THUNDER

C++11 Data Analysis and Graph Plotting Library

User Manual

(Version 1.0)

2016-04-08

E.C. Birdsall (8386390)
The University of Manchester
School of Physics and Astronomy
edward.birdsall@student.manchester.ac.uk

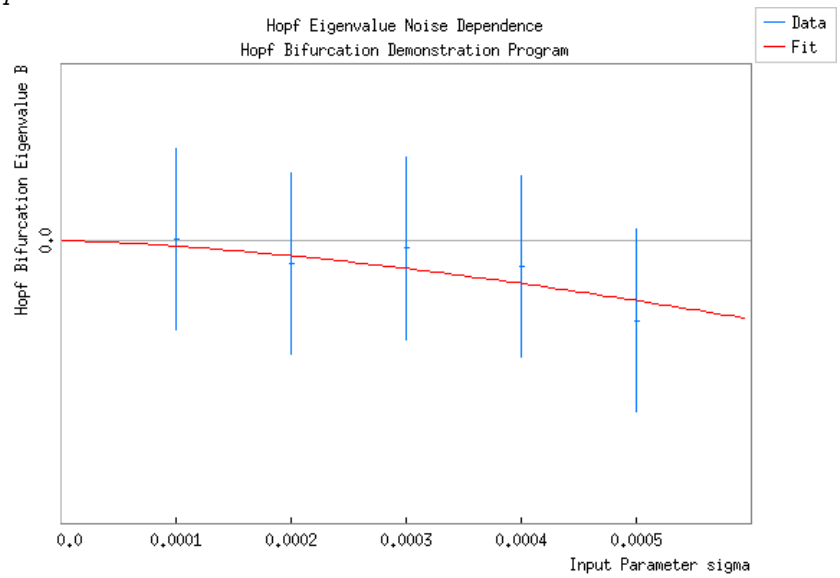
April 2016

Print color document

Abstract

THUNDER.[1] A C++11 compliant data-analysis and 2D graph plotting library, based on the CERN Root data analysis framework.

A simpler CERN Root.



1 Introduction

THUNDER is a lightweight 2D graph plotting library which can be seamlessly integrated into C++ 11 code. By design, a minimal number code lines are required to graph data. THUNDER is provided with a basic "SpreadSheet-like" container library with automatic memory management. THUNDER and the THUNDER support library are designed to be easily user extensible.

THUNDER is based on the graph plotting components of the CERN Root data analysis framework. Root is a framework specialized for analysis of particle physics data. THUNDER is a general graph plotting library; it is lightweight and convenient to use. The developers are currently unaware of any C++ library which can be used to plot data during runtime of a C++ program. THUNDER satisfies this requirement.

The principle of operation of THUNDER is the manipulation of an array of bytes which represent pixels to be written to a bitmap file. [1] [2] THUNDER systematically manipulates the contents of this array in memory to create the representation of a graph in a bitmap format. The array is written to disk, with a standard format bitmap file header, which can be opened with any standard image viewing software. THUNDER uses advanced C++ 11 features such as move semantics and *copy and swap* [3] for exception safe code, RAII [4], templates, polymorphism, exception handling, lambda functions and smart pointers [5] or STL containers [6] for memory allocation.

THUNDER is efficient, THUNDER is Object-Oriented, THUNDER is syntactically minimal.

1.1 Preamble

Conventions used in this document are as follows. `monospace` font indicate code. Function names are given with parentheses `my_function()`, but no arguments or return type by convention. In many instances, source code should be consulted for implementation details including function return types and function prototypes. Namespaces `my_ns::` are given with trailing colons. The anonymous namespace is `::`.

2 Quick Start Guide

THUNDER is primarily a graph plotting *library*, however an example program is provided to demonstrate the features of THUNDER. The following instructions are for Linux. Windows users can open the Visual Studio 2013 Project file. To compile the example THUNDER program, execute the bash script `./build.sh` This produces the executable THUNDER. If GSL is not installed, build the program with `./build_nogsl.sh` The line `#define GSL_ENABLE 1` must be changed to `#define GSL_ENABLE 0` at the start of `./include/hopf_main.hpp`. The demo program accepts the following command line arguments.

- `./THUNDER --run-THUNDER-demo`
  This executes the `THUNDER` demo program. Example outputs, including basic graph, multi-graph and histogram are produced.

- `./THUNDER --run-hopf-demo`
  This executes the Hopf Bifurcation simulation demo. (See Appendix.)

  NOTE: Requires GSL-2.1. Program must be compiled with `#define GSL_ENABLE 1`.

  NOTE: The output directory `./bmp` must be created before running this demo to enable per-timeseries output.

  WARNING: Several hours of computation time are required to complete this analysis.

  WARNING: Several gigabytes of disk space are required on the device pointed to by `./bmp`.

3 Code Design and Implementation

3.1 File Organization

`THUNDER` is designed as a "header-only" library. This is primarily to make compilation trivial. `#include "THUNDER"` is required to obtain functionality supported by the library. The individual files are listed below. The `loground.cpp` and `Color.cpp` must be compiled and linked to any program which uses `THUNDER`.

- `AxisLimit.hpp`. Contains `AxisLimit` class. Holds data for graph axis limits.
- `BMPBuilder.hpp`. Contains `BMPBuilder` class. Interface for drawing graphs to `BMPCanvas`.
- `BMPCanvas.hpp`. Contains `BMPCanvas` class. Controls reading/writing of data in memory and to disk to draw bitmap formatted images.
- `BMPFont.hpp`. Contains `BMPFont` and `BMPFontElement` classes. Manages loading of bitmap formatted font file, and separation of bitmap data into individual letters. ASCII characters [7] in the range `' '` (space, 32) to `'~'` (tilde, 126) can be printed. Any characters which fall outside this range are drawn as a shaded box.
- `BMPGraph.hpp`. Contains `BMPGraph` class. Manages data to be drawn to `BMPCanvas` object via `BMPBuilder`, axis ranges and graph labels.
- `Color.hpp`. Contains `Color` class. Represents 24 bit RGB / BGR color.
- `DrawMode.hpp`. Contains `DrawMode` class and enum classes to control drawing of data to `BMPCanvas`.
- `GraphArea.hpp`. Contains `GraphArea` class. Manages data to represent region where graph is drawn. Contains functions to convert between coordinate systems.

