CSharpCalc v1.0

Calculation, Simulation, Visualization, Prototyping

Programming Manual

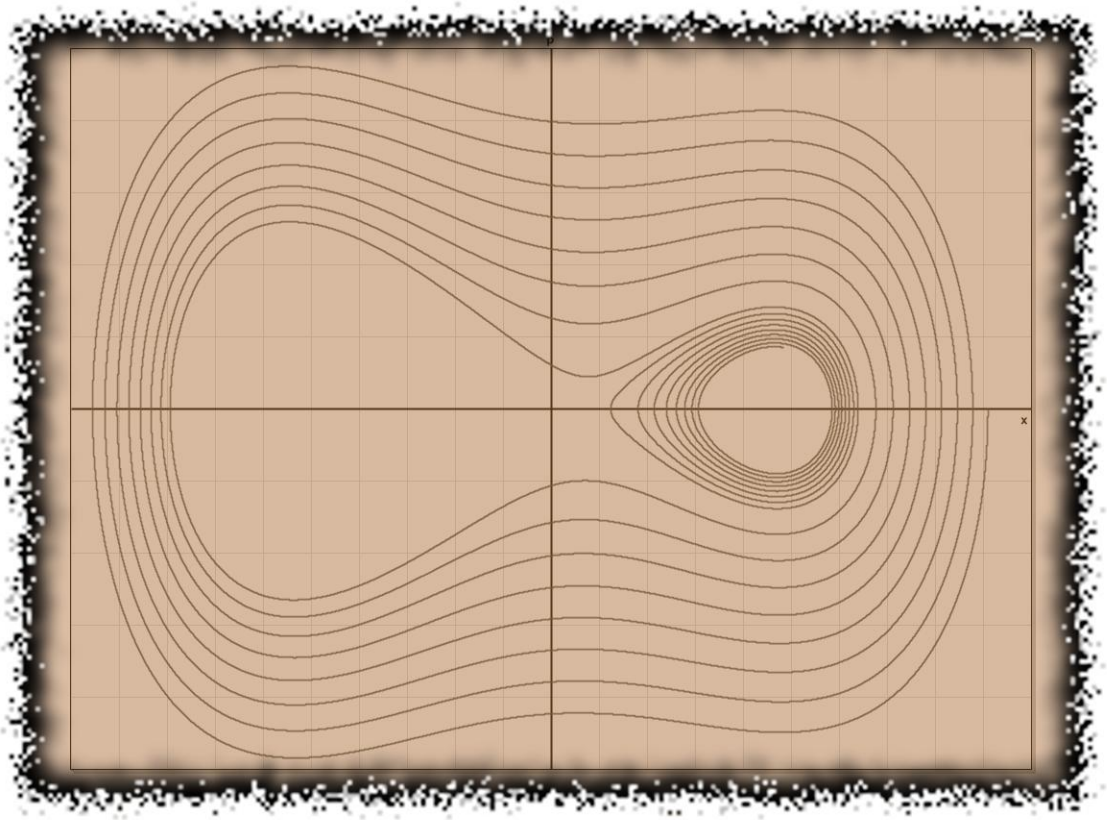

Table of Contents

Terms of Use

Preamble

This document covers selected topics concerning the use of the CSharpCalc software. It is in no way complete and must not be understood as an exhaustive documentation of CSharpCalc. This manual is part of the CSharpCalc software and must not be distributed as a separate document. The script code presented in this manual is provided under the regulations stated in the CSharpCalc license agreement.

Permission to use

You may download and use this document as part of the CSharpCalc software free of charge but without any warranty or liability. If you use this document you accept full responsibility and liability for all consequences of malfunction or faulty results caused by using any information contained in this documentation including, but not limited to, the sample scripts.

Disclaimer of warranty

There is no warranty for the information contained in this manual to the extent permitted by applicable law. This document is provided 'as is'. This includes, but is not limited to, the fitness for any particular purpose. The entire risk of any consequences of errors or ambiguities in this document is with you. Should parts of the information contained in this manual prove false, misleading, inaccurate or incomplete you assume the full cost of all related consequences.

Limitation of Liability

In no event, unless required by applicable law, shall the author of this manual be liable for any damage arising from using information contained in this document, not even if the author has knowledge of the possibility of such damage. Possible damage includes, but is not limited to, loss of data, wrong or inaccurately rendered results or the failure of CSharpCalc (the software covered by this manual) to operate with any other programs.

Distribution of this document

This document is bundled with the CSharpCalc software release it pertains to. Consequently, it is neither intended nor permitted to be distributed as a separate document. If you wish to distribute CSharpCalc, you may use the link:

http://www.csharpcalc.org/download.php

for offering downloads of the CSharpCalc package, including this manual, on your web site.

Governing law

This license agreement shall be governed by the laws of the Republic of Austria. Venue of jurisdiction is Graz, Austria.

Severability clause

In the case that any part of this license agreement is found to be invalid, the validity and legality of all remaining provisions stated in this license agreement shall not be affected or impaired thereby.

Introduction

This manual is an introductory text to be used as a starting point for working with CSharpCalc. In order to present a comprehensive primer many advanced topics have been omitted. The interested reader is referred to the sample scripts and project presentations on www.csharpcalc.org for additional information. Also, as explained in detail below, the integrated class interface documentation gives useful information on features which did not find their way into this text.

What is CSharpCalc?

CSharpCalc is a lightweight calculation and simulation environment for Windows which uses C# as scripting language. The software is heavily focused on numerical calculations and simulations within the fields of applied mathematics and science. Applicability of the software ranges from simple tasks such as the evaluation of a finite sum or the solution of a recursive equation to the investigation of complex mathematical models such as Monte Carlo simulations or systems of non-linear differential equations.

How does CSharpCalc work?

Calculations and simulations in CSharpCalc, are carried out in three-consecutive steps.

1. Write (or load) a script in C# which is able to perform all necessary calculations and which renders the results either as text or graphically (or both).
2. Execute this script within the CSharpCalc software.
3. View the results of your calculations. Output can be written to a read-only text console or it can be drawn to a graphic display. Additionally, all output can be written to text files or portable network graphics (.png) image files using built-in functionality.

In this sense, CSharpCalc is a pocket calculator with its keypad replaced by C# source code. The programming overhead required to implement CSharpCalc scripts has been kept as low as possible. For example the script:

```
long x1 = 0; long x2 = 1; long F = x1 + x2;
CSCConsole.Writeln("F0: " + x1.ToString());
CSCConsole.Writeln("F1: " + x2.ToString());
for(int i = 2; i <= 50; i++)
{
CSCConsole.Writeln("F" + i.ToString() + ": " + F.ToString());
x1 = x2; x2 = F; F = x1+x2;
}
```

writes the first 50 Fibonacci numbers to the CSharpCalc console(see below). The above code is executable in CSharpCalc as it stands here without functions or classes. A .zip file containing all example scripts used in this text is bundled with the official CSharpCalc release!

CSharpCalc comes with predefined classes (like *CSCConsole* used in the above code example) for writing output to the respective virtual devices available in the software, i.e. the console and the graphic display. Hence, there is no need (and in fact no possibility) to include any external

libraries or software frameworks in a .csc (CSharpCalc) script. Users may fully focus on the calculations they are interested in without extra bells and whistles.

The above example is deliberately simple. However, CSharpCalc can also be used to implement complex simulations. Since all .csc scripts are compiled using the C# compiler built into the .net framework, the runtime performance of such a script is equal to any compiled C# software as long as you keep the output to the CSharpCalc devices low.

Who will benefit from using CSharpCalc?

CSharpCalc users share two distinguishing features.

- A strong affinity to scientific calculations and/or simulations.
- The ability to write basic C# code or the intension to learn how to do this. Of course, more sophisticated programming skills count as well but are not required for using the software.

Due to its simplicity and low demand for programming skills CSharpCalc is well suited for people who wish to start programming in the C# language. The software can be seen as a 'back-to-the-roots' environment resembling one of the early 1980's computers which had little more than a text editor and a text or graphic display without multitasking or windowing environment.

What are the preliminaries for using CSharpCalc?

In order to use CSharpCalc you need to be able to write basic C# code. If you know the fundamentals of the language, including how to call member functions of predefined classes as explained below you are good to go. No knowledge of the .net environment is required. Also, interest in numerical mathematics is a central point since the software is focused on this.

Caveat – it's a calculator not an IDE!

Due to this specialization, CSharpCalc is not a general purpose integrated development environment (IDE). If you are looking for a general C# development suite please visit developer.microsoft.com for detailed product information. In addition to professional development environments Microsoft is also offering quality software free of charge.

Installation

CSharpCalc requires Windows 7 or a higher version of the Microsoft operating system family. The software does not require any form of installation. It comes as a single executable without libraries or additional resources and it does not make any entries in the Windows registry. Owing to this design, the software is well suited to be operated from arbitrary locations such as your PC's desktop, a portable USB disk or even a flash drive. After downloading the executable and scanning it with your antivirus software, simply copy it to a location of your choice (e.g. the Desktop) and (double)click on it.

When CSharpCalc is started for the first time, the software creates a so-called workspace in the My Documents folder of your user profile. The workspace is the default location for storing your scripts and inline files. You may keep your scripts in other locations but it is strongly recommended that you use the workspace. Inline files must be installed in the active workspace (see below). You may organize your workspace using the regular windows file manager. A detailed description is given below.

Installation as portable software (e.g. on a flash drive)

In order to install CSharpCalc on a mobile data storage copy the file CSharpCalc.exe to a location on that storage device and (double)click on this copy in order to start the software. Once the software has started up you may change the workspace location to local mode of operation. In the lower right corner of the window you will find a menu button labeled "Workspace location". Click on that button and select the option "Local folder". Once this has been accomplished you are, literally, good to go. The local workspace folder name is "CSharpCalc Workspace". It is created in the same folder that also contains your local copy of CSharpCalc.exe. (Double)click on this copy for starting the local installation every time you wish to do so. If you delete or rename the workspace folder the software will return to the workspace located in the user profile or create one if this folder does not already exists. Making a backup before deleting the workspace is strongly recommended.

The Workspace

The workspace is the default location for storing scripts and the mandatory location of inline files. CSharpCalc supports the use of folder hierarchies within the workspace as long as these hierarchies are located below the appropriate root folders named *Etc*, *Inline*, *Source* and *Templates*, each of which having a different purpose.

Etc is a system folder containing a configuration file. All settings stored in this file can be modified from within the CSharpCalc user interface. Hence, there is no reason to change anything here.

The *Inline* folder contains inline files. Inline files work similar to include files in C/C++ or PHP. In order to be usable from within a .csc script inline files must be installed in the *Inline* folder of the active workspace!

The *Source* folder is the default location for saving .csc scripts. Scripts may be stored in other locations but it is recommended to keep your scripts within the *Source* folder.

The *Templates* folder is used for storing frequently used code snippets which can be conveniently inserted into your code with a simple click. Experience shows that its contents grows over time.

You may add subfolders to the *Inline*, *Source* and *Templates* branches of the workspace and store files in these subfolders. Within .csc scripts, inline files contained in subfolders need to be referred to relatively to the root folder.

Example: If you store an inline file named *test.csc* in the *Inline* folder you need to inline it into your scripts using the code statement:

```
//inline test.csc
```

However, if you create a folder named *Test* in your Inline folder and put the file *test.csc* there you need to inline this file using the code line:

```
//inline Test\test.csc
```

Deeper hierarchies are also supported. While .csc files may be stored in folders outside the workspace, installing inline files in the *Inline* folder of the active workspace is mandatory.

The purpose of the workspace is to collect all scripts, inline files and templates in one single folder. Should the need occur to keep files in a different location, for example a network drive for automated backups, please consider using CSharpCalc in local mode on that drive.

The CSharpCalc user interface

Introduction

As mentioned before, CSharpCalc uses a strictly linear mode of operation:

1. Write code
2. Execute code
3. View results

If required, you may to go back to 1, apply appropriate modifications to the script and repeat the procedure.

There is no mechanism for interaction with the software while a calculation or simulation is running. This mode of operation is reflected in the user interface (UI) which consists of two parts. The first part of the UI is the code editor. Here you write C# script code, compile that code and receive information on compiler errors. The code editor is a state of the art C# editor with syntax highlighting, line numbering and collapsing language constructs.

Once you are done editing your script you may execute it. This is done by pushing the *Run* button in the center of the menu bar. When you push this button, CSharpCalc automatically compiles the code in the editor and, if the compilation is successful, begins execution. During the code execution phase, the user interface will switch to the runtime interface. This interface displays the output generated by your scripts. It has only three buttons which look like the controls of a media player. The button labeled *Edit* takes you back to the code editor. The stop button terminates the execution of the currently running code. The play button allows you to re-run the current script.

Hence, switching between the code editor and the runtime interface is simply done by pushing the *Run* and *Edit* buttons in their respective interfaces.

Try this ..

When the editor window does not contain any code it contains a so-called empty script. This counts as valid source code and, unsurprisingly, it does nothing. Push the *Run* button when the editor contains no code to bring up the runtime interface. Once there, push the *Edit* button in order to return to the code editor.

When back in the code editor push the *Compile* button and see a *Compilation successful* message in the compiler messages below the code editor.

The code editor

The code editor is a lightweight but state of the art C# text editor featuring syntax highlighting and collapsible language constructs. Moreover it offers multiple undo/redo steps and dynamic (with respect to collapsed code) line numbering. Most editor commands can be invoked using keyboard shortcuts such as ctrl-c or ctrl-v. In addition the button bar at the top margin of the window offers shortcut buttons for the most frequently used editor commands. A full list of commands is available from the *File* and *Edit* menus.

