The simulator

Technical documentation of the "Simulator" program

The computer program "Simulator", the use of which is described in a separate manual, enables the simulation of simple dynamic systems and experimentation with them. The code is publicly accessible on GitHub, written in VB.NET, provided with detailed comments and can be extended as required. This requires the free community version of Microsoft Visual Studio, at least version 17.9, which is based on Microsoft Framework 8.0.

This document describes the technical structure and architecture of the simulator. The mathematical principles can be found in the document "Dynamic systems".

Version 6.2 - 01.12.2024

Contents

Introduction

The "Simulator" enables the simulation of simple dynamic systems. The mathematical principles for this and the concepts for implementation are described in the document "Dynamic systems". The "User manual" explains how to use the "Simulator".

The code of the program is published in GitHub and is available as open source. It is written in VB.NET and extensively commented (in English). The development environment is the community version of Microsoft Visual Studio 2022, which is available free of charge and easy to install. At least version 17.9 is required, which is based on Microsoft Framework 8.0, which must also be downloaded and installed if it is not already available.

The current version of the "Simulator" is 6.2.0 published on 1.12.2024.

The GitHub link is as follows:

https://github.com/HermannBiner/Simulator

This document contains the technical documentation for the "Simulator" and instructions if you want to develop your own dynamic systems in the "Simulator".

1. Basic concepts

1.1 Designations and programming standards

In principle, all designations and every new word in them begin with capital letters.

The type of an object is usually abbreviated with three letters, e.g.

- Cls for a class, e.g. *ClsGraphicTool*
- Frm for a Windows form, e.g. *FrmPendulum*
- Txt for a text field, e.g. *TxtParameter*

Etc.

An exception is an interface. This is simply identified with an "I" in front of it. E.g.: *IBilliard*.

Abstract classes are provided with a trailing "Abstract". E.g.: *ClsGrowthModelAbstract*.

Variables of type Boolean usually have the prefix "Is". For example, *IsFormLoaded*.

Basically, all designations are in English. Likewise, all comments in the code.

Parameters are always transferred between classes using *properties*. This allows further code to be executed or a check to be carried out when the parameter is passed. If a parameter with the name *Example* is passed, it is designated "*MyExample*" within the class. This distinguishes parameters that are available outside the class from the private parameters of a class.

Example: *ClsGeneralParameter*

```
18 Verweise
Property DefaultValue As Decimal
    Set(value As Decimal)
        MyDefaultValue = value
    End Set
    Get
        DefaultValue = MyDefaultValue
    End Get
End Property
```

In addition, each parameter and each method of a class is provided with *Private, Protected, Public.*

Implicit type conversion is not permitted. Because of the calculation accuracy, the number types *decimal* and also *integer* are usually used. Sometimes (for large numerical values) *double* is also used.

1.2 Versioning

The *MainVersion.MinorVersion.Patch* numbers are updated as follows:

Main version

This is increased when a new category of dynamic systems is implemented. The categories in main version 6 are e.g. Billiards, Growth models, Mechanics, Complex iteration. A new category leads to a new menu item in the main menu. The current status "6" of the main version is historically conditioned, due to major architectural unifications or profound performance optimizations.

Minor version

This is increased if another dynamic system has been implemented within an existing category or if a new representation is available for an existing dynamic system.

Patch

Increased for bug fixing and general optimizations.

If the main version number is increased, the other numbers start at 0 again.

1.3 State space of a dynamic system

A *dynamic system* is a triple (T, X, f) . The set T is the *time period*. Only discrete dynamic systems are considered in the "Simulator". Therefore, always $T = \mathbb{N}$.

The set X is the *state space*. For the systems implemented in the "Simulator", the state space is a subset of $\mathbb{R}, \mathbb{R}^n$ or $\mathbb{C}, \bar{\mathbb{C}}$. The current system state is defined by corresponding parameters.

Examples:

- Billiards: $(t, \alpha) \in [a, b] \times]0, \pi[\subset \mathbb{R}^2$
- Growth models: $x \in [a, b] \subset \mathbb{R}$
- Iteration in the complex: $z \in \mathbb{C}, \bar{\mathbb{C}}$
- Mechanics: n-tuple $\subset \mathbb{R}^n$

The dynamic system "knows" its parameters. For a parameter to be defined, it needs:

- One ID
- A name
- The definition range in which the parameter is permitted
- The type oft he parameter
- A default start value, which is set before the user changes it

This is in the class

ClsGeneralParameter

made available.

The types of the parameter can also be found in *ClsGeneralParameter*.

```
Public Enum TypeOfParameterEnum
    Variable
    DS
    Constant
End Enum
```

variable is a "normal" value parameter of the dynamic system. It defines the current state of a system and changes during the iteration (e.g. the deflection angle of the double pendulum).

DS is the parameter type that plays a special role in the transition to chaotic behaviour and thus to the essential property of the dynamic system, e.g. the parameter a in logistic growth. This parameter plays an important role in the formula of the law of motion of the dynamic system and thus in corresponding diagrams (e.g. Feigenbaum diagram or CDiagram), where they are mapped in the direction of the x-axis.

Constant is a parameter that can be defined at the start of the iteration, but does not change during the iteration (e.g. the lengths of the double pendulum).