- `GraphData_Base.hpp`. Contains all base class types for data classes. (See data inheritance hierarchy.)
- `GraphData_Generic.hpp`. Contains intermediate class types for data classes.
- `GraphData.hpp`. Contains `GraphData` and `GraphData_Drawable` classes. There are implementations of usable data containers.
- `HistogramData.hpp`. Contains `HistogramData` and `HistogramData_Drawable` classes. These are implementations of usable data containers for histograms.
- `dynarray.hpp`. Contains basic implementation of `dynarray` class. Do not confuse with C++ 14 `std::experimental::dynarray`. [8]
- `loground.hpp`. Contains logarithmic rounding support function for drawing graph axis ticks.
- `loground.cpp`, `Color.cpp`. The only two "non-header-file" components of `THUNDER`.
- Other files are not specifically part of the `THUNDER` library.

3.2 Data Type Inheritance Hierarchy

`THUNDER` provides support for basic types of data required to plot 2D graphs. The corresponding inheritance hierarchy diagram is shown in Figure 1. The user may define classes by inheriting from components of the hierarchy; a suggested format is indicated.

The hierarchy is designed to separate fundamentally differing groups of data and corresponding functions into encapsulated modules which can be used to design classes containing only components required by the user.

- `GraphData_Base`. Abstract base class from which all other components inherit.
- `GraphData_Base_Legend`. Base class with data and functions required to draw a legend.
- `GraphData_Base_Drawable`. Declares pure virtual function `void Draw()`. Any data object which can be drawn to `BMPCanvas` must inherit from this class, forcing an implementation of `void Draw()` to be defined. This class also contains the data `DrawMode`. (See Section 3.4.)

```
void Draw(BMPCanvas& canvas, const GraphArea& graph_area,
    const AxisLimit& axis_limit) const
```

- `GraphData_Generic`. Declares functions required for all data types. This class allows for planned future modifications.
- `GraphData_Generic_Drawable`. Declares a base class pointer for use when storing the address of data to be drawn.

```
// Use iterator to draw all pointers to data
std::vector<GraphData_Generic_Drawable*> my_data;
for(std::vector<GraphData_Generic_Drawable*>::iterator
    it = my_data.begin(); it != my_data.end(); ++ it)
{
    (*it)->Draw(BMPCanvas, GraphArea, AxisLimit);
}
```

- `GraphData_Generic_Interface`. Template class, interface between generic data type, which represents data, `GraphData_Base_Legend` and `GraphData_Base_Drawable`.
- `GraphData`, `HistogramData`. Implement data and functions required to store x-y data or histogram data.
- `GraphData_Drawable`, `HistogramData_Drawable`. These classes inherit from `GraphData_Generic_Interface` forcing an implementation of `void Draw()`.

3.3 THUNDER Graph Drawing Class Organization

`THUNDER` is organized into several OOP components which handle different aspects of memory management and graph drawing. The main classes are summarized below.

3.3.1 `BMPCanvas`

This class manages an array to represent a *bitmap canvas*. Bounds checks are included to prevent the user from performing illegal memory write operations. The function `SaveAs()` writes the raw memory content to a correctly formatted bitmap file, with standard bitmap file header. [1] [2] A `try-catch` block wraps `std::ofstream::open()` to enable error reporting and recovery. If an error is raised when writing a `BMPCanvas` to file, it is likely that the output path does not exist or the device is out of space.

Memory is managed `dynarray` (Dynamic Array) class. The implementation of this class can be found in the `dynarray.hpp` header. `dynarray` is a member of the `non_std::` namespace, to make explicit the distinction between `std::experimental::dynarray` STL (Standard Template Library) class of C++ 14. [8] `dynarray` uses the copy-and-swap idiom for safe memory management. [3]

`BMPCanvas` contains implementations of low-level primitive drawing algorithms; points, circles, rectangles and lines can be drawn by calling `SetPixel()`, `DrawCircle()`, `DrawRect()`, `DrawLine()`. The functions `DrawCircle()` and `DrawLine()` are overloaded to take an additional argument of type `GraphArea`. No points outside the graphable area specified are drawn when the overloaded functions are called. The `DrawLine()` function contains an implementation of *Bresenham's Line Algorithm* [9] implemented entirely using

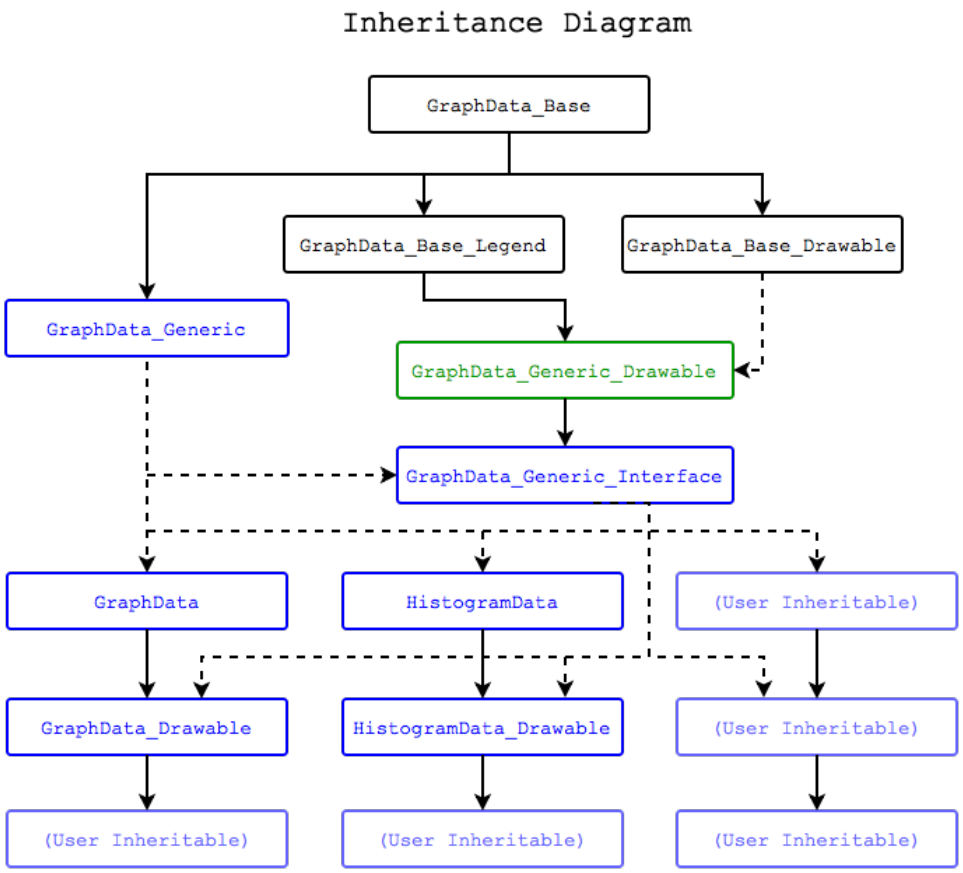


Figure 1: Data Types inheritance structure diagram. Non-template classes are black, template classes are dark blue. Suggested user inheritable hierarchy extension points shown in light blue. The inheritance point `GraphData_GenericDrawable` used as a base class pointer is highlighted in green. Dashed lines indicate virtual inheritance, solid lines indicate regular inheritance.

integer arithmetic for increased performance. It is not expected that the end-user make use of these functions except in extending the data `Draw()` functions or when implementing a custom data type and container. Refer to the header file `BMPCanvas.hpp` in conjunction with `GraphData.hpp` or `HistogramData.hpp` for template.