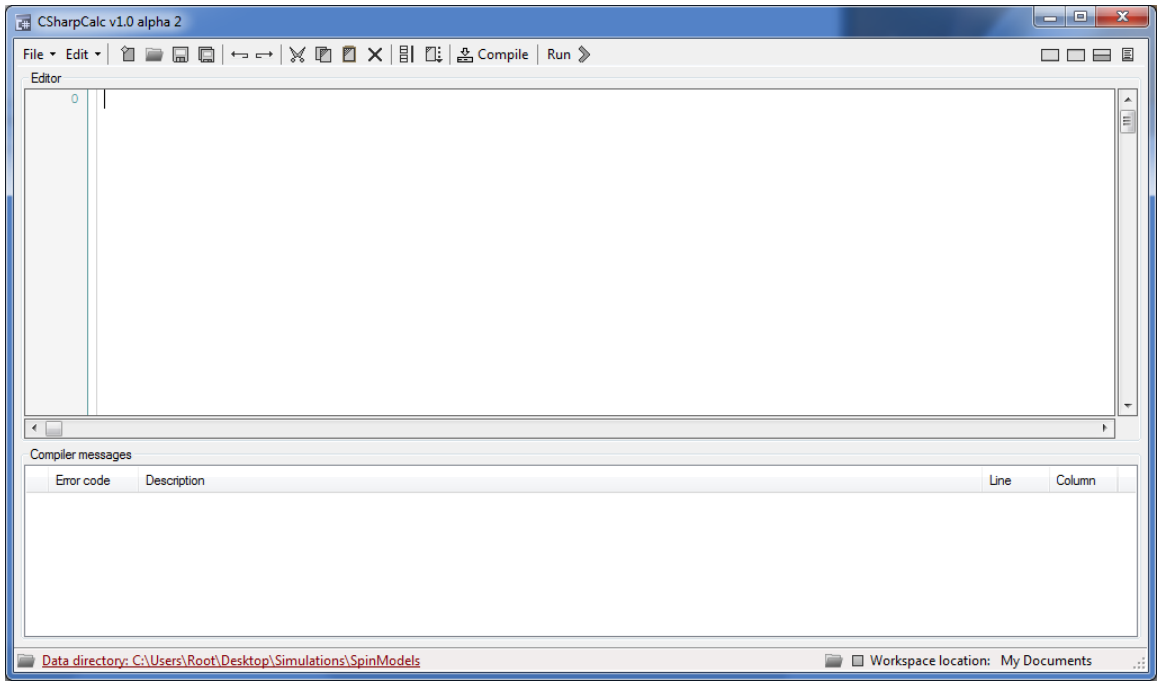

Figure 1. The CSharpCalc code editor interface. From top to bottom: The menu and button bar, the code editor, the compiler messages and the status line.

The rightmost part of the button bar contains four buttons which offer the functionality for changing the views in the editor window. You may choose to display only the editor, only the compiler messages (useful when the list gets crowded) or both. The fourth button brings up the *Browser* which will be described below.

The File menu commands

New: Initializes a new (empty) script. When you start CSharpCalc a new script is initialized by default. If the editor contains unsaved text you will be asked for confirmation to discard this text before re-initialization.

Open: Loads a script from a .csc file. The extension .csc is given to all CSharpCalc scripts by default. The upcoming open file dialog is set to the *Source* folder in the current workspace as default folder.

Save: Saves the code contained in the editor. If the script has not been saved to a file before, this function acts like the Save as command.

Save as: Saves the code in the editor to a new .csc file. By default the *Source* folder in the current workspace is offered as storage location. The file dialog provides the option to create subfolders for better structuring of your .csc script library.

Export classes: Exports the classes contained in the currently edited source code to the Inline folder of the active workspace. Its functionality will become clear with the introduction of inline classes. This function is specific to CSharpCalc.

Online info: This function opens the CSharpCalc web site in your default browser for latest information and news. A connection to the Internet is required.

License: Displays the license information. This dialog also offers an option to print a hard copy of the license.

Quit: Terminates CSharpCalc

The Edit menu commands

Undo: Goes one step back in the history of text changes

Redo: Goes one step forward in the history of text changes. Undo and redo fully resemble the usual commit/rollback mechanism available in most modern text editors.

Cut: Copies the selected text to the clipboard and removes it from the editor.

Copy: Copies the selected text to the clipboard without removing it.

Paste: If the clipboard contains text this function inserts that text at the current cursor position. If no text is available from the clipboard this function has no effect.

Delete: Deletes the selected text. If no text is selected this function deletes the character at the current cursor position.

Select all: Extends the selection to the entire text contained in the text editor.

CSharpCalc specific Edit menu commands:

Copy visible text: This function makes a copy of the currently visible text and places this text copy on the clipboard. Collapsed text will not be copied. This function is useful for making overviews of class interfaces when used in conjunction with the collapse all function.

Collapse all: This function collapses all second level collapsible text blocks. When applied to classes within the classcode context (see below) this function has the effect of collapsing all methods.

Insert template: This function allows for the selection of a .csc file from the *Templates* folder in the current workspace. Upon confirmation, the text contained within the selected file will be inserted at the current cursor position. This function facilitates the reuse of code snippets. The contents of a template is not required to compile as a standalone code. It may even contain a standardized comment or a documentation template.

The shortcut buttons

The buttons located to the right of the menus are shortcuts for menu commands. The only two button-triggered commands not contained in the menus are:

Compile: Compiles the currently edited script within the CSharpCalc framework. Use this function for syntax checking. Before the execution of a script CSharpCalc enforces compilation of the code.

Run: Forces compilation of the currently edited script and, upon successful compilation, executes the script within the CSharpCalc environment. For execution and viewing of the calculation results the user interface automatically switches to the runtime interface. Press the *Edit* button in the runtime interface in order to return to the code editor. In the case of compiler errors, the command does not execute the script. Instead a list of compiler errors is displayed below the editing window.

The view buttons

On the right margin of the menu bar there are four buttons. The first three of these allows you to control the views. By default, both the editor window and the compiler messages are visible. These buttons let you hide either the compiler messages or the editor. Like most buttons in CSharpCalc, these buttons display tooltips explaining their function when you hover the mouse pointer above them. The last button in the row brings up the *Browser* which is explained below.

The editor

The actual code editor works like a common text editor with three major differences.

- It performs syntax highlighting specific to the C# language. This is a convenience feature which significantly increases the readability of the source code.
- It shows line numbers in a separate area on the left side of the editor window. This helps finding specific lines in the case of compilation errors.

- It allows for collapsing and expanding certain code constructs which is convenient for maintaining the overview in long scripts. Whenever there is a construct that is enclosed between curly brackets { and } the editor will display a small rectangle on the left side of the text (or the right side of the line number whichever you prefer). Click on this rectangle in order to expand and collapse the text enclosed between { and }.

Generally speaking, the CSharpCalc editor offers state of the art code editing functionality needed for writing scripts without the complexity of an integrated development environment.

Try this ..

Start CSharpCalc and enter the following text with all newlines into the editor:

```
test
{
 hello
}
```

Once the parentheses are closed, the rectangle on the left side will appear. Click on that rectangle in order to collapse the text. Click again in order to re-expand the text.

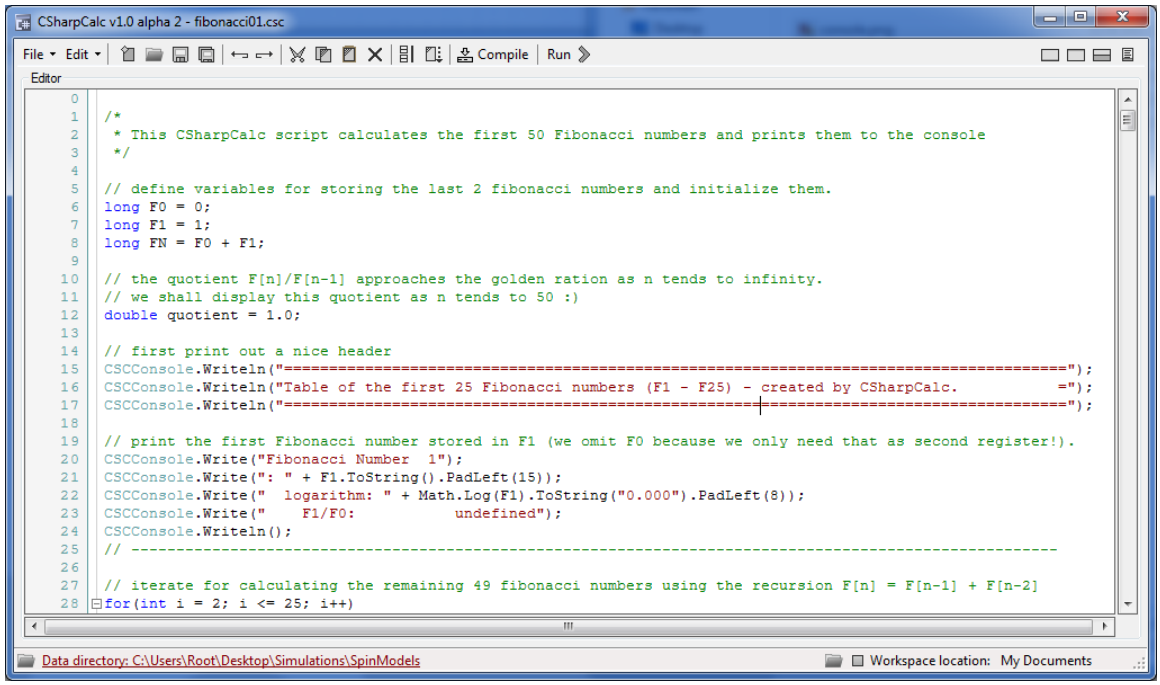

Figure 2. View of the code editor with highlighted syntax and line numbering

Collapsed text can be nested, i.e. it is possible to have collapsed text within collapsed text and so on. Whenever you implement classes in the CSharpCalc text editor, the *Collapse all* function described above can be used for collapsing all class methods in one go.

The status line

The lower border of the CSharpCalc window contains the status line which serves for choosing the locations of the current data directory and workspace.

The data directory

The data directory is a folder on your computer which can be accessed from within a CSharpCalc script for persistently storing calculation results in files. To be more precise, it is the only folder on your computer that can be accessed from within a CSharpCalc script. This helps maintain a certain level of security. This is not, however, a full-fledged security concept and you should never execute code coming from third parties without thoroughly examining it prior to execution. Clicking on the link Data directory in the status bar allows you to select a folder to be the current data directory. It is strongly recommended to specify this directory before using CSharpCalc. A click on the open folder icon to the left of the Data directory link opens the active data directory in the windows file manager.

The workspace

The function of the workspace has been explained above. In order to change its location click on the menu button labeled *Workspace location* on the right side of the status line. The label to the right of this menu button shows the current workspace location, in the above screenshots this is the My Documents folder. In order to open the workspace directory in a windows file manager, click on the folder icon to the left of the menu button.

The runtime interface

The runtime interface comes up whenever you execute a script within CSharpCalc. It provides windows for text output and for graphical output as well as a minimalistic set of functions similar to a media player.

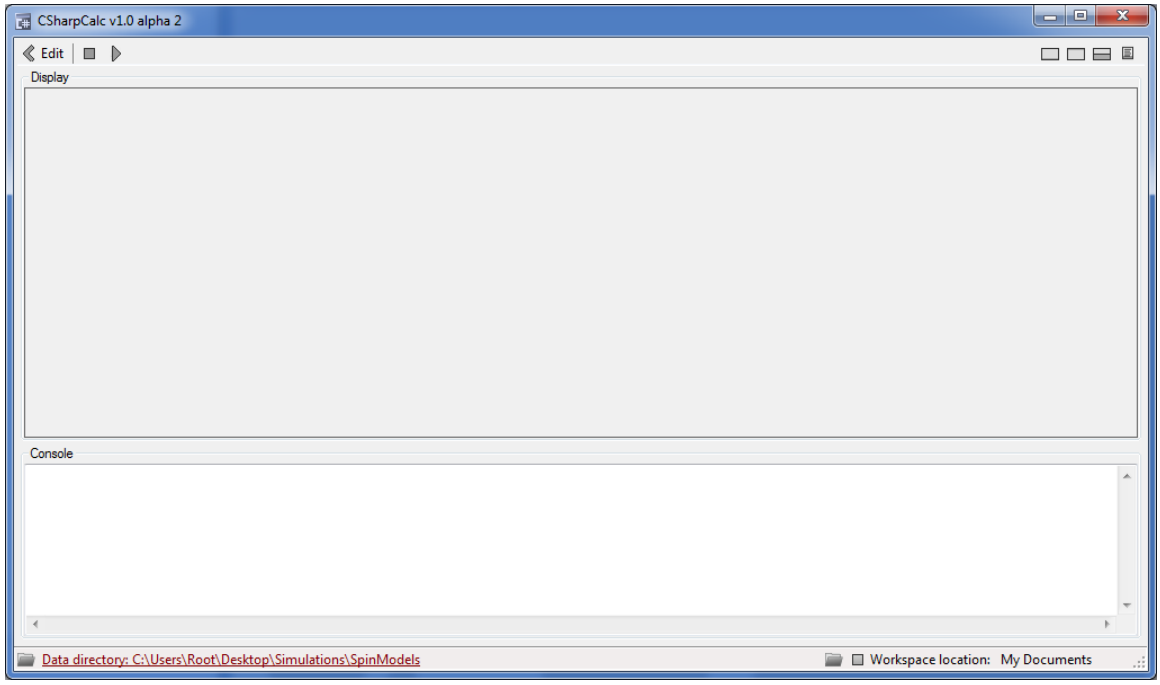

Figure 3. The CSharpCalc runtime interface. From top to bottom: The button bar, the graphical display, the console for displaying text output and the status line.