In a class that represents a dynamic system, these parameter definitions must be made. This leads, for example, to the following code (here in the *ClsDoublePendulum* class in the *New* method):

```
'Inizialize all parameters
'Tag is the Number of the Label in the Pendulum Form
'L1
ValueParameter(0) = New ClsGeneralParameter(1, "L1", New ClsInterval(CDec(0.1), CDec(0.85)),
                                            ClsGeneralParameter.TypeOfParameterEnum.Constant, CDec(0.7))
MyValueParameterDefinition.Add(ValueParameter(0))

'L2
ValueParameter(1) = New ClsGeneralParameter(2, "L2", New ClsInterval(CDec(0.1), CDec(0.85)),
                                            ClsGeneralParameter.TypeOfParameterEnum.Constant, CDec(0.2))
MyValueParameterDefinition.Add(ValueParameter(1))

'Phi1
ValueParameter(2) = New ClsGeneralParameter(3, "Phi 1", New ClsInterval(-CDec(Math.PI), CDec(Math.PI)),
                                            ClsGeneralParameter.TypeOfParameterEnum.Variable, CDec(Math.PI / 4))
MyValueParameterDefinition.Add(ValueParameter(2))

'Phi2
ValueParameter(3) = New ClsGeneralParameter(4, "Phi 2", New ClsInterval(-CDec(Math.PI), CDec(Math.PI)),
                                            ClsGeneralParameter.TypeOfParameterEnum.Variable, CDec(Math.PI / 6))
MyValueParameterDefinition.Add(ValueParameter(3))
```

1.4 GeneralClassLibrary

This library contains many simple auxiliary classes, such as

- *ClsInterval*
- *ClsGeneralParameter*
- *ClsMathPoint* (a point in the $\mathbb{R}^2$ in mathematical coordinates, in contrast to a *point* from the .NET library, which is always in pixel coordinates)
- *ClsValuePair* (exactly like a *ClsMathPoint* in terms of structure, but has a different role and represents a value parameter pair and not a point in the $\mathbb{R}^2$)
- *ClsNTuple* (here the dimension N can be defined when instantiating the tuple).
- *ClsMathHelperDS* (these contain mathematical helper functions that are tailored to the respective dynamic system, but are in principle available everywhere). What these helper classes "can" do is best seen in the code itself, which is commented accordingly.
- *ClsSystemBrushes* (for coloured representations where the colour depends on a number of iteration steps (e.g. when iterating in the complex), the standard system colours are defined here)
- *ClsCheckXY* (various checks are available here, e.g. whether a value is numerical, whether a pair of values represents an interval). A special role is played by
- *ClsCheckUserData* (a single value is passed here and checked to see whether it has a permitted format and whether it is in the permitted definition range of the parameter. It is also possible to check whether an interval lies within this permitted definition range).

Other classes in this library are explained below.

1.5 ClsGraphicTool

The movement of a dynamic system is mapped in the "Simulator" in a PictureBox *PicDiagram* or a BitMap *BmpDiagram*. We will see their different roles later. This is about the conversion between pixel and mathematical coordinates.

The *ClsGraphicTool* class takes care of this so that this does not have to be done individually for each dynamic system. It plays the main role in all graphical representations. It is passed:

- The PictureBox or BitMap in which you want to draw
- The intervals for the mathematical coordinates

$$(x, y) \in MathXInterval \times MathYInterval$$

Since the size or the number of pixels of the PictureBox or BitMap are thus known, the *ClsGraphicTool* class can carry out the corresponding conversions itself.

It then provides all the necessary drawing options, such as drawing a point, a line, an ellipse, a rectangle, etc.

1.6 ClsDiagramAreaSelector

After a diagram has been drawn, in certain cases (e.g. Feigenbaum diagram) the user should be able to select a rectangle in the diagram for further examination. The sizes of the corresponding mathematical intervals are then reset according to the user's selection.

This means that in this context, a distinction must always be made between the definition ranges in *ClsGeneralParameter* (these are transferred to the *ClsDiagramAreaSelector* class as *XRange* and *YRange*) and the sub-intervals selected by the user (these are *UserXRange* and *UserYRange*).

The typical procedure is then as follows:

- The *ClsDiagramAreaSelector* class is instantiated and set as the default*:*
  UserXRange = XRange, UserYRange = YRange
- At the same time, the *PicDiagram* in which the user selection is to take place is passed to the class
- Since the size of this diagram and the ranges are known, the class can perform conversions of pixel and mathematical coordinates itself.
- If the user now selects a section of the diagram, they do so by holding down the mouse button. The respective mouse position is transferred to the class. The class then draws a corresponding rectangle in the diagram. It converts the associated pixel coordinates into mathematical coordinates and sets the corresponding intervals *UserXRange, UserYRange* in the associated text boxes *TxtXMin, TxtXMax, TxtYMin, TxtYMax* in the user window. The reference to these fields is transferred to the class during instantiation. See the corresponding code in *ClsDiagramAreaSelector*.
- Now the dynamic system will no longer calculate with the variable parameters in the range *XRange x YRange*, but in *UserXRange x UserYRange*. If the user then makes another selection, this will be relative to this new range.

Example code for this can be found in *ClsFeigenbaumController* during instantiation:

```
1 Verweis
Private Sub InitializeMe()

    DS.Power = 1

    ActualParameterRange = DS.FormulaParameter.Range
    ActualValueRange = DS.ValueParameter.Range

    With DiagramAreaSelector
        .XRange = ActualParameterRange
        .YRange = ActualValueRange
        .PicDiagram = MyForm.PicDiagram
        .TxtXMin = MyForm.TxtAMin
        .TxtXMax = MyForm.TxtAMax
        .TxtYMin = MyForm.TxtXMin
        .TxtYMax = MyForm.TxtXMax
    End With

    BmpGraphics = New ClsGraphicTool(BmpDiagram, ActualParameterRange, ActualValueRange)

End Sub
```

The dynamic system must then also distinguish between the ranges defined in *ClsGeneralParameter* (these are referred to as *DS.FormulaParameter.Range* and *DS.ValueParameter.Range*) and the ranges defined by the user according to their selection. For the formula parameter, this is referred to as *ActualParameterRange* and for the value parameter as *ActualValueRange*.