3.3.2 `BMPGraph`

This class manages data required to draw a graph to `BMPCanvas` object.

Class constructor. No default constructor is provided. The x and y axis ranges are specified as arguments to the constructor.

```
// Constructor
// Specifies the graph axis range to be used when plotting
// x_min - x axis range lowerbound
// x_max - x axis range upperbound [!] (x_max > x_min) [!]
// y_min - y axis range lowerbound
// y_max - y axis range upperbound [!] (y_max > y_min) [!]
BMPGraph::BMPGraph(const double x_min, const double x_max,
    const double y_min, const double y_max)
```

`AddData()` function. Sets of data to be drawn using a `BMPGraph` object are registered to each instance using the `BMPGraph::AddData()` function. Data sets are tracked using a `std::vector` of base class pointers, type `GraphData_Generic_Drawable *`. (For efficiency purposes, the data is *not* copied; therefore it is imperative that the lifetime of instances of data storage classes are maintained until the `DrawGraph()` function is called.)

```
// Add pointer to base class of type GraphData_Generic_Drawable
// to vector of base class pointers
// NOTE: Data must exist until DrawGraph() function
// is called to prevent memory corruption
void AddData(const GraphData_Generic_Drawable* const graphdata)
```

`SetAxisNumber()` function. Sets axis numerical tick label drawing flags. Graphs can be drawn with both *internal* and *external* axis tick labels enabled (`true`) or disabled (`false`). *Internal ticks* are drawn alongside the internal axis drawn inside the graph region. *External ticks* are drawn outside the graph region. Internal ticks are drawn only if the axis lie within the graphable region. If internal tick flag is set but no labels are drawn, increase axis limits.

```
// enable/disable axis tick number labels
// out_flag: enable/disable tick label numbers drawn
// outside of graph area
// in_flag: enable/disable tick label numbers drawn
// inside of graph area
// Internal ticks are drawn next to axis ticks, if
// axis ticks are not visible in graph region, then
// labels will not be drawn
void SetAxisNumber(const bool out_flag, const bool in_flag)
```

`SetLegend()` function. Enables/disables legend drawing flag. Legend is only drawn if individual data legend flags are set to `true`. The legend line color is automatically obtained from the color stored in the `DrawMode` object contained within each data object.

```
// enable/disable drawing of legend
void SetLegend(const bool draw_flag)
```

Several text drawing functions are exposed for user calls. These functions use specialized coordinate systems, each of which is designed for a specific task. All function convert to the *pixel coordinate system*, hence only `DrawTextPixel()` function is called to execute the text drawing procedure. Three coordinate systems are used.

1. *Pixel Coordinate System.* Integer coordinates; range 0 to graph size in pixels. All other coordinate systems are converted to this system, via `std::round()`.

2. *Float coordinate system.* Floating point coordinates; range 0.0 to 1.0. 0.0 corresponds to pixel coordinate 0, 1.0 corresponds to the maximum pixel coordinate. (Size of graph in pixels.) Used to address coordinate independent of graph resolution.

3. *Graph coordinate system.* Floating point coordinates, corresponding to point as located on the graphable area defined by the axis limits. `x=0.0, y=0.0` corresponds to the origin. Converts location of a point as if it were a data point to be plotted to pixel coordinate. Point may be inside the graphable area, in the canvas area but outside of the graphable area, or outside the canvas area. It is safe to draw outside of the canvas area, as range-checks are implemented.

Each function is overloaded to accept additional pixel offset arguments; `offset_x` and `offset_y`. These allow the user to re-define the origin of text to be printed. The origin is "bottom-left" by default. Each text character is 6 by 13 pixels in size.[2] The origin can be re-defined as the centre of a string by setting `offset_x = -3 * text.size()`, `offset_y=-6`. *Note that y-pixel-coordinate increases "up vertical".* Vertical text can be drawn by setting `vertical_flag = true`.

The `mode` flag (default `0`) controls how text is drawn outside of the graph area. `mode = 0` all text is drawn. `mode = 1`, all letters of a string which lie inside the graphable area are drawn. `mode = 2` a string is only drawn if all letters lie within the graphable area.

[2]It is a planned development update that future versions will include different fonts and font sizes.

1. `DrawTextPixel()`. Draw text using pixel coordinate system.

```
// Draw text using pixel coordinate system, with
// additional offset in pixels
// Offset used to re-define origin of text to be printed
void DrawTextPixel(BMPCanvas& canvas, const GraphArea&
    graph_area, const std::string& text,
const uint32_t addr_x, const uint32_t addr_y,
const int32_t offset_x, const int32_t offset_y,
const bool vertical_flag,
const int mode = 0)
```

```
// Draw text using pixel coordinate system
void DrawTextPixel(BMPCanvas& canvas, const GraphArea&
    graph_area, const std::string& text,
const uint32_t addr_x, const uint32_t addr_y,
const bool vertical_flag,
const int mode = 0)
```

2. `DrawTextFloat()`. Draw text using floating point coordinate system.

```
// Draw text using floating point coordinate system,
// range 0.0 to 1.0, with additional offset in pixel
// coordinate system
// Offset used to re-define origin of text to be printed
void DrawTextFloat(BMPCanvas& canvas, const GraphArea&
    graph_area, const std::string& text,
const double pos_x, const double pos_y,
const int32_t offset_x, const int32_t offset_y,
const bool vertical_flag,
const int mode = 0)
```

```
// Draw text using floating point coordinate system,
// range 0.0 to 1.0,
void DrawTextFloat(BMPCanvas& canvas, const GraphArea&
    graph_area, const std::string& text,
const double pos_x, const double pos_y,
const bool vertical_flag,
const int mode = 0)
```

3. `DrawTextGraph()`. Draw text using graph coordinate system, as defined by graph axis ranges.

```
// Draw text using graph coordinate system, as defined by
// graph x axis range and graph y axis range, with
// additional offset in pixel coordinate system
// Offset used to re-define origin of text to be printed
void DrawTextGraph(BMPCanvas& canvas, const GraphArea&
    graph_area, const std::string& text,
const double x, const double y,
const int32_t offset_x, const int32_t offset_y,
const bool vertical_flag,
const int mode = 0)
```

```
// Draw text using graph coordinate system, as defined by
// graph x axis range and graph y axis range
void DrawTextGraph(BMPCanvas& canvas, const GraphArea&
    graph_area, const std::string& text,
const double x, const double y,
const bool vertical_flag,
const int mode = 0)
```