The left side of the menu bar offers only three functions. The Edit button takes you back to the code editor. The stop button terminates execution of the currently running script. Finally, the play button restarts execution of the current script. Please keep in mind that poorly initialized code might exhibit unwanted and/or unexpected side effects when restarted.

The right side buttons in the menu bar hide and show the display and the console. Like the corresponding buttons in the menu bar of the code editor these buttons have tooltips explaining their respective function when hovering over them with the mouse pointer.

The display

The display serves as a virtual graphical output device. It displays graphics drawn within your scripts. CSharpCalc offers specialized classes for initializing and using the display which will be described in the subsequent sections of this manual dedicated to graphical output.

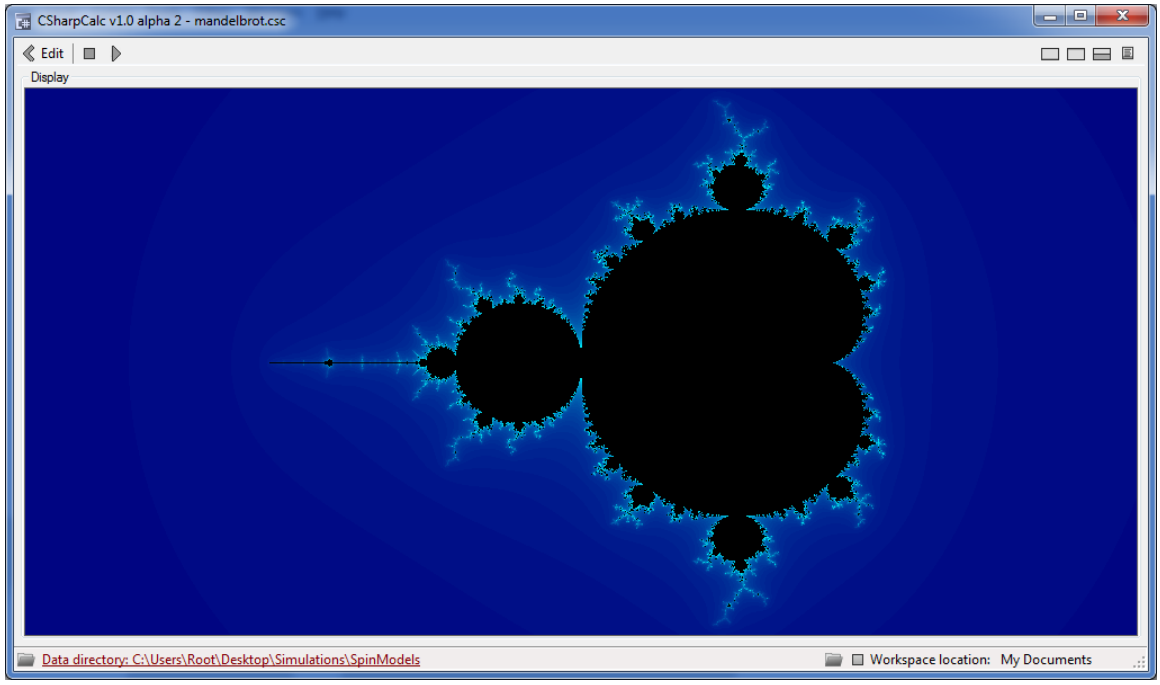

Figure 4. View of the display showing a simulated Mandelbrot set. Rendering this image with CSharpCalc takes less than one second on a regular off the shelf PC.

In order to keep persistent copies of the images rendered to the display, the class *CSCDataIO* (see below) offers a method named *WriteSurface()* for saving snapshots as .png images in your active working directory. Alternatively a right click on the display with the mouse allows you to save the displayed image as .png file in a directory of your choice.

The console

While the display provides a graphical surface for drawing, the console is used for text output. Text will be used for printing numerical calculation results as well as for printing information on a script's execution progress in long-running calculations. The class *CSCDataIO* (see below) also

offers a method for storing the text contents of the console to a file for persistent storage of your calculation results.

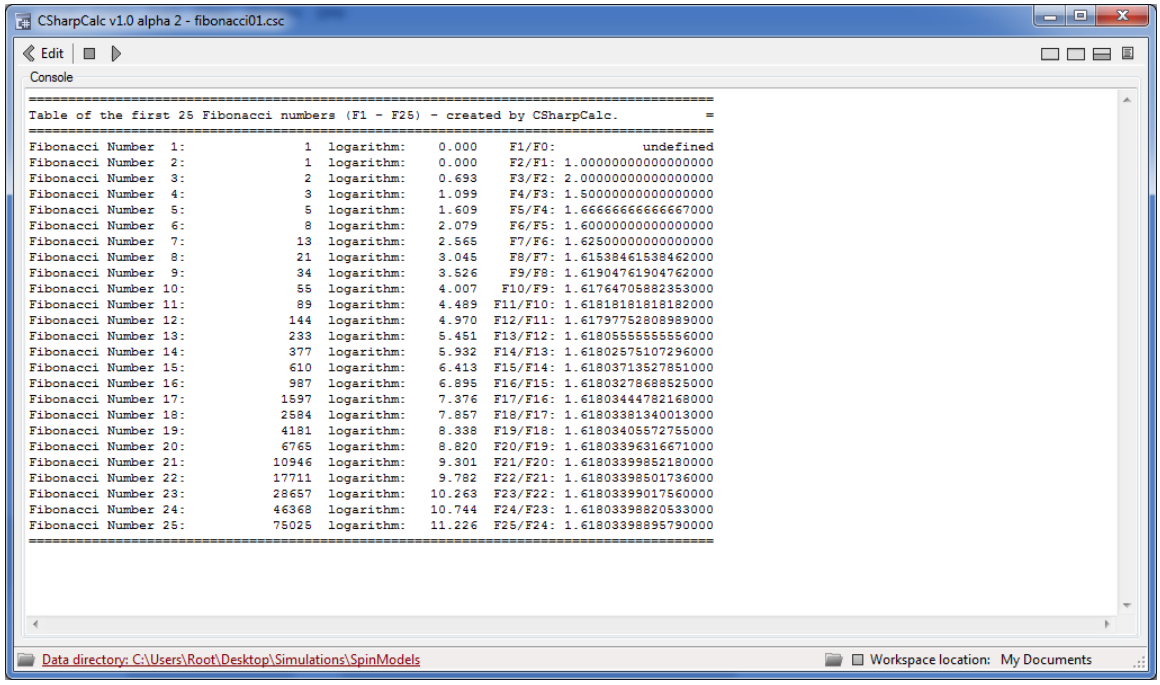

Figure 5. The CSharpCalc console displaying formatted text output. This view displays the first 25 Fibonacci numbers along with the sequence of ratios which converges to the golden ratio.

Try this ..

Copy the line:

```
CSCConsole.Writeln("Hello world of CSharpCalc");
```

into the editor. This single line can be compiled and executed without further ado. Proceed as follows.

- Save the script. Use the *Save as..* command in the *File* menu and save the script to a file named hello.csc in the *Source* folder of your active workspace. You will use this saved script later for experimenting with the browser (see below).
- Execute the script by pressing the *Run* button in the Code Editor's button bar. This will bring up the runtime interface. Switch to console only view. The console should contain the text: *Hello world of CSharpCalc.* Switch to display only view. The display should be gray and without contents. Finally, bring up the split view showing display and console again.
- Return to the editor using the *Edit* button. Switch to code only view. This view is the preferred one for editing code. Upon compilation the compiler message view automatically becomes visible if its hidden.

Using the *Run* and *Edit* button in combination with the view buttons allows for navigating freely through the different views of CSharpCalc.

The Browser

Pushing the rightmost button in the menu bar brings up the CSharpCalc browser which serves two distinct purposes.

The browser provides a fast way to look up interfaces of the internal CSharpCalc classes. In order to browse an interface you need to choose the class you are interested in from the class menu above the browser's text display. The class selection menu is verbosely labeled and self explanatory.

The browser also allows for loading and displaying script files for quick cut and paste. The starting folder for the file selection is the *Source* folder in the active workspace. By default only .csc files will be shown.

Both the class interfaces and the contents of loaded .csc files are written to the text display. If several commands are invoked, the text outputs of these calls will be printed in consecutive order without erasing the output of previous commands. In order to clear the text display you need to push the *Clear* button in the upper right corner of the window.

Try this ..

- Load the file hello.csc you saved in your workspace earlier into the browser. Push the File... button in the upper part of the browser window, select the file and press OK. The file content will be displayed in the text display
- Copy the file contents into the code editor using the regular copy/paste shortcuts i.e. ctrl-C and ctrl-V. You can now execute a copy of the script. In realistic applications one will typically use this feature for replicating small code snippets from other scripts.
- Clear the browser window by pressing the Clear button in the upper right corner of the browser window.
- Choose a class (e.g. *CSCDataIO*) from the class interface menu above the text display and get an impression of the class interface documentation.

This concludes our presentation of the CSharpCalc user interface. Now that you know the software's functions and views we shall proceed to the implementation of calculations and simulations.

Working with CSharpCalc

Working with CSharpCalc means to write scripts which do mathematical calculations and/or simulations. Although a .csc script is not as complicated as a full-fledged C# application for MS Windows, it needs to interact with the CSharpCalc framework and it is required to obey a few, simple, rules in order to function properly. This section of the manual explains the CSharpCalc framework and gives examples how to use the built in classes in order to make use of the display and the console. As a first example we will develop a script which calculates the first 50

Fibonacci numbers, prints properly formatted results to the console and exhibits the convergence of the ratio of subsequent numbers towards the golden ratio.

Writing a CSharpCalc script

Simple .csc scripts consist of a linear sequence of commands without function declarations or class definitions. More complex calculations may utilize user defined classes for better structuring. This will be described in the subsequent sections. For now, we shall develop a simple version of our Fibonacci script without classes. The first step, when starting a new project with CSharpCalc is a proper setup. Please, proceed as follows...

Specify a data directory for the Fibonacci project. Click on the data directory link in the status line of CSharpCalc and create a folder named Fibonacci in a location of your convenience e.g. the My Documents folder in your user profile. CSharpCalc will save all files it generates to this directory.

Start a new project using the *New* function in the file menu or, alternatively, just start CSharpCalc.

Enter this code (a single line comment) into the editor:

```
// This CSharpCalc script calculates the first 50 Fibonacci numbers
```

Save the new script to a folder in your active workspace using the *Save as..* function. The starting location of the upcoming dialog is the *Source* folder in your active workspace. Create a new directory named Manual navigate into this folder and store the script in a file named Fibonacci.csc.

Now that the project setup is complete we can begin to develop our script. The mathematical basics of the Fibonacci sequence {F(n); n=0, 1, 2, ..} are quite simple. The sequence is defined recursively by:

F(0) = 0, F(1) = 1; F(n) = F(n-1) + F(n-2)

In plain words, The nth Fibonacci number is the sum of the two preceding numbers. Let us add this sequence initialization to our script. Since Fibonacci numbers become quite large as n increases we chose *long* as variable type for the registers X, Y and F (see code below). Add the following code to the script below the comment we started with:

```
// define variables for storing the last 2 Fibonacci numbers and
// initialize them.
long X = 0;
long Y = 1;
long F = X + Y;
```

In addition to the actual Fibonacci numbers we want to calculate the quotient of two subsequent Fibonacci numbers as the infinity limit of this ratio is the so-called golden ratio. Hence, we need a variable for the quotient. Add this code to your script.

```
// the quotient: F[n]/F[n-1] approaches the golden ration as n tends
// to infinity. we shall display this quotient as n tends to 50 :)
double quotient = 1.0;
```

Now that we have defined all required variables, we may start generating formatted text output. In order to write text to the CSharpCalc Console we need to use the class *CSCConsole*. This class is part of the CSharpCalc framework and it provides a small but powerful set of functions. For now we will use *Write()* and *Writeln()*. *Write()* adds the text in the argument to the text in the console while *Writeln()* (read: write line) adds the text in its argument plus a newline. At this point you might want to bring up the browser and read up on the few remaining functions of the class *CSCConsole*. Consult the documentation of the browser above for a revision of how to do this.

In order to produce nicely formatted output of our Fibonacci sequence, we begin with printing a header. Add the following lines to your code:

```
// first print out a nice header
CSCConsole.Writeln("================================================");
CSCConsole.Writeln("Table of the first 50 Fibonacci numbers (F1-F50)");
CSCConsole.Writeln("================================================");
```

At this point the script already produces text output. Compile it and run it for the first time. You should see the above header printed to the console. Return to the editor to proceed with script development.

Since the first two Fibonacci numbers are fixed we omit F[0] and print F[1] separately. Add the following code to your script:

```
// print the first Fibonacci number stored in F1
CSCConsole.Write("F01: " + Y.ToString().PadLeft(15));
CSCConsole.Write("  F01/F00:  undefined");
CSCConsole.Writeln();
```

The text formatting methods (*ToString(..)*, *PadLeft(..)*) we are using here are part of the C# language. Any basic tutorial of C# should cover these and we need to refer to the C# language documentation for an explanation.

