subscripts with, e.g. `feature = gdfs[35]` and changing the attribute directly with the `set_colour` method: `feature.set_colour(colors.red)`. An example is given below:

```
from Bio import GenBank
from reportlab.lib import colors
parser = GenBank.FeatureParser()
```

```
fhandle = open('/data/genomes/Bacteria/Nanoarchaeum_equitans/NC_005213.gbk', 'r')
genbank_entry = parser.parse(fhandle)
fhandle.close()
from GenomeDiagram import GDDiagram, GDUtilities
gdd = GDDiagram('NC_005213.gbk')
gdt1 = gdd.new_track(1, greytrack=1, name='CDS features')
gdt2 = gdd.new_track(2, greytrack=1, name='GC content')
gdfs = gdt1.new_set('feature')
gdgs = gdt2.new_set('graph')
graphdata = GDUtilities.gc_content(genbank_entry.seq, 100)
gdgs.new_graph(graphdata, 'GC content', style='line')
for feature in genbank_entry.features:
    if feature.type == 'CDS':
        gdfs.add_feature(feature, colour=colors.red)
gdd.draw(format='linear', orientation='landscape',
         tracklines=0, pagesize='A5', fragments=5, circular=1)
gdd.write('NC_005213.ps', 'PS')
```

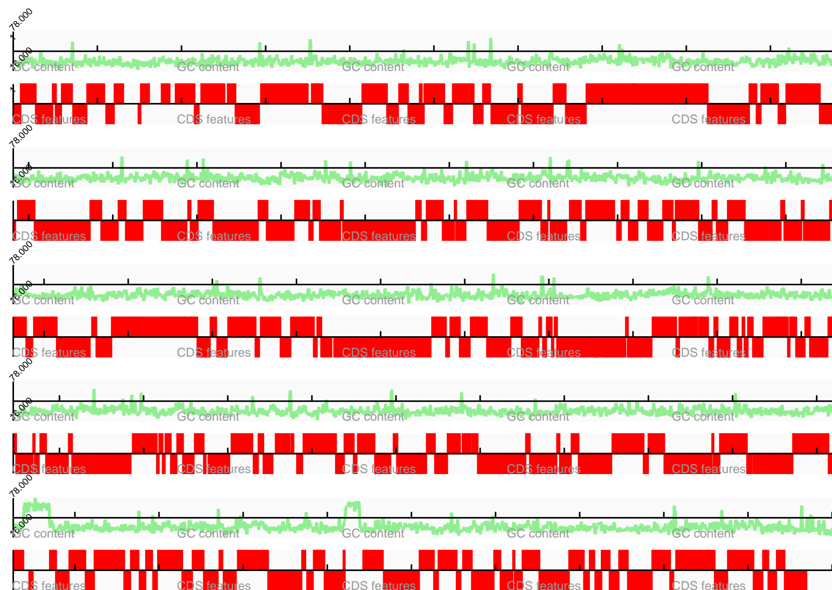

Figure 4: Output from section 6.3, all features have been coloured red.

6.4 How Do I Change the Graph Format?

The graph format may be changed between a simple line graph, a bar graph, and a 'heat' graph. This may done by changing the argument to `new_graph`, e.g. `gdgs.new_graph(graphdata, 'Graph name', style='bar')`, or by directly changing the style attribute of a graph object, once created. The easiest way to do this is by retaining a handle to the GDGraph object created by `new_graph`, e.g. `graph = gdgs.new_graph(graphdata, 'Name', 'heat')` and changing the *style* attribute direclty with `graph.style='bar'`. For line graphs, the thickness of the line may be changed by setting the graph's *linewidth* attribute, e.g. `graph.linewidth=0.5`. Example code is given below:

```
from Bio import GenBank
from reportlab.lib import colors
parser = GenBank.FeatureParser()
fhandle = open('/data/genomes/Bacteria/Nanoarchaeum_equitans/NC_005213.gbk', 'r')
genbank_entry = parser.parse(fhandle)
fhandle.close()
from GenomeDiagram import GDDiagram, GDUtilities
gdd = GDDiagram('NC_005213.gbk')
gdt1 = gdd.new_track(1, greytrack=1, name='CDS features')
gdt2 = gdd.new_track(2, greytrack=1, name='GC content')
gdfs = gdt1.new_set('feature')
gdgs = gdt2.new_set('graph')
graphdata = GDUtilities.gc_content(genbank_entry.seq, 100)
graph = gdgs.new_graph(graphdata, 'GC content', style='line')
graph.style='bar'
for feature in genbank_entry.features:
    if feature.type == 'CDS':
        gdfs.add_feature(feature, colour=colors.red)
gdd.draw(format='circular', orientation='landscape',
         tracklines=0, pagesize='A5', fragments=5, circular=1)
gdd.write('NC_005213.ps', 'PS')
```