Two additional hybrid functions are provided. These specialist functions are used to draw numerical axis tick labels. `DrawTextGraphX()` takes an x-coordinate argument in the graph coordinate system, and a y-coordinate argument in the pixel coordinate system. `DrawTextGraphY()` takes a y-coordinate in the graph coordinate system, and an x-coordinate in the pixel coordinate system.

```
// Specialist hybrid function, draw text with x-coordinate in
// graph coordinate system and y-coordinate in pixel
// coordinate system, with additional offset in pixel
// coordinate system
void DrawTextGraphX(BMPCanvas& canvas, const GraphArea&
    graph_area, const std::string& text,
const double x, const uint32_t addr_y,
const int32_t offset_x, const int32_t offset_y,
const bool vertical_flag,
const int mode = 0)
```

```
// Specialist hybrid function, draw text with y-coordinate in
// graph coordinate system and x-coordinate in pixel
// coordinate system, with additional offset in pixel
// coordinate system
void DrawTextGraphY(BMPCanvas& canvas, const GraphArea&
    graph_area, const std::string& text,
const uint32_t addr_x, const double y,
const int32_t offset_x, const int32_t offset_y,
const bool vertical_flag,
const int mode = 0)
```

3.4 DrawMode

The style of drawing is controlled by setting `DrawMode` class flags. `PointType` specifies the style of points to be drawn. If no `pointsize` is specified, the default `1.0` is used.

```
DrawMode(const Color& color, const PointType pointtype)
```

```
DrawMode(const Color& color, const PointType pointtype, const
    double pointsize)
```

```
enum class PointType
{
    NONE,
    POINT,
    HBAR,
    PLUS,
    CROSS,
    CIRCLE
};
```

The enum class `LineType` specifies the style of line to be drawn. [3]

```
enum class LineType
{
    NONE,
    SOLID,
    DASHED
};
```

`DrawMode PointType` can be changed by calling the function

```
void SetPoint(const PointType& pointtype, const double
    pointsize)
```

The `ErrorbarType` can be set by calling one of three functions.

```
// Set errorbar type
void SetErrorbar(const ErrorbarType& errorbartype)
```

```
// Set errorbar type and width
void SetErrorbar(const ErrorbarType& errorbartype,
const double errorbarwidth)
```

```
// Set errorbar, line width and stop types
void SetErrorbar(const ErrorbarType& errorbartype,
    const double errorbarwidth,
    const ErrorbarStopType errorbarstoptypex,
    const ErrorbarStopType errorbarstoptypey)
```

The point color can be set by calling `SetColor()`.

```
void SetColor(const Color& color)
```

The `ErrorbarType` and `ErrorbarStopType` enum classes are defined as follows.

```
enum class ErrorbarType
{
    NONE,
    ERRORBARX,
    ERRORBARY,
    ERRORBARXY
};
```

[3] `LineType::DASHED` is not implemented in Version 1.0.

```
// HBAR: horizontal bars at ends of errorbars (traditional)
// WEDGE: inward pointing arrows at the end of the errorbars
enum class ErrorbarStopType
{
    NONE,
    HBAR,
    WEDGE
};
```

Errorbars are drawn with 0 size if data to be drawn does not have valid errorbar data stored. (See the demonstration program for example usage.)

3.5 Overview of GraphData Class

Two implementations of data storage classes are provided; `GraphData` and `HistogramData`. These containers operate in a similar manner from the user point of view. Two drawable draw containers are derived from these classes. The key user accessible functions of `GraphData` are described below.

The class constructor takes a pair of boolean arguments to enable/disable x and y errorbar data.

```
/* Documentation Notes: Errorbars
*
* Errorbar data can be switched on/off using the true/false
* arguments to class GraphData.
* If switched off, less storage is used and all errorbar
* values, x and y, are assumed to be 0.0.
*
* The style of errorbar to be drawn is set using the
* DrawMode::SetErrorbar() function. To draw errorbars,
* errorbar data and drawing must be enabled.
*/
GraphData(const bool x_error_enable, const bool y_error_enable)
```

Three overloads of `AddData()` which take single value arguments are implemented.

```
// Add data (x, y) pair, x and y are added to internal storage
// If x or y error flag is set to true, the value 0.0
// is added to the error storage
void AddData(const T& x, const T& y)
```

```
// Add data (x, y) pair with y error, x and y are added
// to internal storage, and y_error is added to y error
// internal storage, if y error flag is set to true
// If x error flag is set to true, the value 0.0 is
// added to the x error storage
void AddData(const T& x, const T& y, const T& y_error)
```

```
// Add data (x, y) pair with x and y error, x and y
// are added to internal storage, and x_error and
// y_error are added to x and y error internal storage
// respectively, if x and y error flags are set to true
```

```
// If either flag is false, corresponding error value
// is ignored
void AddData(const T& x, const T& y, const T& x_error,
    const T& y_error)
```

`AddData()` is overloaded to accept STL iterators. This can be used to add data from any STL container type, including `std::list`, `std::deque` and `std::vector`. [6] The `x` data iterator is used to determine the number of values to be added. There must be at least as many values available in STL containers for y value and error data.

```
// Add x,y data from iterators
template<typename InputIterator>
void AddData(InputIterator x_first, InputIterator x_last,
    InputIterator y_first)
```

```
// Add x,y data from iterators with y errors
template<typename InputIterator>
void AddData(InputIterator x_first, InputIterator x_last,
    InputIterator y_first,
    InputIterator y_error_first)
```

```
// Add x,y data from iterators with x and y errors
template<typename InputIterator>
void AddData(InputIterator x_first, InputIterator x_last,
    InputIterator y_first,
    InputIterator x_error_first,
    InputIterator y_error_first)
```

These functions can be used in conjunction with the `GetSTLVector()` function implemented in the `MathSpreadSheet*` class, as explained in Section 4.