After printing the first member of the sequence we can now compute and print the remaining Fibonacci numbers F2-F50 recursively in a simple *for* loop. Add the following code to your script:

```
// iterate for calculating the remaining 49 Fibonacci numbers using the
// recursion F[n] = F[n-1] + F[n-2]. Initially the register F contains F2!
for(int i = 2; i <= 50; i++)
{
 // print the current Fibonacci number stored in F
 CSCConsole.Write("F" + i.ToString("00").PadLeft(2));
 CSCConsole.Write(": " + F.ToString().PadLeft(15));
 string qlabel = " F" + i.ToString("00") + "/F" + (i-1).ToString("00") + ": ";
```

```
 CSCConsole.Write(qlabel.PadLeft(11) + quotient.ToString("0.00000000"));
 CSCConsole.Writeln();

 // recursively calculate the next Fibonacci number and the next quotient
 X = Y; Y = F; F = X + Y; quotient = (double)F/(double)Y;
}
```

At this point our script is almost completed. In order to obtain a perfect printout we add a final ruler at the end of the list. In addition we store the output printed to the console in a file named fibonacci.txt. This file will be placed in your active data directory. Add the following code to your script.

```
// print a terminating horizontal ruler
CSCConsole.Writeln("================================================");

// finally store the console text in a file named fibonacci.txt in your
// active data directory
CSCConsole.SaveText("fibonacci.txt");
```

Now that the script is complete you can compile and run it. The complete Fibonacci script is printed below for your convenience:

```
// This CSharpCalc script calculates the first 50 Fibonacci numbers define
// variables for storing the last 2 Fibonacci numbers and initialize them.
long X = 0;
long Y = 1;
long F = X + Y;

// the quotient: F[n]/F[n-1] approaches the golden ration as n
// goes to infinity. We shall display this quotient as n tends to 50 :)
double quotient = 1.0;

// first print out a nice header
CSCConsole.Writeln("================================================");
CSCConsole.Writeln("Table of the first 50 Fibonacci numbers (F1-F50)");
CSCConsole.Writeln("================================================");


// print the first Fibonacci number stored in F1

CSCConsole.Write("F01: " + Y.ToString().PadLeft(15));
CSCConsole.Write("  F01/F00:  undefined");
CSCConsole.Writeln();

// iterate for calculating the remaining 49 Fibonacci numbers using the
// recursion F[n] = F[n-1] + F[n-2] Initially the register F contains F2!

for(int i = 2; i <= 50; i++)
{
 // print the current Fibonacci number stored in FN
 CSCConsole.Write("F" + i.ToString("00").PadLeft(2));
 CSCConsole.Write(": " + F.ToString().PadLeft(15));
 string qlabel = " F" + i.ToString("00") + "/F" + (i-1).ToString("00") + ": ";
 CSCConsole.Write(qlabel.PadLeft(11) + quotient.ToString("0.00000000"));
 CSCConsole.Writeln();
```

```
 // recursively calculate the next Fibonacci number and the next quotient.
X = Y; Y = F; F = X + Y; quotient = (double)F/(double)Y;
}
// print a terminating horizontal ruler

CSCConsole.Writeln("================================================");

// finally store the text displayed in a file named fibonacci.txt in your
// active data directory
CSCConsole.SaveText("fibonacci.txt");
```

Bring up the console only view in the runtime interface for a good view on the text output.

User defined classes

Considering the simplicity of the task, the Fibonacci script does not require further structuring. More complex applications, however, will require a more structured code design. Hence, CSharpCalc offers the possibility to implement user defined classes. Classes can be added to special sections of a script which are labeled by the

```
/* <classcode> */
```

statement. In order to distinguish class code from code that is immediately executable the latter kind of code should be put into sections labeled

```
/* <maincode> */
```

If no classes are used, as it was the case in the above Fibonacci example, the *<maincode>* statement can be omitted. *<maincode>* and *<classcode>* serve as context flags which switch between the two contexts. As a rule of thumb, we recommend that you start with the main code and append classcode below it.

Below you find a re-written Fibonacci script which encapsulates the calculation of the sequence and the generation of formatted text output in a static class named Fibonacci. The script starts with its main code as indicated by the *<maincode>* flag, which is not needed since the maincode context is the default context, but shown here for completeness. This short maincode sequence illustrates the proper use of the Fibonacci class interface. Subsequently, the script is switching to the *<classcode>* context for the definition of the Fibonacci class. The mathematical calculations are mirroring the computations of the previous version of this script.

```
/* <maincode> */
// initialize the Fibonacci class
Fibonacci.Initialize(0, 1);

// compute the sequence and the ratios
Fibonacci.Calculate(50);

// write the results to the console
CSCConsole.Writeln(Fibonacci.TextOutput());

// finally store the console text
```

```
CSCConsole.SaveText("fibonacci.txt");

/* <classcode> */
public static class Fibonacci
{
 public static void Initialize(int f0, int f1)
 {
  _X = (long)f0;
  _Y = (long)f1;
 }

 // this method calculates the first n members of the sequence
 // and stores them along with the ratios in the F and R lists.
 public static void Calculate(int n)
 {
  F.Add(_X); R.Add(0.0);
  F.Add(_Y); R.Add(0.0);
  long f = _X + _Y;
  double r = ((double)f) / ((double)_Y);
  for(int i = 2; i <= n; i++)
  {
   F.Add(f); R.Add(r);
   _X = _Y; _Y = f; f = _X + _Y;
   r = ((double)f) / ((double)_Y);
  }
 }

 // this method returns formatted text output
 public static string TextOutput()
 {
  int N = F.Count - 1;
  StringBuilder SB = new StringBuilder();
  SB.Append("================================================\r\n");
  SB.Append("The first " + N.ToString() + " Fibonacci numbers\r\n");
  SB.Append("================================================\r\n");
  for(int i = 1; i <= N; i++)
  {
   SB.Append("F" + i.ToString().PadLeft(2));
   SB.Append(": " + F[i].ToString().PadLeft(15));
   string qlabel = " F" + i.ToString() + "/F" + (i-1).ToString() + ": ";
   SB.Append(qlabel.PadLeft(18) + R[i].ToString("0.00000000"));
   SB.Append("\r\n");
  }
  SB.Append("================================================");
  return SB.ToString();
 }

 // public accessors
 public static List<long> F { get { return _f; } }
 public static List<double> R { get { return _r; } }

 // private variables
 private static long _X = 0;
 private static long _Y = 1;

 // two lists for storing the sequence and the ratios
 private static List<long>      _f = new List<long>();
```

```
 private static List<double>     _r = new List<double>();
}
```

There are a few rules which apply to the classcode context

- The classcode context may contain code for as many classes as you need.
- There must not be any language statements outside of a class context within classcode.
- Classcode may contain both static and non-static classes.

Generally, we recommend structuring of scripts into classcode, which does the actual computations, and maincode, which uses these classes with different parameters.

Inline classes

The Fibonacci class is quite short and simple. In more complicated applications a larger number of more sophisticated classes will be used for modeling the problem under consideration and in such cases the script's size may well exceed several thousand lines of code. For even better structuring CSharpCalc offers the functionality to replace the full class code with an inline placeholder. The actual class code is stored in a separate file which will not be displayed in the editor. Instead, the full class code is replaced by the following placeholder directive:

```
//inline <relative path to the file containing the class>
```

The inline path is understood to be a relative path with respect to the Inline folder in your active workspace. As an example we will restructure the Fibonacci example we already have. Please follow this procedure step by step as the correct sequence of actions is essential.

Load the Fibonacci script from the previous section which contains the Fibonacci class (not the first version of the script which does not contain a class).

Save the script under a new name using the *Save as..* command in the *File* menu. This makes sure that the original script will not accidentally be overwritten.

Use the *Export classes..* function in the *File* menu in order to export the Fibonacci class to an inline file. By default the Inline folder in your active workspace will be offered as location. Create a subfolder named Manual navigate into that folder and save the class in a file named Fibonacci.csc. In order to be able to use an inline file it must be located in the Inline folder of the active workspace! Subfolders are allowed. Now delete the script code you have and replace it by the code displayed below.

```
// An inline version of the Fibonacci script
// inline the Fibonacci class
//inline Manual\Fibonacci.csc

// and use it
Fibonacci.Initialize(0, 1);
Fibonacci.Calculate(50);
CSCConsole.Writeln(Fibonacci.TextOutput());
// ********************************************************
```

The statement `//inline Manual\Fibonacci.csc` instructs the compiler to add the code contained in the specified file to the list of class declarations before compiling the script. Please note that there must not be any whitespace between // and inline because `//inline` is one statement. Run this script and you will see the same output as before. If you find Fibonacci numbers entertaining you may vary the initialization of the sequence, i.e. replace 0,1 in the *Initialize* method by some other numbers and experiment with the results. Hint: Observe the value of the ratio as you modify the initial numbers and find an interesting fact about the Fibonacci numbers and their connection to the golden ratio.

Inline classes are useful in complicated simulations and calculations. Moreover, the inline mechanism described above allows for the implementation of user defined data structures which is often required. Inline classes are in no way different to classes contained in the script, they only reside in separate files. Hence, all rules for the implementation of classes given in the previous section also apply to inline classes.

Graphical output

While the console is the device for displaying text, the display provides the functionality needed for drawing graphics. Graphics in CSharpCalc is slightly different than graphics for e.g. a computer game or a presentation slideshow which uses pixel or dtp units for the specification of graphical primitives. CSharpCalc requires you to specify your own system of units natural to the computations under consideration, for example micrometers or megaparsec. The size of the graphical surface measured in these custom units is called the physical size. In addition, you need to specify the size of the canvas in pixels, i.e. the pixel size of the surface. The class *CSCDisplay* provides the functionality for doing this. The script below draws a simple picture.

```
// specify the size of the surface (i.e. the image size) in pixels
CSCDisplay.SetSurfaceSize(600, 450);

// also specify the size of the display in physical units.
// This call specifies a rectangle by its corner points (-1, -1) and (1, 1).
CSCDisplay.SetPhysicalSize(-1.0, -1.0, 1.0, 1.0);

// Initialize the surface with white as background color.
CSCDisplay.Clear(Color.White);

// Begin the rendering process
CSCDisplay.BeginRendering();

// Render a few simple primitives. Geometry is defined in physical units.
CSCRender2D.DrawRectangle(-0.5, -0.5, 0.25);
CSCRender2D.FillRectangle(0.0, 0.0, 0.15);
CSCRender2D.DrawRectangle(0.5, -0.5, 0.25);

// Finish the rendering process
CSCDisplay.EndRendering();

// Present the surface in the CSharpCalc display
CSCDisplay.Present();
```

Copy this code into the CSharpCalc editor and execute the script. You should see a 600x450 pixel image (*SetSurfaceSize*). The physical size of the surface has been specified to range from the lower left corner (-1, -1) to the upper right corner (1, 1) using *SetPhysicalSize*. Three squares are drawn using *DrawRectangle* and *FillRectangle* with their positions and extensions given in these physical units. The result is this picture:

Figure 6. Simple graphical output in CSharpCalc

For rendering this image we used two classes, *CSCDisplay* for initializing the display and *CSCRender2D* for drawing graphical primitives. The method *CSCDisplay.Present()* is called for displaying the surface in the display window. Alternatively, a rendered image can also be written to a .png file using the class *CSCDataIO*.

Graphical primitives are rendered between *BeginRendering()* and *EndRendering()*. Contrariwise, all initializations like *SetSurfaceSize()* or *Clear()* must be called outside of this rendering context. The logic behind this closely follows the logic used when painting a real picture. You first choose a canvas of the required size and mount it on an easel. This corresponds to the initialization sequence. Then you begin painting. While painting you, paint one object after the other. This corresponds to drawing graphical primitives between *BeginRendering()* and *EndRendering().* Once the painting phase is finished you find a place to put your newly created picture. This corresponds to presenting the surface or storing it in a file.

An exhaustive list of supported graphical primitives can be found in the interface documentation of the class *CSCRender2D*. It does not differ significantly from other graphics packages and shall not be discussed here in further detail. We trust that CSharpCalc users will be capable of finding their way through these definitions on their own.

Direct pixel manipulations

The class *CSCDisplay* encapsulates the initialization and presentation methods already described. In addition, this class also provides methods for directly accessing the pixels of the surface. In order to give an example for using these methods we present the following script which draws a Mandelbrot set in fancy colors.

```
// Mandelbrot set simulation

// Set the active colormap. Alternatives are available and documented
// in the CSCColorMap class interface.
CSCColorMap.SetTo(CSCColorMapType.RedToYellow);

// Increase the iteration depths for eventual close ups
Mandelbrot.IterationDepth = 50;

// Initialize console and display
CSCConsole.Clear();
CSCDisplay.SetSurfaceSize(600, 450);

// set the physical size to the rectangle enclosing the Mandelbrot set.
CSCDisplay.SetPhysicalSize(-2.0, -1.2, 0.5, 1.2);
CSCDisplay.Clear(Color.Black);

// render the Mandelbrot set
CSCDisplay.BeginRendering();

for(int j = 0; j < CSCDisplay.PixelHeight; j++)
{
 for(int i = 0; i < CSCDisplay.PixelWidth; i++)
 {
  // Use XFromPixel and YFromPixel to find the physical position of a pixel
  double x = CSCDisplay.XFromPixel(i);
  double y = CSCDisplay.YFromPixel(j);
  Color c = Color.Black;
  int speed = Mandelbrot.DivergenceSpeed(x, y);
  if(speed > 0) { c = CSCColorMap.GetColor(speed); }
  CSCDisplay.SetPixel(i, j, c);
 }

 // Report the rendering progress to the console every 100 lines.
 if(j > 0 && j % 100 == 0) { CSCConsole.Writeln("Lines: " + j.ToString()); }
}

CSCConsole.Writeln("Lines: " + CSCDisplay.PixelHeight.ToString());
CSCDisplay.EndRendering();
CSCDisplay.Present();
CSCDataIO.WriteSurface();
CSCConsole.Writeln("Mandelbrot finished");

/* <classcode> */
// we need a function for measuring the speed of divergence...
public static class Mandelbrot
{
 public static int DivergenceSpeed(double x, double y)
 {
```

```
  // Use the build-in complex number class CSCComplex for calculations
  CSCComplex c = new CSCComplex(x, y);
  CSCComplex z1 = new CSCComplex(0.0, 0.0);
  CSCComplex z2 = new CSCComplex(0.0, 0.0);
  for(int i = 0; i < IterationDepth; i++)
  {
   z2 = z1*z1 + c;
   z1 = z2;
   if(z1.R > 4) { return (int)(i*255.0/IterationDepth); }
  }
  return 0;
 }
 public static int IterationDepth = 1;
}
```

This script does not draw primitives, but instead sets pixels to color values using the method *CSCDisplay.SetPixel(..)*. In order to determine the physical coordinates pertaining to a particular pixel *CSCDisplay* offers the methods *XFromPixel* and *YFromPixel*. The Mandelbrot class does the actual calculations which make use of the class *CSCComplex* representing complex numbers in CSharpCalc (see below). Finally the class *CSCColormap* is used for mapping values of the divergence speed to colors. In order to find out more details on classes you are interested in, please use the browser which will display a detailed class interface documentation for all of these classes.