When starting an iteration in the selection window, the previously transferred values in the text fields, which *ClsDiagramAreaSelector* has set, are transferred to these ranges. It is important that the mathematical intervals, which are held by *ClsGraphicTool*, are also adjusted accordingly. This results in the following code snippet, for example:

```
1 Verweis
Public Sub StartIteration()

    ResetIteration()
    If IterationStatus = ClsDynamics.EnIterationStatus.Stopped Then
        If IsUserDataOK() Then
            With MyForm
                DiagramAreaSelector.IsActivated = False
                IterationStatus = ClsDynamics.EnIterationStatus.Ready
                ActualParameterRange = New ClsInterval(CDec(.TxtAMin.Text), CDec(.TxtAMax.Text))
                BmpGraphics.MathXInterval = ActualParameterRange
                ActualValueRange = New ClsInterval(CDec(.TxtXMin.Text), CDec(.TxtXMax.Text))
                BmpGraphics.MathYInterval = ActualValueRange
                DiagramAreaSelector.UserXRange = ActualParameterRange
                DiagramAreaSelector.UserYRange = ActualValueRange
                .BtnStartIteration.Enabled = False
                .BtnReset.Enabled = False
                .BtnReset.Enabled = False
                .BtnDefault.Enabled = False
            End With
        Else
            'Message already generated
        End If
    End If
```

The DiagramAreaSelector must of course not be active during a running iteration.

1.7 ClsLanguageManager and localization

The *ClsLanguageManager* class is used to ensure that all displays appear in the selected language (currently D/E). It is implemented as a singleton and is then available globally.

The class has the *GetString("StrID")* method. The string that is passed must be a unique ID so that the corresponding text is returned in the defined language. The entries for this are listed in resource files

- *LabelsEN.resx*
- *LabelsDE.resx*

If no entry is found, *StrID* is returned.

A special case is when the user implements their own dynamic system. This will be explained in more detail later. The following applies to the linguistic designation.

Suppose the user creates a new class *ClsMyDynamicSystem*. This must implement an interface for the corresponding class of dynamic systems (e.g. *IBilliard*). When the user window is loaded, all classes that implement this interface are loaded *by reflection.* The *ClsMyDynamicSystem* class now also "appears". The Language Manager then searches the resource files for an entry with the class name *ClsMyDynamicSystem*. If an entry is found there, the system programmed by the user is displayed under this name in the selection combo of the user window. If there is no entry, the system appears under the name *ClsMyDynamicSystem*.

If further languages are to be made available one day, you simply have to change the enumeration

```
7 Verweise
Public Enum LanguageEnum
    English
    Deutsch
End Enum
```

The new language must be added and then another resource file must be created for this language. The existing entries with the same IDs are simply translated into the new language.

1.8 Parameter categories

It is useful to distinguish between the following categories of parameters.

Specifying parameters (DS parameters)

These are parameters that specify the type of dynamic system. For example, in the Newton iteration, the exponent n of the unit roots to be analyzed must be specified. Or for logistic growth, the *formula parameter* a.

Changing these parameters is equivalent to selecting a new dynamic system.

Start parameters

These define the state of the dynamic system at startup. These are *ValueParameters*. At the first start, these are set to the default values defined in *ClsGeneralParameter*. The user can change these start values without this being equivalent to changing the dynamic system. If the default values are to be set again, this is done using the *SetDefaultUserData* method, which is called this everywhere.

Display parameters

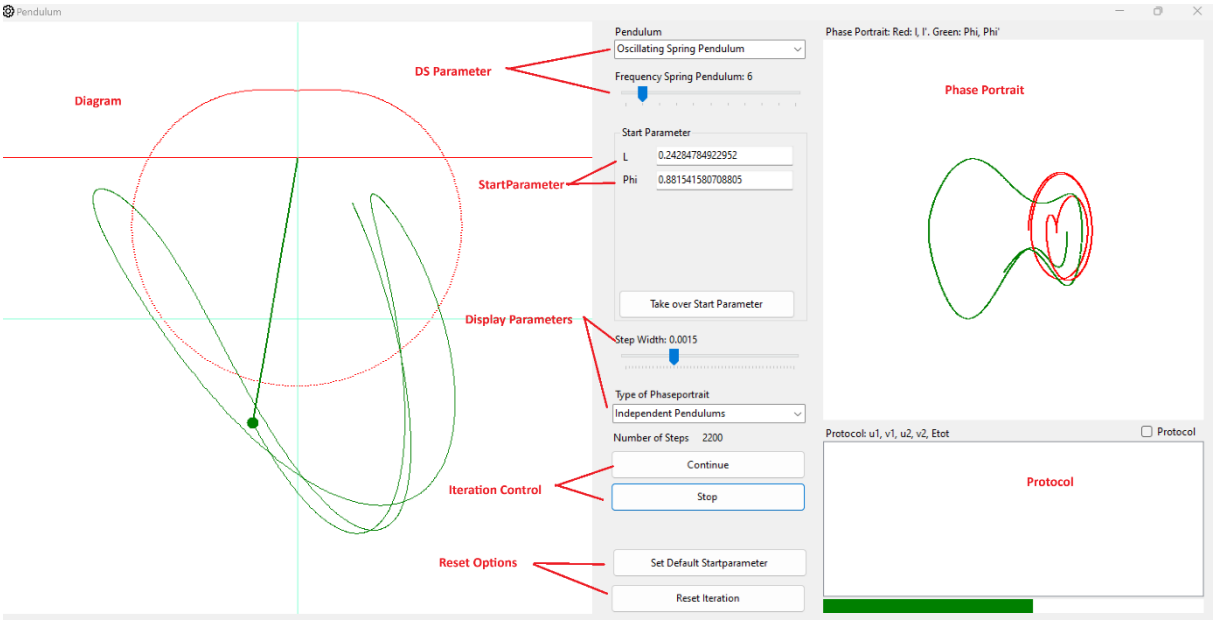
These are parameters that relate to the type of display. For example: a display should appear in colour, or a log should be shown/hidden. Or the speed of billiard balls should be set.