4 Memory Operation and IO Support

`THUNDER` is provided with an implementation of a template data container. Class `Data` is a template class containing an element of "data" and an initialization flag. `SpreadSheet` is a generic 2D (matrix) container with `Get()` and `Put()` functions which retrieve and insert data respectively. Allocation is automatic; `Put(i, j, val)` ensures enough memory is allocated to store `val` at `i, j`. Allocation is done "in powers of 2" to ensure efficiency when sequentially inserting values in either direction `i` or `j`. The number of elements allocated y is given by

$$y = 2^{\lceil \log_2(x) \rceil} \tag{1}$$

if allocation for x elements is requested. The algorithm used is implemented entirely using bit-shift and count.

```
// call with size = N to allocate at least N elements
// if user requires allocation to ACCESS element N
// call with N = N + 1
size_t capacity_query(size_t size) const
{
    if(size == 0)           // unsigned integer, cannot call
        return 0;           // --, return if 0
    -- size;                // decrement input parameter to
                            // prevent exact powers of 2
                            // triggering reallocation
                            // without this: example;
                            // 8 -> return 16 (should be 8)
    size_t c = 0;           // counter init
    for(; size != 0; size >>= 1, ++ c);
    // test size == 0, if not 0, size = size >> 1, LSB fall off
    // counts number of SBL (>>) required to obtain 0 result
    // shifted size >> (SBL) by c bits to get zero result
    // hence, 1 << c will be the MSB of size if exact power
    // of 2, or MSB << 1 if not exact power of 2
    // this is equal to the nearest power of 2 required to
    // store size elements, with constraint that storage
    // allocated with size which is a power of 2
    return (1 << c);
}
```

Class `SpreadSheetManager` manages instances of `SpreadSheet` using `std::map<std::string, SpreadSheetBase*>` lookup table, associating a unique name "key" to each `SpreadSheet` instance.

`MathSpreadSheet*` are a set of classes which inherit from `SpreadSheet`. The template type `T` is substituted for `Data<T>`. `SpreadSheet` can be used for any template type, `MathSpreadSheet` can only be used with types which emulate the behaviour of numerical types. Such a type must overload *at least* `operator+`, `operator-`, `operator*` and `operator*` where the left argument is of type `double`. The possible substitutions for `*` are listed in Table 1. [10] The allowed template types are defined using `typedef`.

`MathSpreadSheet*`	template type	Description
`MathSpreadSheetf`	`float`	32 bit float
`MathSpreadSheetd`	`double`	64 bit float
`MathSpreadSheeti32`	`int32_t`	32 bit signed integer
`MathSpreadSheetui32`	`uint32_t`	32 bit unsigned integer
`MathSpreadSheeti64`	`int64_t`	64 bit signed integer
`MathSpreadSheetui64`	`uint64_t`	64 bit unsigned integer

Table 1: Substitutions for `MathSpreadSheet*` class set.

```
template<typename T>
class MathSpreadSheet : public SpreadSheet<T>
{
    // Default constructor
    MathSpreadSheet()
        : SpreadSheet<Data<T>>()
    {
    }

    // Capacity initialization constructor
    MathSpreadSheet(const size_t size_x, const size_t size_y)
        : SpreadSheet<Data<T>>(size_x, size_y)
    {
    }

    // Element initialization constructor
    MathSpreadSheet(const size_t size_x, const size_t size_y,
        const Data<T> val)
        : SpreadSheet<Data<T>>(size_x, size_y, val)
    {

    }
};

typedef MathSpreadSheet<float>    MathSpreadSheetf;
typedef MathSpreadSheet<double>   MathSpreadSheetd;
typedef MathSpreadSheet<int32_t>  MathSpreadSheeti32;
typedef MathSpreadSheet<uint32_t> MathSpreadSheetui32;
typedef MathSpreadSheet<int64_t>  MathSpreadSheeti64;
typedef MathSpreadSheet<uint64_t> MathSpreadSheetui64;
```

Calls to pre-implemented statistics functions ignore uninitialized data. The `MathSpreadSheet*` class is designed to allow the user to write custom functions for statistical data analysis and numerical operations. Some example functions have been implemented for demonstration purposes. Example `Read()` and `Write()` functions for reading and writing data to binary format file are included.

4.1 Pre-Implemented Statistical Functions

Mean

The mean [11] of a set of data is given by

$$\bar{x} = \frac{1}{N} \sum_{j=0}^{N-1} x_j. \tag{2}$$

`Mean(i)` computes the mean of column `i`. `Mean()` is overloaded to take an additional argument of type `bool`. To compute the mean of *row* `i`, call `Mean(i, true)`.

Variance and Standard Deviation

The variance [11] of a set of data is given by

$$\mathrm{Var}(x) = \frac{1}{N-1}\sum_{j=0}^{N-1}(x_j - \bar{x})^2. \tag{3}$$

The standard deviation is related to the variance by $\sigma = \sqrt{\mathrm{Var}(x)}$.

`Var(i)` and `StdDev(i)` compute the variance and standard deviation of column `i`. `Var(i, true)` computes the variance of *row* `i`.

4.2 Pre-Implemented Functions for Data Operations

Copy Rect Block of Data

```
BlockCopy(index_in_x, index_in_y, index_out_x, index_out_y,
    size_x, size_y)
```

Copies block of data from `index_in_x, index_in_y` to `index_out_x, index_out_y`, size `size_x, size_y`.

Warning: Blocks of data to be copied must not overlap.
Warning: Block of data as input must not exceed bounds.
Note: Block of data as output may exceed bounds.

Clear Rect Block of Data

```
BlockClear(index_x, index_y, size_x, size_y)}
```

Sets all initialized flags to `false` for all `Data` elements inside block, starting at location `index_x, index_y`, size `size_x, size_y`. This function is used after `BlockCopy()` to emulate a move operation. The data values in memory are unchanged.

Copy Column

`ColCopy(index_in, index_out)` copies column of data from `index_in` to `index_out`. The input index must not exceed valid bounds.

Clear Column

`ColClear(index)` sets all initialized flags to `false` for all `Data` elements in column `index`.

Copy Row

`RowCopy(index_in, index_out)` copies row of data from `index_in` to `index_out`. The input index must not exceed valid bounds.

Clear Row

`RowClear(index)` sets all initialized flags to `false` for all `Data` elements in row `index`.

Return Row or Column as STL Vector

This function is used to send data from a `MathSpreadSheet*` object to a `BMPGraph` object. `GetSTLVector(i)` returns `std::vector<T>` of all initialized elements in column `i`. `GetSTLVector(i, true)` returns all initialized elements of *row* `i`.

4.3 Pre-Implemented Mathematical Operations

These functions are provided as examples. It is expected that the end user write required functions depending on the usage context.