After running the above script you should see this output image:

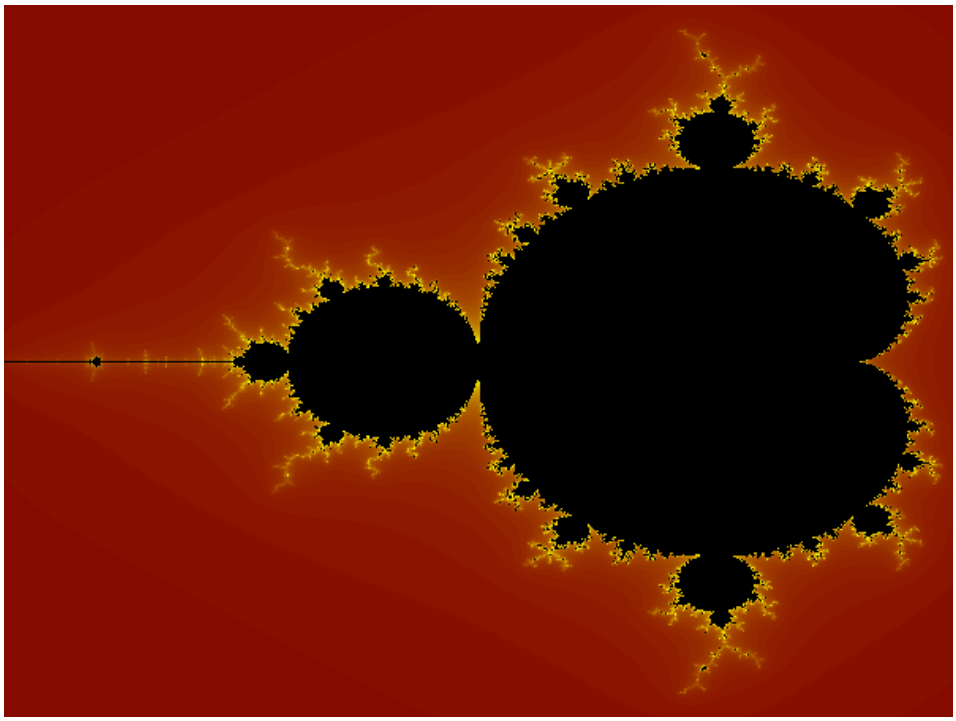

Figure 7.The Mandelbrot set is a fractal set first described by Benoit B. Mandelbrot

Try this:

- Expand the rendering surface to poster size, e.g. 1920x1080 pixels. Add a command that writes the resulting image to a file so you and your friends can use it as background image. Find the appropriate function in the interface documentations in the browser.
- Bring up the interface of *CSCColormap* in the browser and experiment with different color maps. See anything you like? Then use that one instead.
- Modify the physical extensions of the surface and render some close-ups of interesting areas. Start out with a small canvas such as 600x450 pixels. Once you found an interesting window render a large version of it.

Plotting functions and trajectories

Many math application involve drawing function plots within a coordinate system. CSharpCalc supports the visualization of functions and trajectories by supplying the class *CSCFunctionPlot.* The class accepts instances of *CSCRealFunction* and supports plotting multiple function graphs in different drawing styles. Moreover, CSCFunctionPlot also supports the creation of legends and customization of the coordinate frame used for drawing.

The following script draws a Gaussian distribution graph

```
// initialize console and display. the physical size will be specified by
// CSCFunctionPlot.Initialize() below this block and needs no
// specification here.
CSCConsole.Clear();
CSCDisplay.SetSurfaceSize(1000, 500);

// initialize the SCSFunctionPlot class. this version, without parameters
// adapts the physical size to the extensions of the first function plotted.
CSCFunctionPlot.Initialize();

// set a style for the function graph green color and 2.0 line width
CSCFunctionPlot.SetStyle(Color.Green, 2.0);

// plot the function graph. the Gaussian class is defined below
CSCFunctionPlot.Plot(new Gaussian(2.0, -2.0, 2.0, -10.0, 10.0, 150));

// and show the resulting rendering in the display
CSCDisplay.Present();

/* <classcode> */
public class Gaussian : CSCRealFunction
{
 public Gaussian(double A, double mu, double sigma, double xmin, double xmax,
int numsamples) : base(xmin, xmax)
 {
  _sigma = sigma;
  _mu = mu;
  _a = A;
  Resample(numsamples);
 }


 public override double Value(double x)
 {
```

```
  return   _a*(1.0/Math.Sqrt(2.0*Math.PI*_sigma))*Math.Exp(-(x   -   _mu)*(x   -
_mu)/(2.0*_sigma));
 }

 private double _sigma = 1.0;
 private double _mu = 0.0;
 private double _a = 1.0;
}
```

In the maincode part of this script the class *CSCFunctionPlot* initializes the display, specifies a style for plotting the function graph, plots a Gaussian function and presents the rendering result. Using these four lines of code is enough for plotting a function graph.

The classcode part of this script illustrates the definition of customized functions for the class *CSCRealFunction*. In order to define a custom function on the interval [XMin, XMax] one needs to override the virtual *Value(double x)* method. In our example the overridden *Value* returns the Gaussian probability distribution density of *x* with mean value *mu* and standard deviation *sigma*. In addition to the moments, one may also specify a total mass A. These parameters are specified in the constructor. Once all parameters are defined the constructor needs to create sample values between XMin and XMax. This is done by calling *Resample(numsamples)* which generates a new, equidistant array of sample points. The interval of definition [XMax, XMin] is specified in the constructor of the base class which is called using the `: base(xmin, xmax)` construct as illustrated.

Custom complex functions and vector functions can be implemented in a similar manner. In these cases the *Value(..)* virtual needs to return the appropriate types, i.e. a *CSCComplex* or one of the vector types.

The actual function plotting part of the above example script is exceedingly short as it mostly uses default behavior of the class *CSCFunctionPlot*. Without arguments, the *Initialize()* method adapts the physical extensions of the displayed coordinate frame to the extensions of the first function plotted automatically. Subsequent plots are drawn with respect to the first function's physical extensions.

A more sophisticated application is illustrated in the next script which plots Gaussian functions as delta sequence converging to the delta distribution. It employs varying plotting styles, letterings and labels and a legend for each of the plotted functions.

```
// set the display's surface size
CSCDisplay.SetSurfaceSize(1000, 500);

// initialize the pysical extensions of the coordinates. This is similar
// but not equal to calling CSCDisplay.SetPhysicalSize().
CSCFunctionPlot.Initialize(-8.0, 0.0, 8.0, 4.5);

// Set the margins of the coordinate frame in %
CSCFunctionPlot.SetMargins(10, 10, 50, 20);
// This version of SetStyle controls which parts of the coordinate frame
// will be plotted.
```

```
CSCFunctionPlot.SetStyle(true, true, true);

// Set styles for axes, the grid and the frame
CSCFunctionPlot.SetAxesStyle(Color.Black, 2.0);
CSCFunctionPlot.SetGridStyle(Color.Gray, 1.0);
CSCFunctionPlot.SetFrameStyle(Color.SlateGray, 1.0);

// Specify an offest for the legend text in pixels. Here, text will be
// displaced 2 pixels in the X direction and 10 pixels in the Y direction.
CSCFunctionPlot.SetLegendTextOffset(2, 10);

// plot the delta sequence
CSCFunctionPlot.SetStyle(Color.Black, 2.0);
CSCFunctionPlot.Plot(new Gaussian(1.0, 0.0, 0.01, -10.0, 10.0, 500));

// This method plots an entry in the legend
CSCFunctionPlot.PlotLegend(9.5, 0.6, "small sigma value (0.01)", 1.2);

CSCFunctionPlot.SetStyle(Color.Gray, 1.0);
CSCFunctionPlot.Plot(new Gaussian(1.0, 0.0, 0.05, -10.0, 10.0, 500));
CSCFunctionPlot.Plot(new Gaussian(1.0, 0.0, 0.1, -10.0, 10.0, 500));
CSCFunctionPlot.Plot(new Gaussian(1.0, 0.0, 0.25, -10.0, 10.0, 500));
CSCFunctionPlot.Plot(new Gaussian(1.0, 0.0, 0.5, -10.0, 10.0, 500));
CSCFunctionPlot.Plot(new Gaussian(1.0, 0.0, 1.0, -10.0, 10.0, 500));
CSCFunctionPlot.PlotLegend(9.5, 0.3, "intermediate sigma values", 1.2);

CSCFunctionPlot.SetStyle(Color.DarkRed, 2.0);
CSCFunctionPlot.Plot(new Gaussian(1.0, 0.0, 2.0, -10.0, 10.0, 500));
CSCFunctionPlot.PlotLegend(9.5, 0.0, "large sigma value (2.00);", 1.2);

// add own, custom features. in our case a headline
CSCDisplay.BeginRendering();
CSCRender2D.DrawCenteredText(0.0, 4.6, "Gaussian delta sequence", 0, 0);
CSCDisplay.EndRendering();

CSCDisplay.Present();

/* <classcode> */
public class Gaussian : CSCRealFunction
{
 public Gaussian(double A, double mu, double sigma, double xmin, double xmax,
int numsamples) : base(xmin, xmax)
 {
  _sigma = sigma;
  _mu = mu;
  _a = A;
  Resample(numsamples);
 }

 public override double Value(double x)
 {
  return   _a*(1.0/Math.Sqrt(2.0*Math.PI*_sigma))*Math.Exp(-(x   -   _mu)*(x   -
_mu)/(2.0*_sigma));
 }

 private double _sigma = 1.0;
 private double _mu = 0.0;
```

```
 private double _a = 1.0;
}
```

This script uses a different version of the *CSCFunctionPlot.Initialize()* method which sets the coordinate frame to a fixed size. Which version is best suited depends on the application under consideration. If you do not know the appropriate values you might want to use the value-less version for a first test. Once the appropriate values for the physical units are known the parameter version may be used for adapting the viewing area.

The *SetStyle(bool, bool, bool)* method is used for enabling and disabling the rendering of axes, frame and grid. In order to add a nice legend to the function plot we use *PlotLegend()*. Coordinates and lengths for this method are to be specified in physical units.

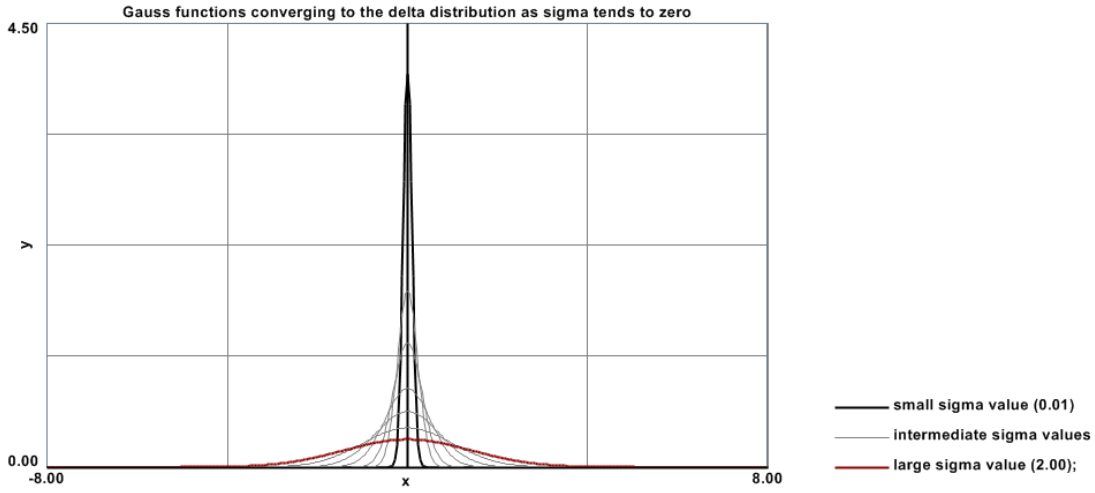

Figure 8. A visualization of Gaussians as delta sequence using the class CSCFunctionPlot

At the end of the script we use our own rendering block for adding custom features to the function plot. In this example we just add a headline. But generally, adding additional graphics such as tangents, thresholds, context aware letterings and similar may be added to the plot in this manner.