6.5 How Do I Change the Graph Colours?

The way graph colours work depends on the graph format. For line graphs, only a single colour is required - that of the line to be drawn. For heat and bar graphs, two colours are needed - one for values above the midpoint, and one for values beolow the midpoint of the data. The easiest way to specify these is when adding the new graph, e.g. `graph = gdgs.new_graph(graphdata, 'Graph name', style='bar', colour=colors.violet, altcolour=colors.purple)`. Here, *colour* is the colour of the line, or of high data values, while *altcolour* is the colour of low data values. Example code is given below:

```
from Bio import GenBank
```

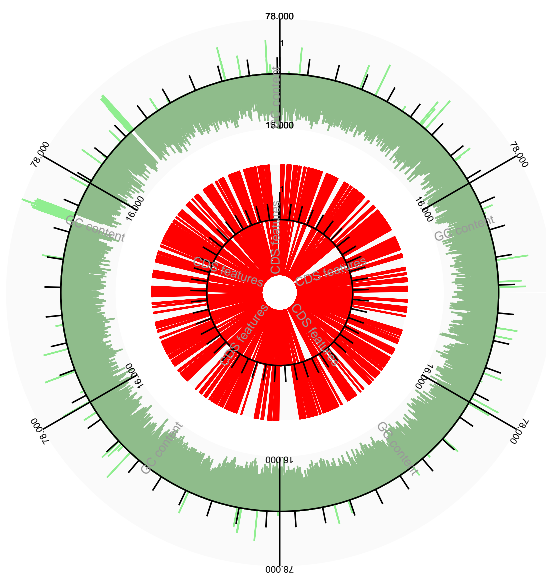

Figure 5: Output from section 6.4, graph in 'bar' style.