These parameters are also set to a default at the start. However, they remain when the *SetDefaultUserData* method is called.

See also the "Form Controller" section below.

1.9 User interface: Display and function

A typical user window then has this structure:



Typical structure of a user window

The role of the start parameters is important. These can be set in various ways:

- Set Default Startparameter
- Manual entries
- Set the start position with the mouse

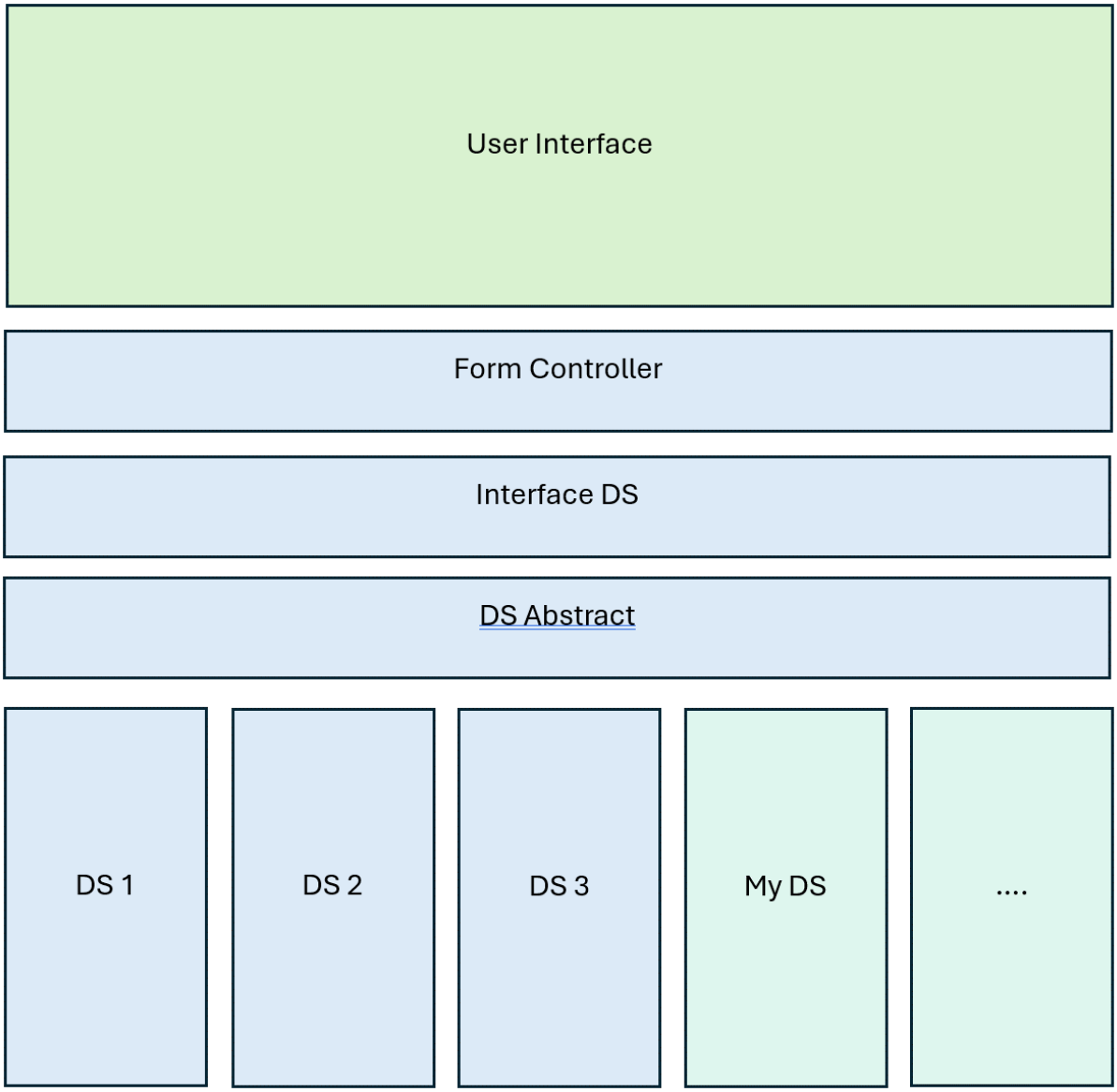
When the iteration is started, these values are transferred to the dynamic system.