Parametric plots

In addition to function plots the class *CSCFunctionPlot* also offers a simple way to plot 2D parametric curves. In CSharpCalc Parametric curves are represented by (derived) instances of the type class CSCRealVectorFunction. This script defines a parametric function and renders it in a fully customized style. Style customizations are explained in the comments.

```
CSCDisplay.SetSurfaceSize(800, 600);
```

```
CSCFunctionPlot.Initialize();
CSCFunctionPlot.SetMargins(10, 10, 10, 10);

// Set a custom color for the background and the text
CSCFunctionPlot.SetBackground(Color.FromArgb(255, 0, 0, 0));
CSCFunctionPlot.SetLabelColor(Color.Gainsboro);

CSCFunctionPlot.SetStyle(true, true, false);
CSCFunctionPlot.SetNumberOfGridLines(4, 3);

// specify custom colors for drawing the coordinate frame
CSCFunctionPlot.SetAxesStyle(Color.FromArgb(255, 0, 80, 140), 2.0);
CSCFunctionPlot.SetGridStyle(Color.FromArgb(255, 0, 80, 140), 1.0);
CSCFunctionPlot.SetFrameStyle(Color.FromArgb(255, 0, 80, 140), 1.0);

CSCFunctionPlot.SetLegendTextOffset(2, 10);

// Draw the parametric function in a very light gray (Gainsboro))
CSCFunctionPlot.SetStyle(Color.Gainsboro, 1.0);

// generate an instance of the parametric curve defined in the classcode
// This instance will be used again in the custom graphics block below.
FImplicit f = new FImplicit(3, 4, 1.0, 200.0, 1.0, 150, 20000, 0.0, 360.0);

// plot the parametric curve
CSCFunctionPlot.ParametricPlot(f);

// add custon graphics.
CSCDisplay.BeginRendering();
CSCRender2D.SetFillColor(Color.White);

// A custom headline
CSCRender2D.DrawCenteredText(0.0, 1.1, "A Parametric plot", 0, 0);

// Render a text line containing the parameters a, b, c and d at the bottom
CSCRender2D.DrawText(1.0, -2.15, f.ToString(), 0, 0);

CSCDisplay.EndRendering();
CSCDisplay.Present();
CSCDataIO.WriteSurface();

/* <classcode> */
// define the parametric curve as CSCRealVectorFunction
public class FImplicit : CSCRealVectorFunction
{
 public FImplicit(int j, int k,
                 double a, double b,
                 double c, double d,
                 int numsamples, double xmin, double xmax) : base(xmin, xmax)
 {

// _j and _k are the exponents of the implicit function
  _j = j;
  _k = k;


// _a, _b, _c and _d are the frequency parameters
```

```
 _a = a;
 _b = b;
 _c = c;
 _d = d;

 // resample the curve and disable interpolation mode
 Resample(numsamples);

 // disable interpolation
 SetInterpolationMode(false);
}

// this method returns the value of the parametric curve. The 0 component
// of the vector function stores the x coordinate and the 1 component stores
// the y coordinate of the implicit function.
public override CSCRealVector Value(double x)
{
 CSCRealVector v = new CSCRealVector(2);

 // convert degrees to rads
 double t = x*Math.PI/180.0;

 v[0] = Math.Cos(_a*t) - Math.Pow(Math.Cos(_b*t), _j);
 v[1] = Math.Sin(_c*t) - Math.Pow(Math.Sin(_d*t), _k);

 return v;
}

// we also override ToString() and return a formatted parameter string to
// be added to the decorations of the rendered plot.
public override string ToString()
{
 string s = "Parameters: a=" + _a.ToString("#####.##") +
                       ", b=" + _b.ToString("#####.##") +
                       ", c=" + _c.ToString("#####.##")
                     + ", d=" + _d.ToString("#####.##");

 return s;
}


// the parameters of the curve as private class attributes. specified in
// the constructor
private int _j = 1;
private int _k = 1;

private double _a = 1.0;
private double _b = 1.0;
private double _c = 1.0;
private double _d = 1.0;
}
```

Executing this script generates the following plot (with a slightly different headline).

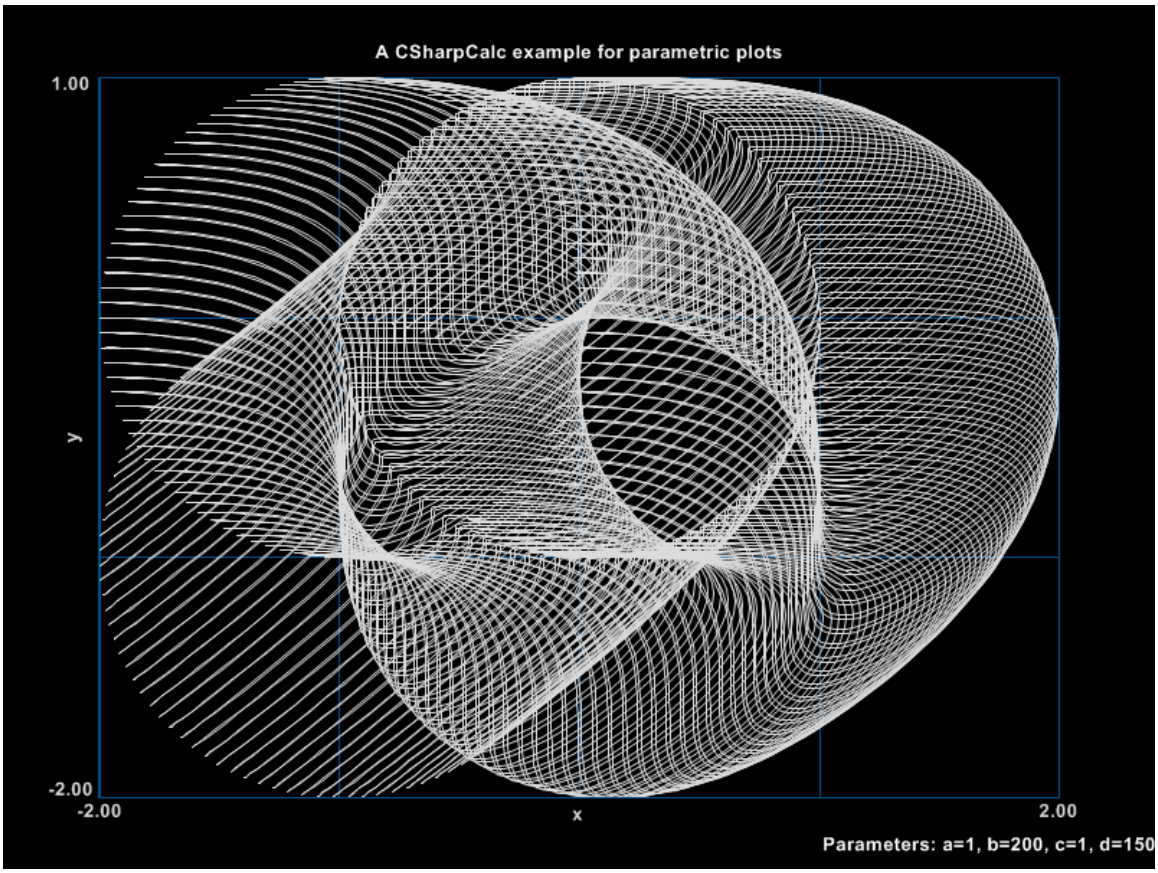

Figure 9. A parametric plot with fully customized rendering style

Advanced topic: Visualization and animation of Newtonian mechanics

As an example for an advanced project in CSharpCalc we shall develop a simulation which solves the classical (Newtonian) equations of motion of a one dimensional non-linear oscillator. The project will be built step by step. In step one we present a class which solves the equations of motion. If you are not into physics you may just use this class as one that calculates the correct solutions. Understanding physics is fun but in no way required for understanding the scripting of CSharpCalc. In the second step we will implement classes which draw customized coordinate frames. Like the calculating class these will be stored in an inline file to be used in the final script. The third step is dedicated to the generation of the static solution plots. Finally in step 4 we shall see how these solution plots can be animated and how static solutions can be overlaid with animated sequences.

Step 1: The one dimensional, non-linear oscillator

The system under consideration is a one-dimensional point particle with unit mass which is subject to the potential: $V(x) = x^4 + \lambda x^2 + \mu x$. Here λ (C# variable lambda) and μ (C# variable mu) are real valued parameters which can be positive, negative or zero. The figure below displays the graph of this potential for lambda = -2.5 and mu = 0.8.

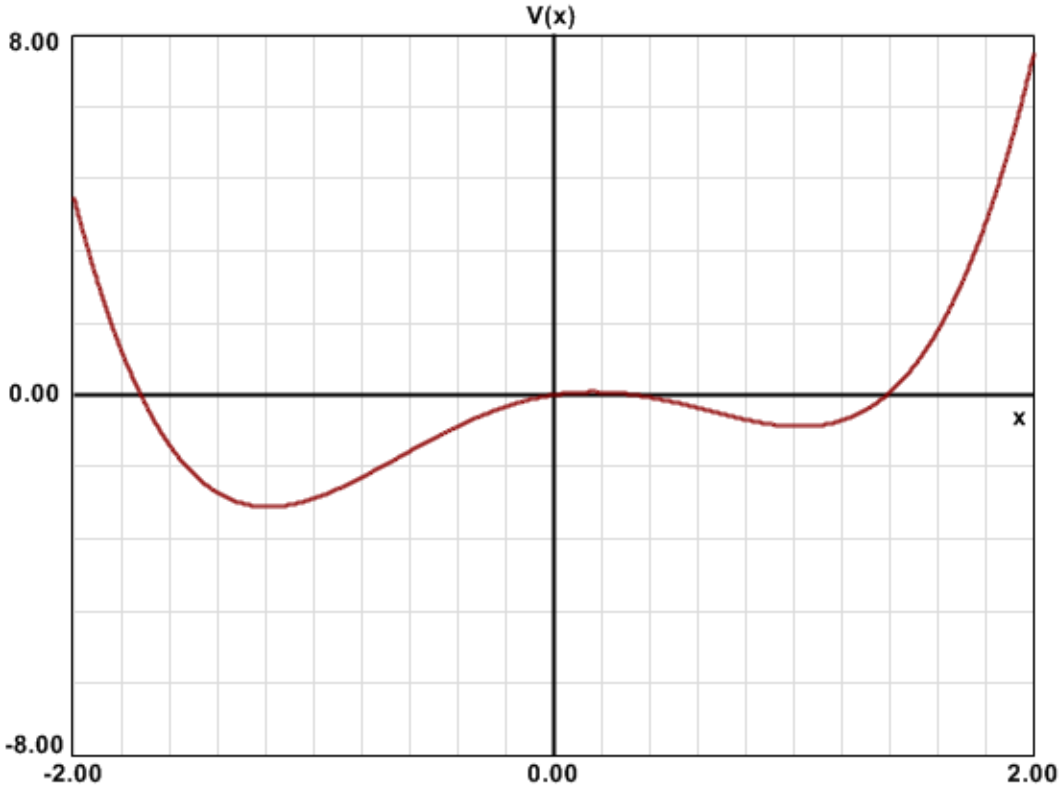

Figure 10. An anharmonic oscillator potential with two minima

In addition to this potential, the particle will also be slowed by a friction force $F(p) = -\delta p$. which is linearly dependent on the particles momentum. The parameter δ (C# variable delta) is called a damping coefficient and must be strictly non-negative. Hence, the particle, once released, will constantly lose energy until its movement finally stops.

We will use a second order series expansion for numerically solving the Newtonian equations of motion:

$$\frac{dx}{dt} = p \qquad \text{and} \qquad \frac{dp}{dt} = -4x^3 - 2\lambda x - \mu - \delta p$$

The damping coefficient delta controls the strength of the friction effect.

The full Oscillator1D class reads as follows:

```
public static class Oscillator1D
{
 public static void Initialize(double lambda, double mu, double delta, double
x0, double p0, double dt, double T)
 {
  _lambda = lambda;
  _mu = mu;
  _delta = delta;
  _x0 = x0; _p0 = p0;
```

```
  _dt = dt; _T = T;
  __x =  _x0;
  __p =  _p0;
  __xx = __x; __pp = __p;

  _x = new CSCRealFunction(0.0, _T);
  _p = new CSCRealFunction(0.0, _T);
  _h = new CSCRealVectorFunction();
 }

 public static void Solve()
 {
  int nsteps = (int)(_T/_dt);
  X.Append(__x); P.Append(__p);
  H.Append(new CSCRealVector(new double[] { __x, __p }));
  for(int i = 0; i < nsteps; i++)
  {
   __xx = __x + __p*_dt + 0.5*(F1(__x) - _delta*__p)*_dt*_dt;
   __pp   =   __p   +   (F1(__x)   -   _delta*__p)*_dt   +   0.5*(F2(__x)*__p   -
delta*_delta*__p -
delta*F1(__x))*_dt*_dt;
   __x = __xx; __p = __pp;
   X.Append(__x); P.Append(__p);
   H.Append(new CSCRealVector(new double[] { __x, __p }));
  }
 }
 public static void Solve(double DT, double MaxT)
 {
  int nsteps = (int)(DT/_dt);
  X.Append(__x); P.Append(__p);
  X.XMax = MaxT; P.XMax = MaxT;
  H.Append(new CSCRealVector(new double[] { __x, __p }));
  for(int i = 0; i < nsteps; i++)
  {
   __xx = __x + __p*_dt + 0.5*(F1(__x) - _delta*__p)*_dt*_dt;
   __pp   =   __p   +   (F1(__x)   -   _delta*__p)*_dt   +   0.5*(F2(__x)*__p   -
_delta*_delta*__p - _delta*F1(__x))*_dt*_dt;
   __x = __xx; __p = __pp;
   X.Append(__x); P.Append(__p);
   H.Append(new CSCRealVector(new double[] { __x, __p }));
  }
 }

 public static CSCRealFunction X { get { return _x; } }
 public static CSCRealFunction P { get { return _p; } }
 public static CSCRealVectorFunction H {  get { return _h; } }

 // the potential V(x)
 public static CSCRealFunction V(double min, double max, double dx)
 {
  CSCRealFunction v = new CSCRealFunction(min, max);
  int nsteps = (int)((max - min)/dx);
  double x = min;
  double y = x*x*x*x + _lambda*x*x + _mu*x;
  v.Samples.Add(y);

  for(int i = 0; i < nsteps; i++)
```

```
  {
   x += dx;
   y = x*x*x*x + _lambda*x*x + _mu*x;
   v.Samples.Add(y);
  }
  return v;
 }

 // the negative first derivative of the potential a.k.a force
 private static double F1(double x)
 {
  return -(4*x*x*x + 2.0*_lambda*x + _mu);
 }

 // the negative second derivative of the potential
 private static double F2(double x)
 {
  return -(12*x*x + 2.0*_lambda);
 }
 private static double _lambda = 1.0;
 private static double _mu = -1.0;
 private static double _delta = 0.0;
 private static double _x0 = 1.0;
 private static double _p0 = 0.0;
 private static double _dt = 0.01;
 private static double _T = 10.0;

 private static CSCRealFunction _x = null;
 private static CSCRealFunction _p = null;
 private static CSCRealVectorFunction _h = null;

 private static double __x = 0.0;
 private static double __p = 0.0;
 private static double __xx = 0.0;
 private static double __pp = 0.0;
}
```

The class interface is simple. Call *Initialize(..)* in order to initialize the oscillator class with the necessary parameters. Lambda, mu and delta define the forces the point particle is subject to. X0 and P0 are the particle's position and momentum at t=0. The parameter dt is the step size of one iteration of our numerical solution method. T (actually the interval [0, T]) is the time range for which the system is solved.