Elementwise Block Add-And-Put

Adds two blocks of data storing result in alternate location.

```
void BlockAdd(const size_t index_in_x_1,
    const size_t index_in_y_1,
    const size_t index_in_x_2, const size_t index_in_y_2,
    const size_t index_out_x, const size_t index_out_y,
    const size_t size_x, const size_t size_y)
{
    size_t y_1 = index_in_y_1;
    size_t y_2 = index_in_y_2;
    size_t out_y = index_out_y;
    for(; y_1 < index_in_y_1 + size_y;
        ++ y_1, ++ y_2, ++ out_y)
    {
        size_t x_1 = index_in_x_1;
        size_t x_2 = index_in_x_2;
        size_t out_x = index_out_x;
        for(; x_1 < index_in_x_1 + size_x;
            ++ x_1, ++ y_2, ++ out_x)
        {
            SpreadSheet<T>::index(out_x, out_y) =
                SpreadSheet<T>::index(x_1, y_1) +
                SpreadSheet<T>::index(x_2, y_2);
        }
    }
}
```

Elementwise Column Linear-Re-Scale

Performs a per-element linear re-scale by $y = mx + c$. Each element x is set to y.

```
void ColLinearRescale(const size_t index, const double m, const
     double c)
{
    for(size_t y = 0; y < SpreadSheet<T>::SizeY(); ++ y)
    {
        SpreadSheet<T>::index(index, y).SetValue(m *
            SpreadSheet<T>::index(index, y).Value() + c);
    }
}
```

5 Examples

5.1 Graph with Y-Errorbars

The code which follows produces output Figure 2. The user is responsible for loading data into memory. In this example, a `for` loop is used to generate data.

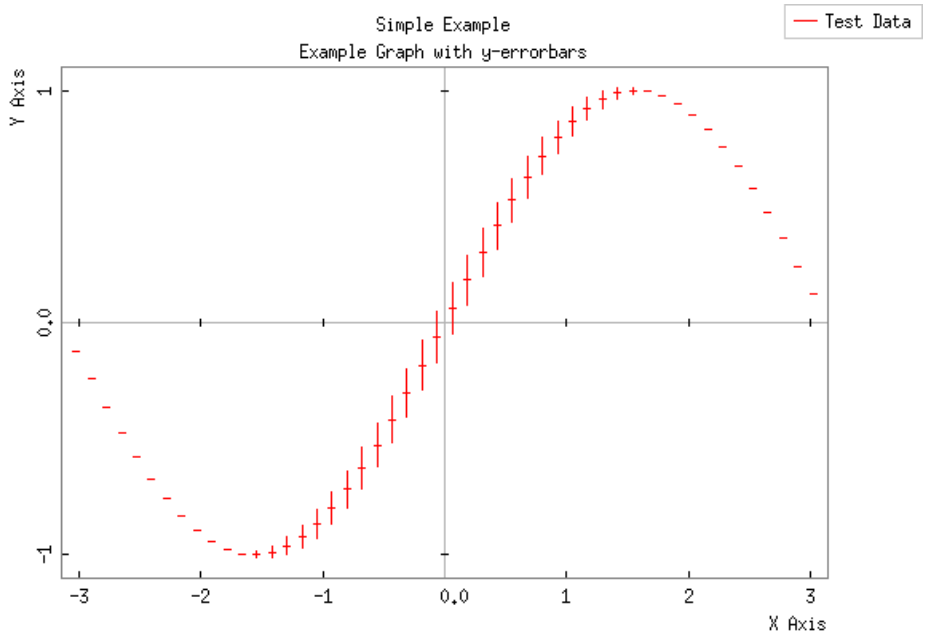


Figure 2: Bitmap graph output produced by example code. The point style `PointType::HBAR` draws points as horizontal bars, improving the clarity of errorbars when drawn without *endstops*.

The canvas size is set in the `BMPBuilder` constructor call, however the graph size is specified when calling `DrawGraph()`. This enables multiple graphs to be drawn to the same `BMPCanvas`. A graph can be drawn at any location, with any size, provided the graph fits within the canvas size specified in the `BMPBuilder` constructor. The warning message `Warning: Out of`

`bounds index in BMPCanvas::SetPixel()` will be printed for each pixel which is drawn out of bounds.

Note that `using namespace THUNDER;` is assumed.

```
// Create BMPBuilder object with canvas size 600 by 400
BMPBuilder builder(600, 400);

// Create graph with limits; x -> [-PI, PI], y -> [-1.1, 1.1]
BMPGraph graph(-PI, PI, -1.1, 1.1);
graph.SetTitle("Simple Example");
graph.SetSubtitle("Example Graph with y-errorbars");
// enable drawing of axis tick numbers outside graph area, no
    internal tick numbers
graph.SetAxisNumber(true, false);
// enable drawing of legend
graph.SetLegend(true);
graph.SetXAxisLabel("X Axis");
graph.SetYAxisLabel("Y Axis");

// Create data object, stored as type double
// no xerrorbar storage
// yerrorbar storage enabled
GraphData_Drawable<double> data(false, true);

// Set legend text
data.SetText("Test Data");

// plot 20 points (index starting at 1)
size_t num_points = 51;
for(size_t i = 1; i < num_points; ++ i)
{
double x = PI * ((2.0 / (double)num_points) * (double)i - 1.0);
double y = std::sin(x);
double y_err = 0.1 * std::cos(x) + 0.01;

// add data, xerror ignored as xerrors are disabled
data.AddData(x, y, 0.0, y_err);
}

// Set drawmode
// color red, horizontal bar point type, 5 pixels wide
DrawMode drawmode(Color(255, 0, 0), PointType::HBAR, 5.0);
// enable drawing of yerrorbars using yerrorbar data
drawmode.SetErrorbar(ErrorbarType::ERRORBARY);

// Set data drawmode
data.SetDrawMode(drawmode);

// Add pointer to data to graph
graph.AddData(&data);

// draw a 600 by 400 graph starting at location 0,0
// using data registered with instance of BMPGraph; graph
builder.DrawGraph(graph, 0, 0, 600, 400);
```

```
// Save the graph canvas to disk in bitmap format
// the file type is detected from the extension ".bmp"
builder.SaveAs("example.bmp");
```

5.2 Histogram Graph

The example code below demonstrates the use of `HistogramData_Drawable` to create a histogram.

