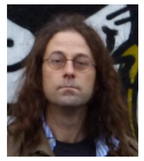
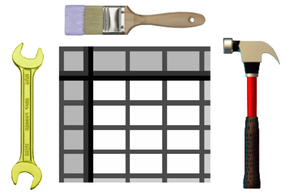
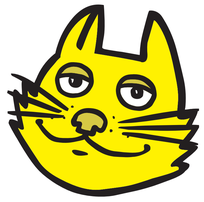
TOPCAT Plotting from STILTS API and Command Line

Mark Taylor, University of Bristol, UK



University of BRISTOL





Introduction

TOPCAT is a desktop GUI application for analysis of tabular data, particularly source catalogues. Among other capabilities it provides high-performance interactive visualisation for large (and small) datasets.

The plotting capabilities are focussed on representations of point clouds in two or three dimensions, with special attention to large (many row) and high-dimensional (many column) datasets. Many options are available.

The visualisation is supported by a custom Java plotting library. This library is now bundled and documented as part of the STILTS package (v3.0) for use outside of TOPCAT. All TOPCAT's plotting capabilities can be used.

This poster describes how you can use the library to generate plots from your own Java application code, or from the STILTS command-line interface.

Plot Model

Each plot is specified as a *plot surface* giving the geometry and zero or more *plot layers*. Different layers may use the same or different datasets, allowing all kinds of overplotting.

Surfaces: plane, sky, cube, sphere, time

Layers: scatter plot, lines, contours, analytic function, error bars, ellipses, pair links, text labels, vectors, sized markers, histogram, spectrogram, colour coding by density or additional coordinates, ...

Features

Fast: interactive graphics works well for several million points
Scalable: plot arbitrarily large datasets in fixed memory
Configurable: many plot types and options
Extensible: pluggable architecture allows runtime extensions

Further Information

Downloads and full documentation:

TOPCAT: http://www.starlink.ac.uk/topcat/
STILTS: http://www.starlink.ac.uk/stilts/

Expect enhancements in future releases. STILTS plotting is working but still somewhat experimental. Email m.b.taylor@bristol.ac.uk for assistance!

STILTS Invocation

The STILTS package provides access to TOPCAT functions from the command line or a Jython front end. New commands `plot2plane`, `plot2sky`, `plot2cube` etc produce interactive plots on the screen or bitmapped/vector output files, also incorporating STILTS's sophisticated pipeline processing options. Animations can also be generated.

API Invocation

There are two ways to configure a plot: the low-level API or key-value pairs.

Low-Level API

Setting up a plot with the low-level API is fairly complex, but it provides compile-time checking. Full javadocs are provided.

Key-Value API

Only a small number (∼4) of values must be supplied to draw a simple scatter plot with default settings, but many more (∼100) parameters are available for complex plots or fine tuning.

Options can always be set with string values, facilitating command-line and inter-process control. Alternatively, from the API values can be given as typed java Objects for convenience and extensibility.

Configuration keys are comprehensively documented in the STILTS user document, but are also self-documenting objects, with methods to return user documentation and GUI components. Application code can either use hard-coded configuration keys, or build its own graphical/text UI by interrogating the plotting API (TOPCAT and STILTS do the latter).

Data Input

Plot data is supplied as a `StarTable` object. This may be a file read from disk in one of STIL's supported formats (FITS, VOTable, CSV, ...) or a custom iterable over user-supplied data rows. The data may be static or may change with time for an animated plot.

Graphics Output

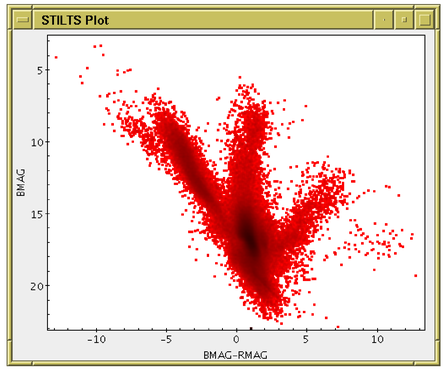
Plots can produce a live resizable and interactive graphics window (`JComponent`) posted to the screen or under control of a host application. This supports interactive user navigation: in 2 dimensions pan and isotropic or anisotropic zoom; in 3 dimensions rotation, pan, isotropic or anisotropic zoom, and recentering.