The first versions of *Solve(),* without parameters, calculates the full solution in one turn. The second version calculates a piece of the solution. This version will be used later for producing animations with a frame by frame rendering technique.

Copy this code into the CSharpCalc editor and export it to an inline file to be used in subsequent scripts.

Step 2: The coordinate frames

In order to display the solutions within a decently looking coordinate frame we need a fully customized class that renders such a frame. For this purpose CSharpCalc natively provides a

class named *CSCCartesianCoordinateSystem* which represents a generic 2D Cartesian coordinate system. We will use this class for implementing a variable-size coordinate frame with centered axes. Moreover, our frame will have a line grid, labels at the axes and numerical letterings around the frame. The code for this coordinate class reads as follows.

```
public static class CenteredCoordinateSystem
{
 public static void Initialize(double xsize, double ysize, double xgridlines,
double ygridlines, string xlabel, string ylabel)
 {
  _xsize = xsize; _ysize = ysize;
  _xgridlines = xgridlines; _ygridlines = ygridlines;

  _ccs = new CSCCartesianCoordinateSystem(-_xsize, -_ysize, _xsize, _ysize,
_xsize/_xgridlines, _ysize/_ygridlines);
  _ccs.XLabel = xlabel;
  _ccs.YLabel = ylabel;
  // Let the surface contain the coordinate system with a 20% margin
  CSCDisplay.SetPhysicalSize(_ccs, 0.2*_xsize, 0.2*_ysize);
 }

 public static void Draw()
 {
  // Draw the coordinate system.
  CSCRender2D.DrawGrid(_ccs, Color.Gainsboro, 1);
  CSCRender2D.DrawFrame(_ccs, Color.Black, 1);
  CSCRender2D.DrawAxes(_ccs, Color.Black, 2);

  CSCRender2D.SetLineStyle(Color.Black, 2);
  CSCRender2D.DrawXLabelAtAxis(_ccs, 0, 0);
  CSCRender2D.DrawYLabelAtAxis(_ccs, 0, 0);

  // draw the units along the x-axis
  CSCRender2D.DrawCenteredText(-_xsize, -_ysize, _xsize.ToString("-0.00"), 0, -
10);
  CSCRender2D.DrawCenteredText(0.0, -_ysize, "0.00", 0, -10);
  CSCRender2D.DrawCenteredText(_xsize, -_ysize, _xsize.ToString("0.00"), 0, -
10);

  // draw the units along the y-axis
  CSCRender2D.DrawCenteredText(-_xsize, -_ysize, _ysize.ToString("-0.00"), -20,
5);
  CSCRender2D.DrawCenteredText(-_xsize, 0.0, "0.00", -20, 0);
  CSCRender2D.DrawCenteredText(-_xsize, _ysize, _ysize.ToString("0.00"), -20, -
5);
 }

 private static double _xsize = 1.0;
 private static double _ysize = 1.0;
 private static double _xgridlines = 5.0;
 private static double _ygridlines = 5.0;
 private static CSCCartesianCoordinateSystem _ccs = null;
}
```

The Initialize method takes the size of the frame along with the number of grid lines and two string labels for the x- and y- axes. Calling Draw() renders this view:

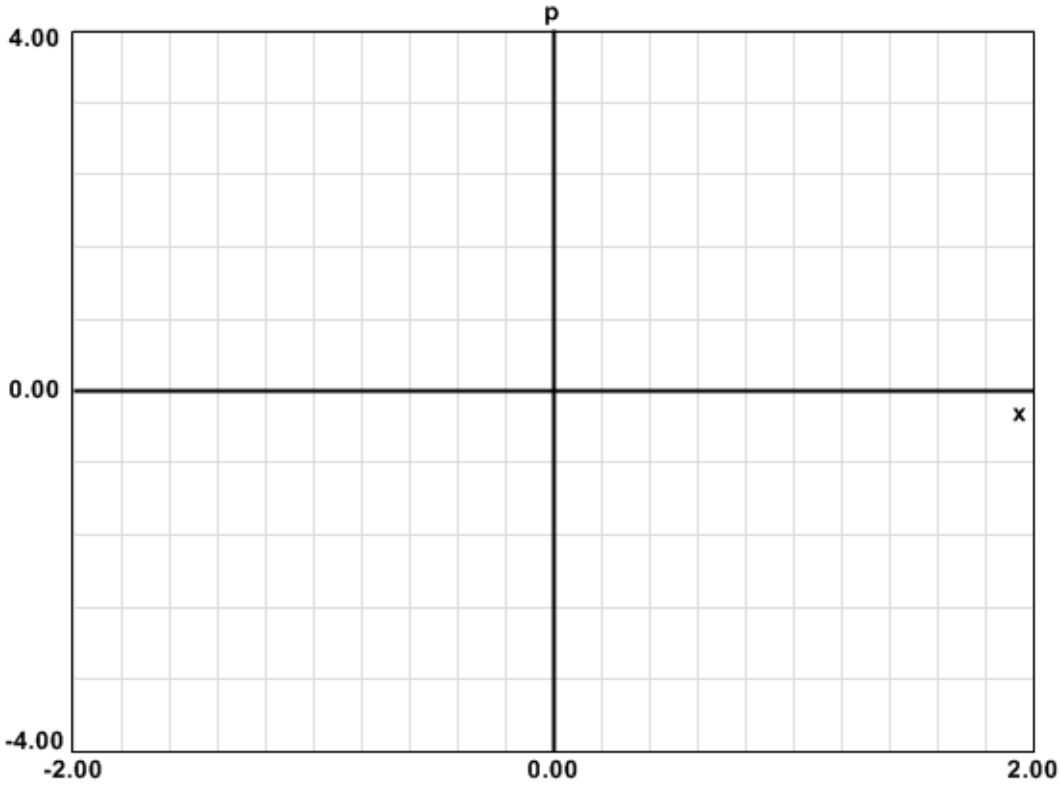

Figure 11. This coordinate frame is initialized by Initialize(2.0, 4.0, 10.0, 5.0, "x", "p")

The *Initialize* method of this class calls a different version of *CSCDisplay.SetPhysicalSize(..)* thus replacing the physical surface initialization. Since a coordinate system already defines the physical extensions of the view, it can be passed to *SetPhysicalSize* as an alternative to the lower left and upper right corner. In addition to its extensions, one needs to specify a margin around the coordinate frame in order to provide enough space for labels and possible other decorations such as legends.

Since the *Initialize* method of the coordinate system class already initializes the physical surface size it replaces this initialization in the main code. In other words, initializing the coordinate system already defines physical dimensions, hence no further initialization is required.

The class *CSCRender2D* has three high-level rendering methods for rendering coordinate systems which are all used in the *Draw()* method of our class. These methods are *DrawGrid* for rendering the grid lines, *DrawFrame* for rendering the outer frame and *DrawAxes* for rendering the coordinate cross.

The Draw() method uses *CSCRender2D.SetLineStyle(color, linewidth);* for modifying the color and thickness of the axes. In order to modify the filling of graphical primitives (not used here) one would call *CSCRender2D.SetFillStyle*.

Modifying this class for rendering different types of coordinate systems is left to the reader as an exercise.

Step 3: Solving the anharmonic oscillator

In this step we will use our classes for producing graphical views of the system's hodograph for selected initial conditions. Due to the fact that we outsourced both the calculation and the rendering of the coordinate frame to inline classes this script is exceedingly short. It reads as follows:

```
//inline Manual\Mechanics1D.CCS.csc
//inline Manual\Mechanics1D.Oscillator02.csc

double lambda = -2.5;
double mu = 0.8;
double delta = 0.05;

double X0 = 2.0;
double P0 = 0.0;

double dt = 0.0001;
double T = 50.0;

CSCDisplay.SetSurfaceSize(600, 450);

// This method replaces the initialization of the physical size
CenteredCoordinateSystem.Initialize(2.2, 4.5, 10.0, 5.0, "x", "p");

CSCDisplay.Clear(Color.White);
CSCDisplay.BeginRendering();

// draw the selected coordinate system
CenteredCoordinateSystem.Draw();

// generate the hodograph
Oscillator1D.Initialize(lambda, mu, delta, X0, P0, dt, T);
Oscillator1D.Solve();

// and draw it
CSCRender2D.SetLineStyle(Color.DarkSlateGray, 1.6f);
CSCRender2D.DrawTrajectory(Oscillator1D.H, 0, 1);

CSCDisplay.EndRendering();
CSCDataIO.WriteSurface();

// Present the surface
CSCDisplay.Present();
```

Running this script generates the following hodograph.

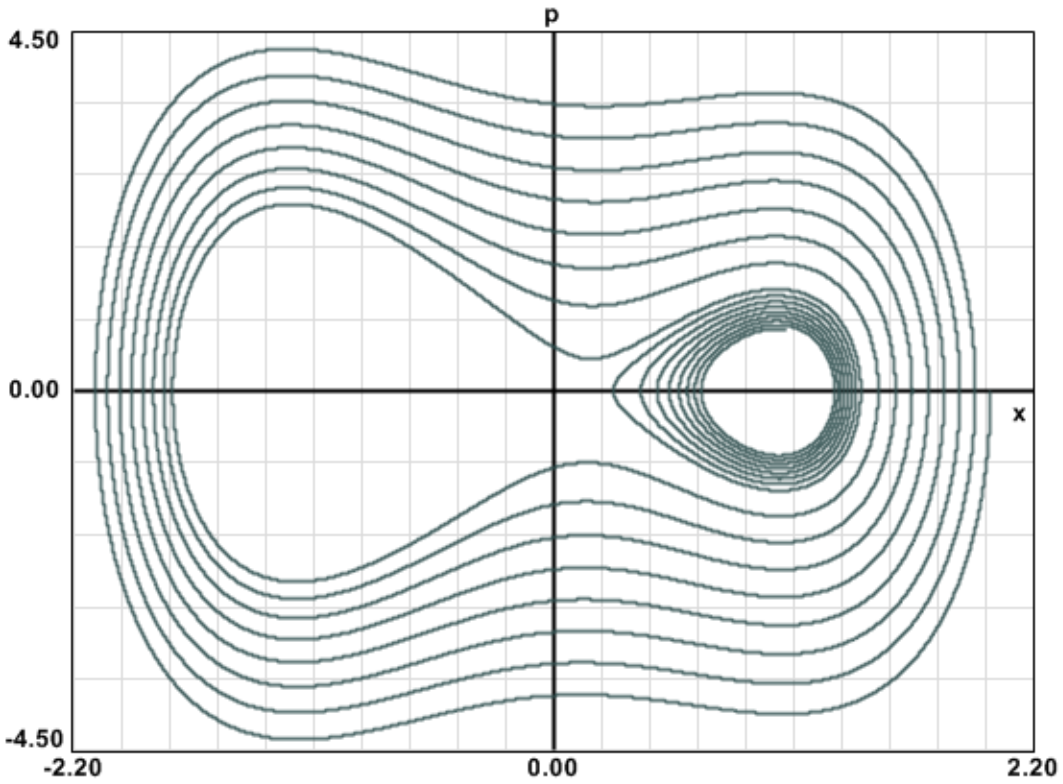

Figure 12. Hodograph of the damped oscillator for lambda=-2.5, mu=0.8, delta=0.05, X0=2.0 and P0 = 0.0.

The inline file *Mechanics1D.CCS.css* contains the class for drawing our Cartesian coordinate system (CCS). The file *Mechanics1D.Oscillator02.css* contains the oscillator class discussed above which solves the equations of motion for the damped, anharmonic oscillator. By inlining these classes instead of stuffing all the code in one file we keep the main script short, structured and readable. If you stored the classes in files with different names please adapt the code of this script accordingly.