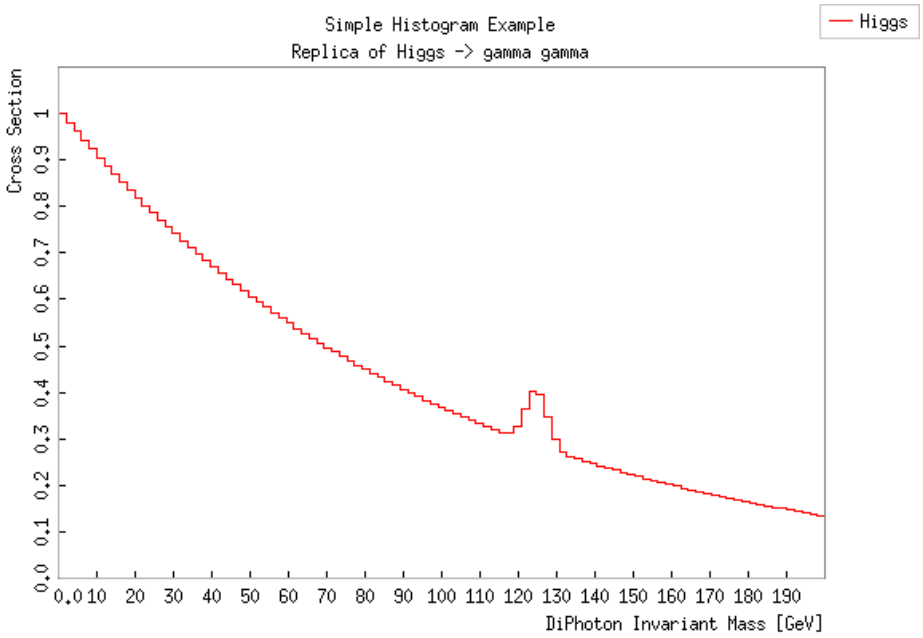


Figure 3: Example histogram output, using false higgs data.

```
// Create BMPBuilder object with canvas size 600 by 400
BMPBuilder builder(600, 400);

// Create graph with limits; x -> [0, 200.0], y -> [0.0, 1.1]
BMPGraph graph(0.0, 200.0, 0.0, 1.1);
graph.SetTitle("Simple Histogram Example");
graph.SetSubtitle("Replica of Higgs -> gamma gamma");
// enable drawing of axis tick numbers outside graph area, no
    internal tick numbers
graph.SetAxisNumber(true, false);
// enable drawing of legend
graph.SetLegend(true);
graph.SetXAxisLabel("DiPhoton Invariant Mass [GeV]");
graph.SetYAxisLabel("Cross Section");

// Create data object, stored as type double, bin values type
    double
```

```
double XMAX = 202.0; double XMIN = 0.0;
double XRANGE = XMAX - XMIN;
size_t N = 101;
HistogramData_Drawable<double, double> data(XMIN, XMAX, N);

// Set legend text
data.SetText("Higgs");

// generate replica higgs data
for(double x = XMIN; x < XMAX + 0.5 * XRANGE/(double)N;
    x += XRANGE/(double)N)
{
    double arg = (x - 125.0) / 4.0;
    double wt = 0.12 * std::exp(-std::pow(arg, 2.0));
    double wt2 = 1.0 * std::exp(-0.01 * x);
    data.Fill(x, wt + wt2);
}

// Set drawmode color to red
DrawMode drawmode(Color(255, 0, 0), PointType::NONE, 0.0);
// Set data drawmode
data.SetDrawMode(drawmode);
// Add pointer to data to graph
graph.AddData(&data);
// draw a 600 by 400 graph starting at location 0,0
// using data registered with instance of BMPGraph; graph
builder.DrawGraph(graph, 0, 0, 600, 400);
// Save the graph canvas to disk in bitmap format
// the file type is detected from the extension ".bmp"
builder.SaveAs("higgs.bmp");
```

6 Conclusion

THUNDER demonstrates the power and utility of a small 2D graph plotting library which can be seamlessly integrated into C++ code. Future releases of THUNDER will

- Expand histogram graph drawing functionality.
- Provide more image output formats such as PDF (Portable Document Format) and PNG (Portable Network Graphics). These compressed formats are significantly more efficient with disk usage.
- Improve plotting output with variable font sizes, a choice of fonts, and anti-aliased plotting.

References

[1] Bitmap File Format Microsoft Developer Page
`https://msdn.microsoft.com/en-us/library/windows/desktop/dd183386(v=vs.85).aspx`

[2] Bitmap File Format Wikipedia Page
`https://en.wikipedia.org/wiki/BMP_file_format`

[3] C++ Copy & Swap Idiom
`https://chara.cs.illinois.edu/sites/cgeigle/blog/2014/08/27/copy-and-swap/`
`http://stackoverflow.com/questions/3279543/what-is-the-copy-and-swap-idiom`
`https://web.archive.org/web/20140113221447/http://cpp-next.com/archive/2009/08/want-speed-pass-by-value`

[4] *Resource Acquisition is Initialization*
`http://en.cppreference.com/w/cpp/language/raii`

[5] C++ 11 Smart Pointer
`https://msdn.microsoft.com/en-GB/library/hh279674.aspx`

[6] C++ STL Container, C++ STL Iterator
`http://en.cppreference.com/w/cpp/container` `http://en.cppreference.com/w/cpp/iterator`

[7] ASCII Character Table
`http://www.asciitable.com/`

[8] C++ 14 `std::experimental::dynarray`
STL runtime allocated fixed storage unit
`http://en.cppreference.com/w/cpp/container/dynarray`

[9] Bresenham's line algorithm
`https://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm`

[10] C Standard Integer
`http://en.cppreference.com/w/cpp/types/integer`

[11] W.H. Press, S.A. Teukolsky, W.T Vetterling, B.P. Flannery *NUMERICAL RECIPES The Art of Scientific Computing* Third Edition (2007). (CH14 P720 *Statistical Description of Data*)

[12] E.C. Birdsall *The Dependence of Hopf Bifurcation Eigenvalue due to the Presence of Noise* (Aug 2015)

[13] P.G. Drazin, *Nonlinear Systems*, Cambridge University Press, Cambridge, UK. (1992) ISBN-13: 978-0521406680 ISBN-10: 0521406684

[14] A. Juel, A.G. Darbyshire and T. Mullin, *The Effect of Noise on Pitchfork and Hopf Bifurcations*, Manchester Institute for Mathematical Sciences, School of Mathematics, The University of Manchester, Manchester, UK. (1997)
`https://www.escholar.manchester.ac.uk/uk-ac-man-scw:1a5968`

[15] D. Lucas, *The Effect of Noise on the Time Constants of Transient Solutions of a Van-der-Pol Chaotic Oscillator*, The University of Oxford, Oxford, UK, 1992. This document can be obtained from edward.birdsall@student.manchester.ac.uk

Hopf Bifurcation Demonstration Program

The demonstration program is a shortened version of an analysis performed during a Summer Research Project with the Non-Linear Physics Group at The University of Manchester; June - August 2015. [12] A brief summary is presented below.