The function of the "Take over Startparameters" button is to adjust the start position of the dynamic system in the diagram accordingly if the parameters are entered manually.

2. Architecture

2.1 Concept

The responsibilities in the "Simulator" are distributed according to the diagram below:



Architecture of the simulator

2.2 User interface: Logic

The user interface is a Windows form and only holds the logic that is directly related to controlling the controls in the form. This also includes their labeling, which is implemented in each form in the method

InitializeLanguage

To prevent unwanted "events" from being generated during loading, such as *TxtChanged*, each form has a Boolean variable *IsFormLoaded*. This is set to *false* at the start of the loading process and to *true* once the loading process is complete. The system checks whether the loading process has been completed for all form events.

In addition, the form must instantiate "its" form controller when loading, which is described in the next section.

This then leads to the following typical code in the form:

```
Public Class FrmFeigenbaum

    Private IsFormLoaded As Boolean
    Private FC As ClsFeigenbaumController
    Private LM As ClsLanguageManager

    'SECTOR INITIALIZATION
    0 Verweise
    Public Sub New()
        'This is necessary for the designer
        InitializeComponent()
        LM = ClsLanguageManager.LM
    End Sub

    0 Verweise
    Private Sub FrmFeigenbaum_Load(sender As Object, e As EventArgs) Handles Me.Load
        IsFormLoaded = False
        FC = New ClsFeigenbaumController(Me)

        'Initialize Language
        InitializeLanguage()

        FC.FillDynamicSystem()
    End Sub

    0 Verweise
    Private Sub FrmFeigenbaum_Shown(sender As Object, e As EventArgs) Handles Me.Shown
        FC.SetDS()
        IsFormLoaded = True
    End Sub
```

And

```
0 Verweise
Private Sub FrmFeigenbaum_Shown(sender As Object, e As EventArgs) Handles Me.Shown
    FC.SetDS()
    IsFormLoaded = True
End Sub

1 Verweis
Private Sub InitializeLanguage()

    Text = LM.GetString("FeigenbaumDiagram")
    ChkColored.Text = LM.GetString("ColoredDiagram")
    ChkSplitPoints.Text = LM.GetString("ShowSplitPoints")
    LblDeltaX.Text = LM.GetString("Delta") & " = "
    LblDeltaA.Text = LM.GetString("Delta") & " = "
    LblValueRange.Text = LM.GetString("ExaminatedValueRange")
    LblParameterRange.Text = LM.GetString("ExaminatedParameterRange")
    BtnStartIteration.Text = LM.GetString("StartIteration")
    BtnReset.Text = LM.GetString("ResetIteration")
    BtnDefault.Text = LM.GetString("DefaultUserData")

End Sub

0 Verweise
Private Sub BtnReset_Click(sender As Object, e As EventArgs) Handles BtnReset.Click
    If IsFormLoaded Then
        FC.ResetIteration()
    End If
End Sub
```

The form controller is designated *FC* in each form.

2.3 Graphics in the user interface

When displaying the movement of a dynamic system, the current system state should be displayed on the one hand (this changes continuously) and the "track" of the movement should be recorded on the other. The solution is to draw the current system state in the PictureBox *PicDiagram* and the

permanent track in a Bitmap *BmpDiagram*. Both must have the same size. The *Image* property of *PicDiagram* then refers to *BmpDiagram* so that both are always displayed simultaneously.

An instance of the ClsGraphicTool *PicGraphics* is then responsible for drawing in *PicDiagram* and the instance *BmpGraphics* for drawing in *BmpDiagram*.

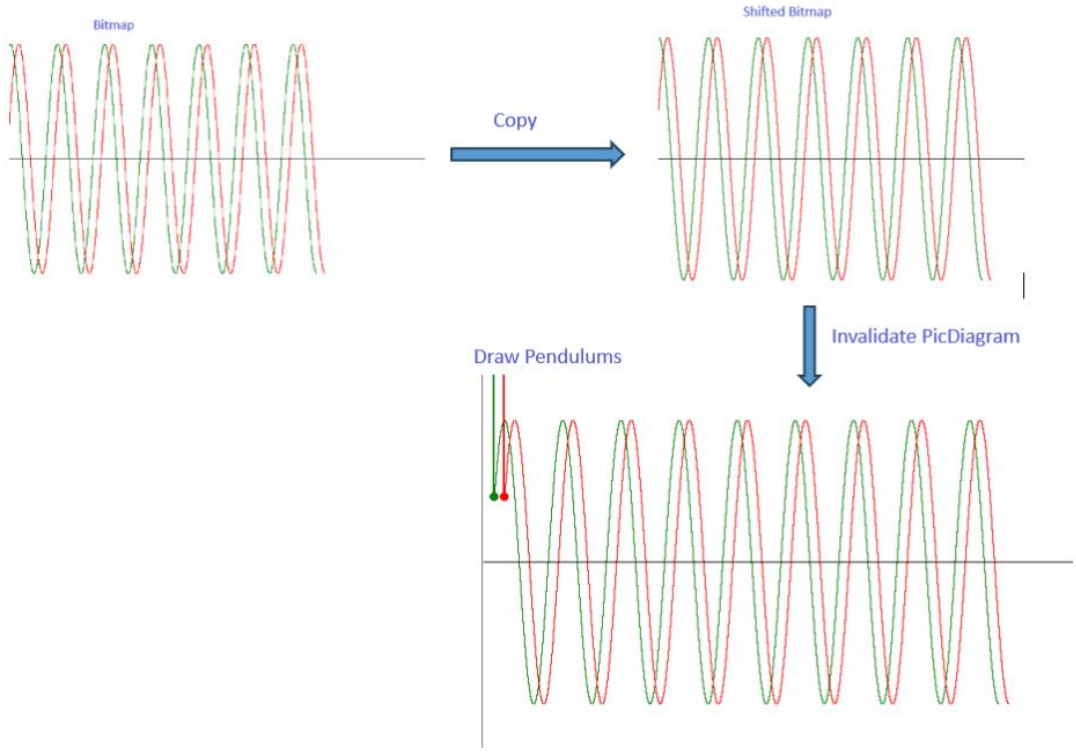
This leads to the following code example for *ClsPendulumAbstract*:

```
3 Verweise
WriteOnly Property PicDiagram As PictureBox Implements IPendulum.PicDiagram
    Set(value As PictureBox)
        MyPicDiagram = value

        'MyPicDiagram should be a square
        Dim Squareside As Integer = Math.Min(MyPicDiagram.Width, MyPicDiagram.Height)
        MyPicDiagram.Width = Squareside
        MyPicDiagram.Height = Squareside

        BmpDiagram = New Bitmap(Squareside, Squareside)

        'The Bitmap MapPendulum is then shown as Image of PicPendulum
        MyPicDiagram.Image = BmpDiagram

        'Graphics
        'The MathInterval is the same for X and Y
        PicGraphics = New ClsGraphicTool(MyPicDiagram, MathInterval, MathInterval)
        BmpGraphics = New ClsGraphicTool(BmpDiagram, MathInterval, MathInterval)

    End Set
End Property
```