The central rendering method used here is *CSCRender2D.DrawTrajectory(f, c1, c2)*. The first parameter *f* denotes an instance of *CSCRealVectorFunction*. This class stores sequences of the type *CSCRealVector* just like a *CSCRealFunction* stores sequences of real values. The parameters c1 and c2 are vector component indices and must not be equal. The method *DrawTrajectory* connects subsequent points projected to the dimensions defined by c1 and c2. As an example consider a vector function storing a sequence of four 3D vectors, i.e. f=(x1, x2, x3, x4) such that each xi is a 3D vector and hence has 3 components xi[0], xi[1] and xi[2]. Assume that c1=0 and c2=2. Then *DrawTrajectory(f, 0, 2);* will draw the following lines on the surface:

line 1 from (x1[0], x1[2]) to (x2[0], x2[2])

line 2 from (x2[0], x2[2]) to (x3[0], x3[2])

line 3 from (x3[0], x3[2]) to (x4[0], x4[2])

Obviously this line sequence is a projection of the vector function's graph onto the XZ-plane.

Step 4: An animated phase plot

In order to obtain a visual impression of phase plot dynamics we will dedicate this last step to rendering an animated hodograph.

For animating the phase plot we shall modify the previous script to render the time evolution of the dynamic system progressively. The resulting images will be written to the data directory as .png image files using the file counting mechanism implemented in *CSCDataIO*. The actual animation needs to be composed with an external program of your choice. However, once all the frames are available this is a rather easy task as there is a variety of freeware tools which do the job. We used ImageMagick for combining the frames into a simple gif animation. The animation script reads as follows:

```
//inline Manual\Mechanics1D.CCS.csc
//inline Manual\Mechanics1D.Oscillator02.csc

double lambda = -2.5;
double delta = 0.1;
double mu = 0.8;
double X0 = 2.0;
double P0 = 0.0;
double dt = 0.0001;
double T = 40.0;
int numframes = 800;
double DT = T/numframes;

CSCDisplay.SetSurfaceSize(600, 450);

CenteredCoordinateSystem.Initialize(2.2, 4.5, 10.0, 5.0, "x", "p");

// set the name of the files to 'phaseplot' and enable file counting
// with 1 (i.e. 00000001) as starting index
CSCDataIO.SetFilePattern("phaseplot", "");
CSCDataIO.EnableFileCounting(1);

// make a preview of the full trajectory and store it in HH.
Oscillator1D.Initialize(lambda, mu, delta, X0, P0, dt, T);
Oscillator1D.Solve();
CSCRealVectorFunction HH = new
CSCRealVectorFunction(Oscillator1D.H.Serialize());

Oscillator1D.Initialize(lambda, mu, delta, X0, P0, dt, T);
CSCDisplay.Clear(Color.White);

// iterate and render the phaseplot step by step
for(int i = 1; i <= numframes; i++)
{
```

```
 CSCDisplay.BeginRendering();

 // solve the oscillator system step by step
 Oscillator1D.Solve(DT, i*DT);

 // draw the coordinate frame first...
 CenteredCoordinateSystem.Draw();

 // then draw the preview trajectory in a very light gray color..
 CSCRender2D.SetLineStyle(Color.Gainsboro, 1.6f);
 CSCRender2D.DrawTrajectory(HH, 0, 1);

 // finally draw the next piece of the animated trajectory in with
 // a darker pen.
 CSCRender2D.SetLineStyle(Color.DarkSlateGray, 1.6f);
 CSCRender2D.DrawTrajectory(Oscillator1D.H, 0, 1);

 // Write the frame. Since file counting is enabled SCSDataIO
 // does the indexing automatically
 CSCDataIO.WriteSurface();

 // give a progress report in the console for every rendered frame
 CSCConsole.Writeln("Rendered animation frame: " + i.ToString("0000") + " / " +
numframes.ToString());
 CSCDisplay.EndRendering();

 // this color choice initializes the surface to be transparent
 CSCDisplay.Clear(Color.FromArgb(0, 0, 0, 0));
}

// finally report completion in the console.
CSCConsole.Writeln("Phase plot animation finished!");
```

Before starting the animation process, the script makes a preview of the phase plot and stores it in the vector function named HH. This preview is drawn in light gray before rendering the animated trajectory in order to achieve a nice "train on rails" effect. The trajectory HH is cloned from the original using its serialization. This mechanism is explained in detail below in the paragraph on function classes.

Copy the script to the CSharpCalc editor. Before you run it please make sure that the data directory is set to an empty folder since this script generates 800 animation frames. Watch the progress reports in the console as the animation frames are rendered.

Best practice

This section gives an incomplete list of best practice recommendations. It is not obligatory to follow these recommendations, but the benefits are obvious.

Before you start using CSharpCalc

Organize your workspace. Over time the number of inline files tends to grow. Organize these files in subdirectories and sort them by function. For example, one might create one folder for

coordinate systems, one for frequently used special functions, one for solving frequently used differential equations and so on. Also, use the *1.2.3.css* file name convention shown in the last script. This is particularly useful when you spawn new inline files from existing ones for specific projects. For example the same type of coordinate frame with minor adaptations for different projects can be organized in files using the naming convention *project1.coord.css*, *project2.coord.css* and so on.

Backup your workspace often. The workspace contains scripts that possibly took a lot of time to develop and it would be a shame to lose this time to neglect. Make regular backups of your workspace. When used right it collects all of your assets in one place and making backups often saves you from annoying data loss.

Make a version history of your workspace. Sometimes, code development goes into the wrong direction and one needs to roll a few steps back. The workspace contains only text and in most cases will not consume a lot of storage space. Making dated snapshots of the workspace enables you to roll back days, weeks or even months which can be life saving on certain occasions.

Before you start writing your script

Define a data directory. Unless you intend to save absolutely no results in files, the data directory should be well configured. It also makes sense to organize and backup the data directories in one common root folder. Keeping the data directories within the workspace is possible but not recommended as this would thwart the backup and version history concept described above. Please keep in mind that data directories might occupy gigabytes of storage space in the case that they contain images and animations. Backup these directories as you see fit for your personal safety demands.

Design a flexible API for your inline classes. In order to maintain a maximum of flexibility and readability within your scripts we strongly recommend to spend some thought and effort on the design and implementation of a well-pondered API design. This is even more true when implementing reusable classes.

If you develop an inline class keep the code used for testing it in a separate script. Sometimes inline classes require adaptations and improvements. Generally, inline classes can be opened and edited with CSharpCalc, but they do not compile since they are not executable. If you keep the scripts you used for developing these classes in a separate folder adaptations and modifications are much easier to accomplish.

Serialize your results and save the serializations in files. Many CSharpCalc classes have a serialize method. Using this method you can save a string representation of any class instance and reconstruct the instance from this string for later use. In principle it is even possible to send a class instance represented by its serialization to someone else via Email.

The CSharpCalc class library

Overview

CSharpCalc comes with a collection of build in classes. These classes offer access to the output devices and implement some basic data types. This manual does not give an exhaustive description of each class. Detailed interface documentations of these classes can be found in the class documentations accessible from the browser. The subsequent text gives an short presentation of each class in order to provide the user with an overview on the capabilities of CSharpCalc.

Class types

Framework classes

Framework classes are classes which provide access to CSharpCalc devices and resources such as the rendering system, system color maps, the display or the console. These classes have no common prototype since each of them represents a different facet of CSharpCalc.

The version 1.0 framework classes are *CSCCartesianCoordinateSystem*, *CSCColorMap*, *CSCConsole*, *CSCDataIO*, *CSCDisplay, CSCRender2D* and *CSCFunctionPlot*.

Algebraic type classes

Algebraic type classes represent algebraic types such as complex numbers, vectors and matrices. They have a common prototype which implements a serialization mechanism, i.e. every instance can be represented as string. One version of the constructor takes such a serialization as argument and (re)constructs the instance stored in the serialization. The serialization of a class instance can be requested using the *Serialize()* method.

In order to facilitate structured text output of algebraic types these classes offer a method called *ToFormatString()* which takes a standard number (C# type double) formatting string as argument.

The version 1.0 algebraic type classes are: *CSCComplex*, *CSCRealVector*, *CSCComplexVector*, *CSCRealMatrix* and *CSCComplexMatrix*.

Function classes

Function classes represent either lists of algebraic types or maps from a closed interval of real numbers [XMin, XMax] to algebraic types. They have a common prototype which allows for linear interpolation of function values between the supporting values. This interface is implemented using the subscript operator. If f is a function and r a real number within the interval of definition then f[r] returns a linearly interpolated value of the function's algebraic type. Outside of the interval of definition function values are clamped to the boundary values. If [XMin, XMax] is the interval of definition then f[r] = f[XMax] for all values of r larger than XMax. Likewise f[r]=f[XMin] for all r smaller than XMin.

If a user-defined function is derived from one of the function classes by overwriting the *virtual Value()* method interpolation should be turned off by calling *SetInterpolationMode(false)*. When value interpolation is turned off, the subscript operator will return the function value returned by *Value(x)* instead of interpolating between sample values.

The serialization interface described for algebraic types is also implemented for functions and works in a similar manner. Also formatted text display using *ToFormatString()* is implemented for these classes.

In order to use functions as lists one may use the *Append(<type>)* method for adding members to the list of supporting values. In the case that the function is used as mere list the interval of definitions can be ignored. The list of samples can be accessed using the *Samples* accessor. Alternatively, vanilla C# *List<type>* constructs are also supported by CSharpCalc.

The version 1.0 function classes are: *CSCRealFunction*, *CSCComplexFunction*, *CSCRealVectorFunction* and *CSCComplexVectorFunction*.

The CSharpCalc v1.0 classes

CSCCartesianCoordinateSystem

The class *CSCCartesianCoordinateSystem* is a data structure for defining coordinate systems. An instance of this class can also be passed to *CSCDisplay* for initializing the physical extensions of the drawing surface. The current version of the class supports the drawing of axes, a grid and textual labels along the axes. All length units passed to this structure are measured in physical units (as opposed to pixel units). Physical units may range from femtometer to mega parsec depending on the rendered graphics.

CSCColorMap

The class *CSCColorMap* is a collection of predefined 256 value color maps. In addition to the capability of mapping index values in the range from 0 to 255 to colors this class also contains static methods for mixing and multiplying color values. The colors passed to these mixing functions need not be members of the active color map. All Argb color definitions are valid in these methods!

In order to select a specific color map one of the *CSCColorMapType*s needs to be passed to the *SetTo(..)* method. A list of available color maps can be found via the browser.

CSCConsole

The class *CSCConsole* provides methods for writing text to the console. Moreover, the *SaveText(..)* method allows you to write the text shown in the console to a file which will be placed in the active data directory.

CSCDataIO

This class offers methods for reading (very limited on purpose) and writing data from and to the active data directory. The class supports the consecutive writing of indexed files for rendering animated sequences through its file counting mechanism.

CSCDisplay

The class *CSCDisplay* represents the graphic display. It's use is explained above in detail. It is recommended to refrain from accessing the surface bitmap and its graphics context directly but if need requires it the class provides read only access to these structures. The class also offers geometry mapping functions which may be used to map physical coordinates to pixel coordinates and vice versa.

CSCRender2D

This class provides methods for rendering graphical primitives such as lines, rectangles and circles. It also offers methods for rendering some CSC classes such as coordinate systems (or parts of them), functions and trajectories. A few style specification methods provide detailed control of colors, line widths and fonts.

CSCFunctionPlot

The *CSCFunctionPlot* class provides an easy and fast way for plotting function graphs and parametric curves. Everything this class does can also be done using the CSCRender2D methods but using this class will save a lot of time and code lines. Adding custom graphics to the output of *CSCFunctionPlot* is simple and encouraged. The graphics drawn by this class are largely customizable.

CSCComplex

This class represents complex numbers. It implements most algebraic operations like addition, subtraction multiplication and division. An example for using complex numbers is found in the above text in the Mandelbrot script.

CSCComplexFunction

CSCComplexFunction represents a complex valued function of a real variable (e.g. time) on a given interval [XMin, XMax]. The subscript operator [] returns linearly interpolated function values for all values of its argument within the interval of definition a.k.a domain. Function values for arguments outside of the domain are clamped to the boundaries. Interpolation can be turned off for derived complex functions by calling *SetInterpolationMode(false)*.

The *Serialize()* method returns a string representation of the given instance of the function. This serialization can be passed to the function's constructor for creating an exact copy of the

function. This is also useful for using results from one calculation in other calculations or models even in different scripts.

CSCComplexMatrix

This class represents a general n x m matrix with complex entries. Most algebraic operations of matrices are implemented as far as they are well defined. Please remember that the matrix product of general n x m matrices is only defined if the width of the first matrix equals the height of the second. The matrix classes implement a static method for numerically calculating the matrix exponential. The subscript operator allows for accessing the matrix elements. Moreover, matrices can be used as operators acting on vectors using the regular multiplication operator, i.e. if M is an n x n matrix and v is an n-dimensional vector then M*v is also an n-dimensional vector equal to M operating on v.

CSCComplexVector

CSCComplexVector represents a n-dimensional vector with complex components. All algebraic operations of vectors are implemented.

CSCComplexVectorFunction

A complex vector function is a map from a closed interval of real numbers to the complex vectors. It implements the function prototype explained above.

CSCRealFunction

This class represents a real valued function. It implements the function prototype explained above.

CSCRealMatrix

CSCRealMatrix implements an n x m matrix with real entries.

CSCRealVector

This class implements an n-dimensional vector with real components.

CSCRealVectorFunction

This class implements a map from a closed interval of real numbers to the real vectors. It implements the function prototype explained above.