Equations (4) and (5) model a system containing a Hopf Bifurcation. [12] [13] [14] [15]

$$\dot{x} = -y + x(\rho - x^2 - y^2) \tag{4}$$

$$\dot{y} = x + y(\rho - x^2 - y^2) \tag{5}$$

In polar form,

$$\dot{\theta} = 1 \tag{6}$$

$$\dot{r} = r(\rho - r^2) \tag{7}$$

Equation (6) has the trivial solution $\theta(t) = t + t_0$. The solution of Equation (7) is the Stuart-Landau Equation. The equilibrium amplitude is $\rho^{\frac{1}{2}}$, the effective initial amplitude is r_0.

$$r(t) = \frac{\rho^{\frac{1}{2}} r_0}{\sqrt{(\rho - r_0^2)\exp(-2\rho t) + r_0^2}} \tag{8}$$

The presence of noise system shifts the bifurcation point. $\epsilon(t)$ is a Gaussian distributed white noise random variable with 0 mean and variance σ^2. σ is an input parameter, read from input file `sigma.csv`. The *effective decay constant* $\tau = \rho^{-1}$ is perturbed by the presence of noise.

$$\dot{y} = x + y(\rho - x^2 - y^2) + \epsilon(t) \tag{9}$$

$$r(t) = \frac{\rho^{\frac{1}{2}} r_0}{\sqrt{(\rho - r_0^2)\exp(-\frac{2t}{\tau}) + r_0^2}}. \tag{10}$$

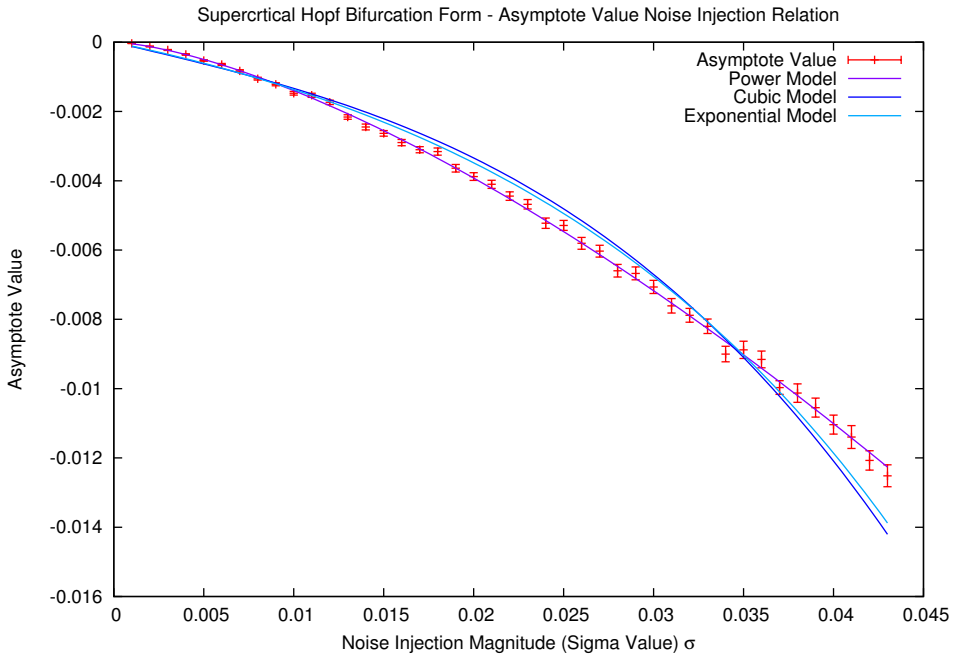


Figure 4: Dependence of bifurcation eigenvalue B on magnitude of noise injection σ. Models; Power Model $B(\sigma) = b\sigma^c$, Cubic Model $B(\sigma) = c_1\sigma + c_2\sigma^3$, Exponential Model $B(\sigma) = k\mathrm{e}^{\lambda\sigma}$.

For particular value of σ, several timeseries runs are computed for a set of values ρ, read from input file `rho.csv`. τ is obtained from a non-linear fit of Equation (10). The effective bifurcation point B is measured by computing a non-linear fit of Equation (11).

$$\tau(\rho) = \frac{1}{\rho - B}. \tag{11}$$

The objective of this analysis was to determine the dependence of B on σ. Several models were trialled. The latest set of data was obtained in October 2015, shown in Figure 4. The best fit model was $B(\sigma) = b\sigma^c$, with $b = -1.33 \pm 0.04$ (2.96 %), $c = 1.489 \pm 0.008$ (0.53 %), $\chi^2 = 0.96$.

Graphical output of data was used to check for convergence of non-linear fits, demonstrating the use of `THUNDER` in a real analysis procedure. Note that the random number generator has been pre-seeded to ensure the same results are generated each run. Remove line `gen.seed(0);` to revert to seeding using the C++ 11 `std::random_device`.

The results obtained from the demonstration program are shown in the Figure on the cover of this document.

List of Advanced C++ Features Used

The following are used consistently throughout. If a feature is used scarcely, then an example of use is given.

- Polymorphism, virtual functions, pure virtual functions, base class pointers, inheritance and virtual inheritance.
- Copy and Swap idiom, used throughout all relevant classes for code efficiency and exception safety.
- Transparent memory management. (Either via dynamic memory allocation, STL or smart pointer.)
- Move semantics used throughout when managing a resource for efficiency.
- RAII (Resource Acquisition is Initialization); memory management idiom.
- Smart Pointers. (See dynamic array implementation.)
- C++ Standard Library including; C++ 11 random number generation and distribution mapping, template containers including `std::map` and `std::vector`.
- Exception safe code and exception handling.
- Template types throughout, including functions and classes.
- Modern C++ typecasting for type-conversion safety.
- Lambda functions. (See support library.)
- Function pointers. (See `BMPCanvas.hpp`)
- Project organization with multiple header files grouped into multiple include directorys.
- Multiple and nested namespaces.

List of Acceptable Compiler Generated Warnings

A list of acceptable warnings produced by GCC 5.2 are listed below. These may vary depending on compiler. The list of warnings is different when compiling with Visual Studio.

- `-Weffc++` warning. `warning: '<X>' should be initialized in the member initialization list [-Weffc++]`
  Reason for not initializing member variables in constructor initialization lists include; syntax differences between compilers which do not conform exactly to a C++ specification, preventing initialization order conflicts with const member variables.

- `-Wunused-variable` warning. `warning: unused variable '<X>'`
  Unused variables may be remnants from old code which has been modified for use in `THUNDER` demonstration programs, or variables which were previously used and are no longer used but have not been removed because they exist in sections of almost identical code. (Code copied and pasted, may be required in future modification.)

- `-Wunused-but-set-variable` warning. `warning: variable '<X>' set but not used`
  Same as above.

- `-Wunused-parameter` warning. `warning: unused parameter '<X>'`
  Same as above.

- `-Wsign-compare` warning. `warning: comparison between signed and unsigned integer expressions` One instance of this warning exists in code imported for `THUNDER` demonstration program.