Alternatively, graphics can be exported to bitmapped (PNG, GIF, JPEG) or publication quality vector (PDF, EPS) graphics files.

Invocation Examples

Command Line

```
stilts plot2plane yflip=true layer_1=mark
                  in_1=data.fits x_1=BMAG-RMAG y_1=BMAG
```



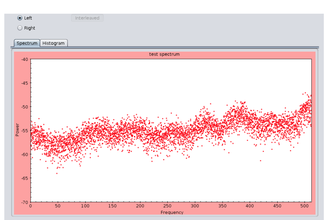
Plot output may be to
either interactive window (user can resize/pan/zoom)
or output file (bitmapped or vector)

Key-Value API

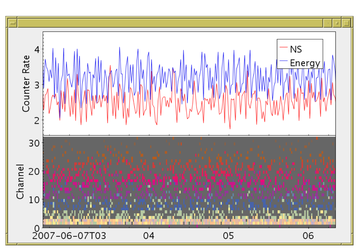
```
public JComponent createPlot() throws Exception {
    StarTable table = new StarTableFactory().makeStarTable( "data.fits" );
    MapEnvironment env = new MapEnvironment();
    env.setValue( "yflip", true );
    env.setValue( "layer_1", "mark" );
    env.setValue( "in_1", table );
    env.setValue( "x_1", "BMAG-RMAG" );
    env.setValue( "y_1", "BMAG" );
    return new PlanePlot2Task().createPlotComponent( env, true );
}
```

Low-Level API

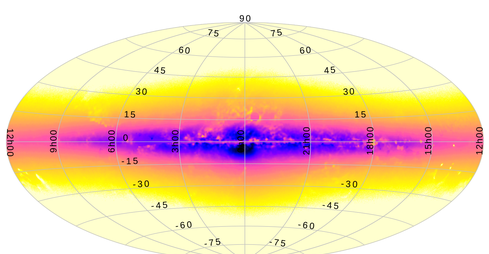
```
public JComponent createPlot() throws Exception {

    /* Read the data from an external FITS file.  We could alternatively
     * supply it from in-memory arrays or dynamically-generated values. */
    StarTable table = new StarTableFactory().makeStarTable( "data.fits" );

    /* Create the Profile for the plot surface.  This encapsulates
     * those things about the geometry and appearance of the plot
     * axes which will not change with window resizing, zooming etc. */
    DataGeom geom = PlaneDataGeom.INSTANCE;
    PlaneSurfaceFactory surfFact = new PlaneSurfaceFactory();
    boolean xlog = false;
    boolean ylog = false;
    boolean xflip = false;
    boolean yflip = false;
    String xlabel = "X axis";
    String ylabel = "Y axis";
    Captioner captioner = new BasicCaptioner();
    double xyfactor = Double.NaN;
    boolean grid = false;
    double xcrowd = 1;
    double ycrowd = 1;
    boolean minor = true;
    Color gridColor = Color.BLACK;
    Color axlabelColor = Color.BLACK;
    PlaneSurfaceFactory.Profile profile =
        new PlaneSurfaceFactory
            .Profile( xlog, ylog, xflip, yflip, xlabel, ylabel, captioner,
                      xyfactor, grid, xcrowd, ycrowd, minor, gridColor, axlabelColor );

    /* Set up a plot Aspect.  This is the initial data range,
     * and is subject to change by user navigation. */
    double[] xlimits = new double[] { 0, 30 };
    double[] ylimits = new double[] { 0, 24 };
    PlaneAspect aspect = new PlaneAspect( xlimits, ylimits );

    /* Set up a Navigator which determines what mouse gestures are
     * available to the user for plot pan/zoom etc. */
    double zoomFactor = StyleKeys.ZOOM_FACTOR.getDefaultValue();
    boolean xZoom = true;
    boolean yZoom = true;
    boolean xPan = true;
    boolean yPan = true;
    double xAnchor = Double.NaN;
    double yAnchor = Double.NaN;
    Navigator<PlaneAspect> navigator =
        new PlaneNavigator( zoomFactor, xZoom, yZoom, xPan, yPan, xAnchor, yAnchor );

    /* We will not use optional decorations for this plot. */
    Icon legend = null;
    float[] legPos = null;
    ShadeAxisFactory shadeFact = null;
    Range shadeFixRange = null;
    boolean surfaceAuxRange = false;

    /* Prepare the data for the scatter plot layer:
     * use the first two columns of the supplied table as X and Y.*/
    int[][] coordIndices = { {0}, {1} };
    DataSpec dataSpec = new ColumnDataSpec( table, geom.getPosCoords(), coordIndices );

    /* Prepare the graphical style of the scatter plot layer:
     * it's a scatter plot with single-position markers, plotted in a single fixed colour. */
    ShapePlotter plotter = ShapePlotter.createFlat2dPlotter( MarkForm.SINGLE );
    MarkShape shape = MarkShape.FILLED_CIRCLE;
    int size = 2;
    Outliner outliner = MarkForm.createMarkOutliner( shape, size );
    Stamper stamper = new ShapeMode.FlatStamper( Color.RED );
    ShapeStyle style = new ShapeStyle( outliner, stamper );

    /* Prepare the list of plot layers; in this case there is only one. */
    PlotLayer[] layers = { plotter.createLayer( geom, dataSpec, style ), };

    /* Prepare the data cache. */
    int nl = layers.length;
    DataSpec[] dataSpecs = new DataSpec[ nl ];
    for ( int il = 0; il < nl; il++ ) {
        dataSpecs[ il ] = layers[ il ].getDataSpec();
    }
    DataStoreFactory storeFact = new SimpleDataStoreFactory();
    DataStore dataStore = storeFact.readDataStore( dataSpecs, null );
    boolean dataMayChange = false;
    boolean caching = ! dataMayChange;

    /* Finally construct, size and return the plot component. */
    Compositor compositor = Compositor.SATURATION;
    PaperTypeSelector ptSel = PlanePlotType.getInstance().getPaperTypeSelector();
    JComponent comp =
        new PlotDisplay<PlaneSurfaceFactory.Profile,PlaneAspect>(
                layers, surfFact, profile, aspect, legend, legPos, shadeFact, shadeFixRange,
                ptSel, compositor, dataStore, surfaceAuxRange, navigator, caching );
    comp.setPreferredSize( new Dimension( 500, 400 ) );
    return comp;
}
```