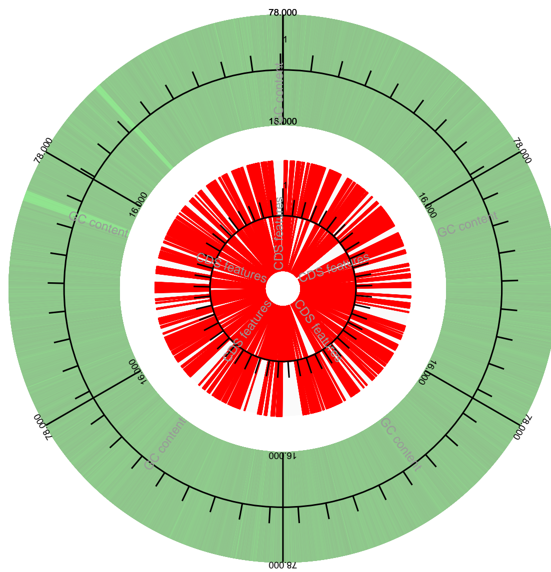

Figure 6: Output from section 6.4, graph in 'heat' style.

```
from reportlab.lib import colors
parser = GenBank.FeatureParser()
fhandle = open('/data/genomes/Bacteria/Nanoarchaeum_equitans/NC_005213.gbk', 'r')
genbank_entry = parser.parse(fhandle)
fhandle.close()
from GenomeDiagram import GDDiagram, GDUtilities
gdd = GDDiagram('NC_005213.gbk')
gdt1 = gdd.new_track(1, greytrack=1, name='CDS features')
gdt2 = gdd.new_track(2, greytrack=1, name='GC content')
gdfs = gdt1.new_set('feature')
gdgs = gdt2.new_set('graph')
graphdata = GDUtilities.gc_content(genbank_entry.seq, 100)
graph = gdgs.new_graph(graphdata, 'GC content', style='bar',
                       colour=colors.violet, altcolour=colors.purple)
for feature in genbank_entry.features:
    if feature.type == 'CDS':
        gdfs.add_feature(feature, colour=colors.red)
gdd.draw(format='circular', orientation='landscape',
         tracklines=0, pagesize='A5', fragments=5, circular=1)
gdd.write('NC_005213.ps', 'PS')
```

6.6 How Do I Move Circular Tracks Out From the Centre of the Diagram?

You can use the `renumber_tracks` method of a GDDiagram object to move tracks outwards from the centre of the circular diagram, e.g. `gdd.renumber_tracks(5)`. Alternatively, you can create the tracks at outer levels to begin with. Example code is given below, moving the first track out to level four on the diagram:

```
from Bio import GenBank
from reportlab.lib import colors
parser = GenBank.FeatureParser()
fhandle = open('/data/genomes/Bacteria/Nanoarchaeum_equitans/NC_005213.gbk', 'r')
genbank_entry = parser.parse(fhandle)
fhandle.close()
from GenomeDiagram import GDDiagram, GDUtilities
gdd = GDDiagram('NC_005213.gbk')
gdt1 = gdd.new_track(1, greytrack=1, name='CDS features')
gdt2 = gdd.new_track(2, greytrack=1, name='GC content')
gdfs = gdt1.new_set('feature')
gdgs = gdt2.new_set('graph')
graphdata = GDUtilities.gc_content(genbank_entry.seq, 100)
graph = gdgs.new_graph(graphdata, 'GC content', style='bar',
                       colour=colors.violet, altcolour=colors.purple)
for feature in genbank_entry.features:
```

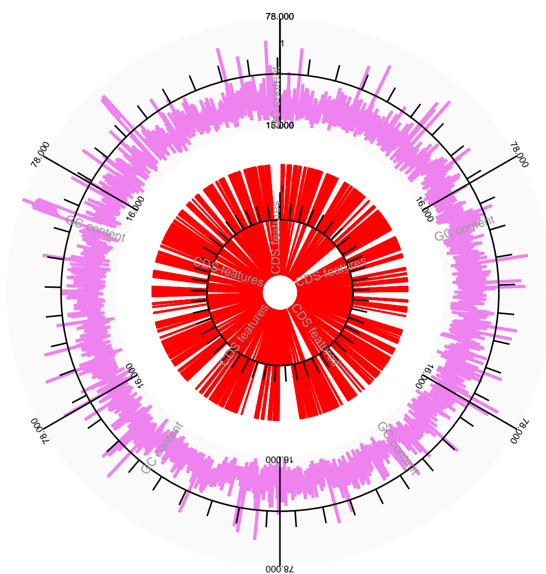

Figure 7: Output from section 6.5, graph in 'line' style.

Figure 8: Output from section 6.5, graph in 'bar' style.

Figure 9: Output from section 6.5, graph in 'heat' style.

```
    if feature.type == 'CDS':
        gdfs.add_feature(feature, colour=colors.red)
gdd.renumber_tracks(4)
gdd.draw(format='circular', orientation='landscape',
         tracklines=0, pagesize='A5', fragments=5, circular=1)
gdd.write('NC_005213.ps', 'PS')
```