If the mathematical definition area changes (e.g. through the *ClsDiagramAreaSelector*), the corresponding parameters of *PicGraphics, BmpGraphics* must also be changed.

When examining the numeric methods in the *NumericMethodClassLibrary* folder, the movement of a pendulum on the time axis, which shifts to the right, must be displayed. This is done by copying the *BmpDiagram* and moving it one step to the right and inserting the copy back into the *BmpDiagram*.

The idea is this:



The current bitmap is copied and then emptied. The copy of the bitmap is then inserted back into the original bitmap, but with a 1-pixel shift to the right. *PicDiagram.Refresh* transfers the shifted bitmap to the PicDiagram. The current position of the pendulums is then drawn into the PicDiagram.

This leads to the following code in *ClsNumericMethodsController*:

```
        NextTraceA = New ClsMathpoint(XPositionA, DSA.ActualParameter(1))
        NextTraceB = New ClsMathpoint(XPositionB, DSB.ActualParameter(1))

        'Draw the line of movement into the bitmap
        BmpGraphics.DrawLine(TraceA, NextTraceA, Color.Green, 1)
        BmpGraphics.DrawLine(TraceB, NextTraceB, Color.Red, 1)

        'copy the bitmap
        ShiftedBmpDiagram = New Bitmap(BmpDiagram)

        If n Mod NumberOfStepsUntilShift = 0 Then

            'Draw this copy right-shifted into the bitmap
            BmpGraphics.Clear(Color.White)
            BmpGraphics.DrawImage(ShiftedBmpDiagram, XShift, 0)

            'Coordinate system x-axis
            BmpGraphics.DrawLine(New ClsMathpoint(-1, 0), New ClsMathpoint(CDec(-0.9), 0), Color.Black, 1)

            'update Picdiagram
            MyForm.PicDiagram.Refresh()
        End If
```

2.4 General Forms

The general Windows forms are stored under *GeneralForms*:

- *FrmMain* is the form that is loaded when the program is started. It contains all menu items.
- *FrmTest* is a form that is not visible by default. It is used for testing purposes when developing new dynamic systems or new representations of these.
- *FrmInfo* shows the user information on the current version of the "Simulator".

2.5 Form Controller

A form controller (e.g. *FrmBilliardController*) contains the control logic for the form. It has a reference to the form (see *Load* event in the code examples) and therefore also to all elements of the form.

The form controller ensures the typical process when loading a form and reacts to user actions. The loading process is explained in the following section.

The form controller also controls the iteration process. This is explained in the following section.

2.6 Interface DS and DS Abstract

The DS interface is the link between the Form Controller and the dynamic system. It supplies all user specifications "downwards" and all relevant data for the form controller "upwards". It also provides the methods for iteration and usually also for setting the start position.

Example: *INumericMethod*

```
Public Interface INumericMethod

    'step width for each approximation step
    'this is set before the approximation starts
    'by the User Interface
    9 Verweise
    Property h As Decimal

    'number of approximation steps before result is returned
    'This number is set by the User Interface
    5 Verweise
    WriteOnly Property NumberOfApproxSteps As Integer

    'The amplitude is constant and set at the startposition
    6 Verweise
    WriteOnly Property Amplitude As Decimal

    'The ActualParameter holds the information about the
    'y-position of the Pendulum in its Component(1)
    'and the "time" t in its Component(0)
    'and additional values like the derivate y' in Component(2)
    15 Verweise
    Property ActualParameter(Index As Integer) As Decimal

    'The variable Parameters are changed during the iteration
    'Iteration performs one approximation step
    9 Verweise
    Sub Iteration()

End Interface
```

The specific dynamic systems of a category implement "their" interface. However, they all have common code, at least the setting of parameters through the properties interface. There are also common objects such as the Language Manager.