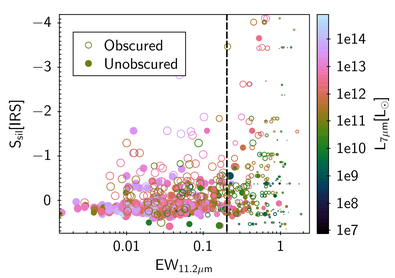
Real-time observing band spectrum,
8 thousand points refreshed at 1 Hz (easily)
(P. Harrison, Jodrell Bank)



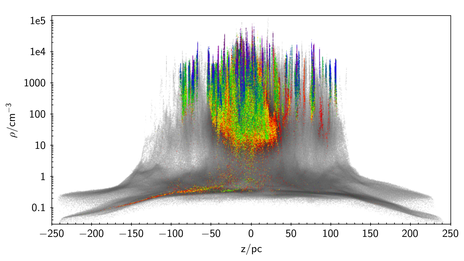
Spectrogram and samples on scrollable time axis



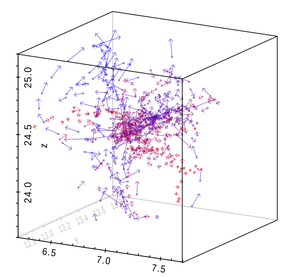
Sky source density map (GUMS-10), 2.1 billion rows, ∼30 minutes to plot



Points coded by marker shape, size and colour
(E. Hatziminaoglou, ESO)



SPH simulation data, 14 million points, 8 seconds to plot
(R. Smilgys, St. Andrews)



Millennium simulation data, positions and velocities

$Id: poster.tex,v 1.17 2014/09/30 09:35:30 mbt Exp $