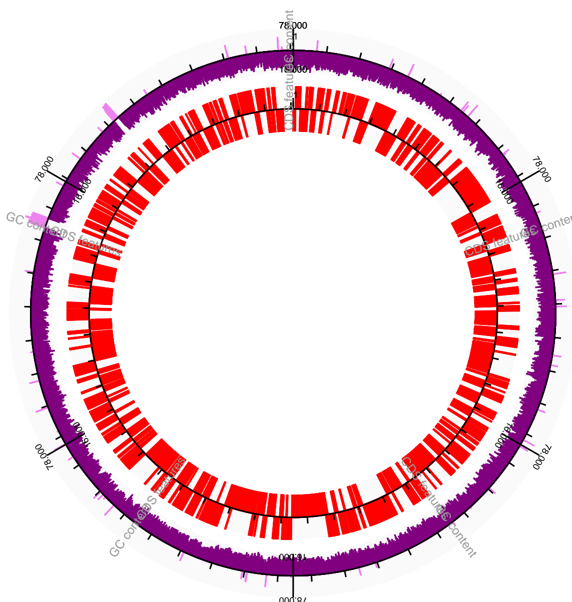

Figure 10: Output from section 6.6, with tracks now beginning at level four.

6.7 How Do I Change the Size of One Track Relative to Another?

Each track has a *height* attribute. On drawing, the height attributes of all tracks are summed to give a total - the relative height of each track is then its *height* divided by the sum of all track heights. Here, as all tracks start off with a default *height* of 1, we double the *height* of the graph track when we create it, using `gdt2 = gdd.new_track(2, greytrack=1, name='GC content', height=2)`. The same effect could have been achieved by changing the *height* attribute directly, e.g. `gdt2.height=2`, once the handle to the GDTrack object had been obtained. Example code is given below, making the graph track twice the height of the feature track.

```
from Bio import GenBank
from reportlab.lib import colors
parser = GenBank.FeatureParser()
fhandle = open('/data/genomes/Bacteria/Nanoarchaeum_equitans/NC_005213.gbk', 'r')
genbank_entry = parser.parse(fhandle)
fhandle.close()
from GenomeDiagram import GDDiagram, GDUtilities
gdd = GDDiagram('NC_005213.gbk')
gdt1 = gdd.new_track(1, greytrack=1, name='CDS features')
gdt2 = gdd.new_track(2, greytrack=1, name='GC content', height=2)
gdfs = gdt1.new_set('feature')
gdgs = gdt2.new_set('graph')
graphdata = GDUtilities.gc_content(genbank_entry.seq, 100)
graph = gdgs.new_graph(graphdata, 'GC content', style='bar',
                       colour=colors.violet, altcolour=colors.purple)
for feature in genbank_entry.features:
    if feature.type == 'CDS':
        gdfs.add_feature(feature, colour=colors.red)
gdd.renumber_tracks(4)
gdd.draw(format='circular', orientation='landscape',
         tracklines=0, pagesize='A5', fragments=5, circular=1)
gdd.write('NC_005213.ps', 'PS')
```

6.8 How Do I Place One Graph on Top of Another?

Since each GDGraphSet object can hold more than one graph, it is possible to place two graphs or more in each graph set, though legibility can be a problem. When building the diagram, ecah graphset's graphs are added in order of increasing id. Since ids are assigned sequentially in order of addition of the graph to the graphset, graphs are also drawn in order of addition to the graphset. It is probably best that line graphs are added to graphsets after bar or heat graphs, or else the thin lines will be obscured by the thick bars or heat blocks.

Also, the thickness of lines in line graphs may obscure the underlying heat or bar graphs at smaller page sizes, so it may be necessary to increase page size for vector graphics, and adjust fonts, etc. accordingly, in order to reduce linewidths enough to make the underlying graphs visible. Example code is given below, adding first a bar graph, then a line graph in contrasting colour, and then increasing page size to A3, as the line thickness is too great for A5::

```
from Bio import GenBank
from reportlab.lib import colors
parser = GenBank.FeatureParser()
fhandle = open('/data/genomes/Bacteria/Nanoarchaeum_equitans/NC_005213.gbk', 'r')
genbank_entry = parser.parse(fhandle)
fhandle.close()
```