Example: Code excerpt from *ClsNumericMethodsAbstract*

```
0 Verweise
Public Sub New()
    LM = ClsLanguageManager.LM
End Sub
9 Verweise
Property h As Decimal Implements INumericMethod.h
    Get
        h = MyH
    End Get
    Set(value As Decimal)
        MyH = value
    End Set
End Property

5 Verweise
WriteOnly Property NumberOfApproxSteps As Integer Implements INumericMethod.NumberOfApproxSteps
    Set(value As Integer)
        MyNumberOfApproxSteps = value
    End Set
End Property

6 Verweise
WriteOnly Property Amplitude As Decimal Implements INumericMethod.Amplitude
    Set(value As Decimal)
        MyAmplitude = value
        MyActualParameter.Component(1) = value
    End Set
End Property
```

2.7 The typical charging process

The various dynamic systems that are offered in a form are listed in a ComboBox. This must be selected when the form is loaded. The form calls a corresponding routine in the form controller, e.g:

```
0 Verweise
Private Sub FrmJuliaSet_Load(sender As Object, e As EventArgs) Handles Me.Load

    'Generate objects
    IsFormLoaded = False
    FC = New ClsJuliaIterationController(Me)

    'Initialize Language
    InitializeLanguage()
    FC.FillDynamicSystem()
End Sub
```

And in the form controller you then have:

```
1 Verweis
Public Sub FillDynamicSystem()

    MyForm.CboFunction.Items.Clear()

    'Add the classes implementing IPolynom
    'to the Combobox CboFunction by Reflection
    Dim types As List(Of Type) = Assembly.GetExecutingAssembly().GetTypes().
                                 Where(Function(t) t.GetInterfaces().Contains(GetType(IJulia)) AndAlso
                                 t.IsClass AndAlso Not t.IsAbstract).ToList()

    If types.Count > 0 Then
        Dim JuliaName As String
        For Each type In types

            'GetString is calle dwith the option IsClass = true
            'That effects that - if there is no Entry in the Resource files LabelsEN, LabelsDE -
            'the name of the Class implementing an Interface is used as default
            'suppressing the extension "Cls"
            JuliaName = FrmMain.LM.GetString(type.Name, True)
            MyForm.CboFunction.Items.Add(JuliaName)
        Next
    Else
        Throw New ArgumentNullException("MissingImplementation")
    End If
    MyForm.CboFunction.SelectedIndex = 0

End Sub
```

As you can see, the "reflection" technique is used here.

The form controller then has a reference to the form: *MyForm*. It then also knows the selected dynamic system, which is referred to as *DS* in each Form Controller. Depending on the context, the Form Controller also "needs" a *DiagrammAreaSelector*. Example:

```
Imports System.Globalization
Imports System.Reflection

3 Verweise
Public Class ClsNewtonIterationController

    Private DS As INewton

    Private MyForm As FrmNewtonIteration
    Private DiagramAreaSelector As ClsDiagramAreaSelector
    Private LM As ClsLanguageManager

    1 Verweis
    Public Sub New(Form As FrmNewtonIteration)
        MyForm = Form
        LM = ClsLanguageManager.LM
        DiagramAreaSelector = New ClsDiagramAreaSelector
    End Sub
```

2.8 Selection of a dynamic system by the user

When the form is loaded, a dynamic system is set as the default. This leads to the same process as when the user selects a dynamic system themselves.

The corresponding methods of the controller are called:

- *SetDS*: The dynamic system is selected and instantiated here. Example:

```
Public Sub SetDS()

    'This sets the type of Polynom by Reflection

    Dim types As List(Of Type) = Assembly.GetExecutingAssembly().GetTypes().
                            Where(Function(t) t.GetInterfaces().Contains(GetType(INewton)) AndAlso
                            t.IsClass AndAlso Not t.IsAbstract).ToList()

    If MyForm.CboFunction.SelectedIndex >= 0 Then

        Dim SelectedName As String = MyForm.CboFunction.SelectedItem.ToString

        If types.Count > 0 Then
            For Each type In types
                If FrmMain.LM.GetString(type.Name, True) = SelectedName Then
                    DS = CType(Activator.CreateInstance(type), INewton)
                End If
            Next
        End If

    End If

    InitializeMe()

    SetDefaultUserData()

    ResetIteration()

End Sub
```

Each time the dynamic system is selected, it must then be initialized together with other objects of the form (e.g. *ClsDiagramAreaSelector*). This is done using the *InitializeMe* method. Example *ClsJuliaIterationController*:

```
Private Sub InitializeMe()

    'The following order is important
    'because changing .N
    'uses e.g. TxtNumberOfSteps
    With DS
        .PicDiagram = MyForm.PicDiagram
        .TxtNumberOfSteps = MyForm.TxtSteps
        .TxtElapsedTime = MyForm.TxtTime
        MyForm.CboN.SelectedIndex = 0
        .N = CInt(MyForm.CboN.SelectedItem)
        .ActualXRange = .XValueParameter.Range
        .ActualYRange = .YValueParameter.Range
        .ProcotolList = MyForm.LstProtocol
        .IsProtocol = MyForm.ChkProtocol.Checked
        .RedPercent = MyForm.TrbRed.Value / 10
        .GreenPercent = MyForm.TrbGreen.Value / 10
        .BluePercent = MyForm.TrbBlue.Value / 10
    End With

    With DiagramAreaSelector
        .XRange = DS.XValueParameter.Range
        .YRange = DS.YValueParameter.Range
        .PicDiagram = MyForm.PicDiagram
        .TxtXMin = MyForm.TxtXMin
        .TxtXMax = MyForm.TxtXMax
        .TxtYMin = MyForm.TxtYMin
        .TxtYMax = MyForm.TxtYMax
    End With

End Sub
```