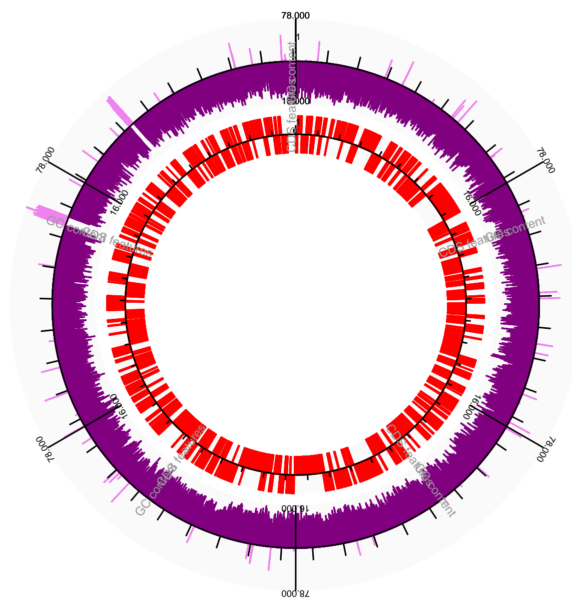

Figure 11: Output from section 6.7, with the graph track now twice the size of the feature track.

```
from GenomeDiagram import GDDiagram, GDUtilities
gdd = GDDiagram('NC_005213.gbk')
gdt1 = gdd.new_track(4, greytrack=1, name='CDS features')
gdt2 = gdd.new_track(6, greytrack=1, name='Graphs', height=2)
gdfs = gdt1.new_set('feature')
gdgs = gdt2.new_set('graph')
graphdata1 = GDUtilities.gc_content(genbank_entry.seq, 100)
graphdata2 = GDUtilities.gc_skew(genbank_entry.seq, 100)
graph1 = gdgs.new_graph(graphdata1, 'GC content', style='bar',
                        colour=colors.violet, altcolour=colors.purple)
graph2 = gdgs.new_graph(graphdata2, 'GC skew', style='line',
                        colour=colors.lightgreen)
for feature in genbank_entry.features:
    if feature.type == 'CDS':
        gdfs.add_feature(feature, colour=colors.red)
gdd.draw(format='circular', orientation='landscape',
         tracklines=0, pagesize='A3', fragments=5, circular=1)
gdd.write('NC_005213.ps', 'PS')
```

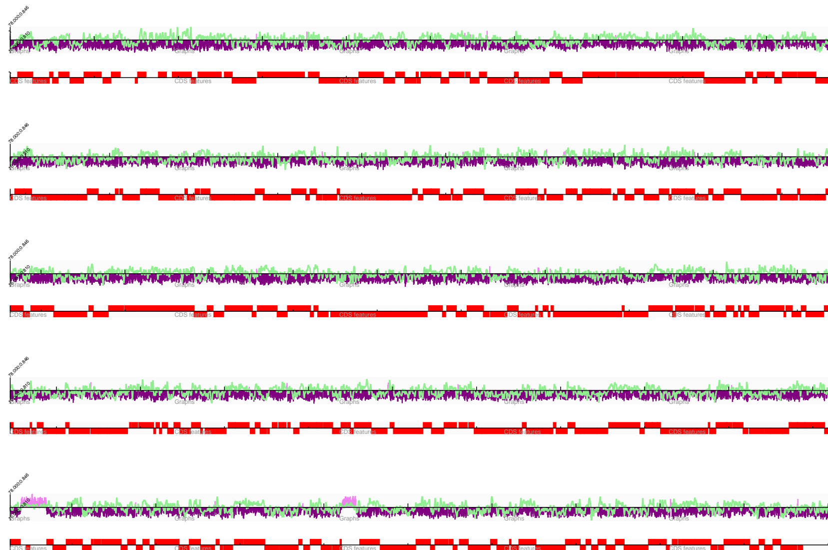

Figure 12: Output from section 6.8, with a line graph of GC skew (green) superimposed on a bar graph of GC content (violet/purple).

7 Acknowledgements

I would like to thank Ian Toth for the excuse to write this package, Kim Rutherford and Keith James for some example Perl code, and Robin Becker for invaluable help with ReportLab.