As the dynamic system has been reloaded, the default user data must be set as the start data for the dynamic system. This is done in the *SetDefaultUserData* method. Example in *ClsJuliaIterationController*:

```
Public Sub SetDefaultUserData()
    With MyForm
        .TxtXMin.Text = DS.XValueParameter.Range.A.ToString(CultureInfo.CurrentCulture)
        .TxtXMax.Text = DS.XValueParameter.Range.B.ToString(CultureInfo.CurrentCulture)
        .TxtYMin.Text = DS.YValueParameter.Range.A.ToString(CultureInfo.CurrentCulture)
        .TxtYMax.Text = DS.YValueParameter.Range.B.ToString(CultureInfo.CurrentCulture)
    End With
    SetDelta()
End Sub
```

As an iteration may have been performed previously, this must be reset. This is done in the *ResetIteration* method. This is often passed on to the dynamic system, depending on the level at which the iteration is controlled. Example in *ClsJuliaIterationController*:

```
7 Verweise
Public Sub ResetIteration()
    MyForm.BtnStart.Text = LM.GetString("Start")
    DS.ResetIteration()
End Sub
```

And then in *DS* (here in *ClsJuliaAbstract*):

```
3 Verweise
Public Sub ResetIteration() Implements IJulia.ResetIteration

    'Clear MapCPlane

    BmpGraphics.Clear(Color.White)
    DrawCoordinateSystem()


    MyTxtNumberofSteps.Text = "0"
    MyTxtElapsedTime.Text = "0"
    L = 0
    Watch.Reset()
    ExaminatedPoints = 0

    'Clear Protocol
    If MyProtocolList IsNot Nothing Then
        MyProtocolList.Items.Clear()
    End If

    MyPicDiagram.Refresh()

    MyIterationStatus = ClsDynamics.EnIterationStatus.Stopped

End Sub
```

These methods

- *SetDS*
- *InitializeMe*
- *SetDefaultUserData*
- *ResetIteration*

Are named the same in all Form Controllers and processed according to the same process.

3. Iteration control

3.1 Start, interruption, stop

Each form controller manages the state of the iteration. The possible states are listed as an enumeration in the *ClsDynamics* class.

```
99+ Verweise
Public Class ClsDynamics

    'Status of the Iteration
    99+ Verweise
    Public Enum EnIterationStatus
        Running
        Interrupted
        Stopped
        Ready
    End Enum


End Class
```

Before an iteration is started for the first time, it is in the state

Stopped

If it is started in this state, it must first be checked whether the user input is within the permitted ranges.

IsUserDataOK

Which returns true or false.

If the user input is OK, the status of the iteration is set to

Ready

In this state, the iteration starts and receives the status

Running

It then either runs through a predefined number of steps and is then terminated, in which case it returns to the *Stopped* state. Or it runs until the user interrupts it by pressing the stop button. It then receives the status

Interrupted

This leads to the following code (e.g. in *ClsPendulumController*):

```
1 Verweis
Public Async Sub StartIteration()

    'UserData are always OK
    If IterationStatus = ClsDynamics.EnIterationStatus.Stopped Then
        If IsUserDataOK() Then
            With DS
                .IsStartparameter1Set = True
                .IsStartparameter2Set = True
                IterationStatus = ClsDynamics.EnIterationStatus.Ready
                StartEnergy = .GetEnergy
            End With
        End If
    End If

    If IterationStatus = ClsDynamics.EnIterationStatus.Ready _
        Or IterationStatus = ClsDynamics.EnIterationStatus.Interrupted Then
        With MyForm
            .BtnStart.Text = LM.GetString("Continue")
            .BtnStart.Enabled = False
            .BtnDefault.Enabled = False
            .BtnReset.Enabled = False
            .BtnTakeOverStartParameter.Enabled = False
            .TrbAdditionalParameter.Enabled = False
        End With

        IterationStatus = ClsDynamics.EnIterationStatus.Running

        Application.DoEvents()
        Await IterationLoop(False)
    End If
End Sub
```

And

```
1 Verweis
Public Sub StopIteration()
    'the iteration was running and is interrupted
    IterationStatus = ClsDynamics.EnIterationStatus.Interrupted
    'the iteration is stooped by reset the iteration
    With MyForm
        .BtnStart.Enabled = True
        .BtnReset.Enabled = True
        .BtnDefault.Enabled = True
        .BtnTakeOverStartParameter.Enabled = True
        .TrbAdditionalParameter.Enabled = True
    End With
End Sub
```

If the iteration is reset by *ResetIteration*, all graphics and status or log fields are cleared. However, the user data remains. This affects both the controller and possibly also the dynamic system. Therefore, both have a reset iteration method. A typical code snippet is as follows (in *ClsPendulumController*):

```
3 Verweise
Public Sub ResetIteration()
    'Clear Diagram and Bitmap and all Iteration Parameters in DS
    DS.ResetIteration()
    MyForm.BtnStart.Text = LM.GetString("Start")
    MyForm.BtnTakeOverStartParameter.Enabled = True
    MyForm.TrbAdditionalParameter.Enabled = True

    N = 0
    MyForm.LblSteps.Text = "0"
    StartEnergy = 0

End Sub
```

And in the DS *ClsPendulumAbstract*:

```
6 Verweise
Public Sub ResetIteration() Implements IPendulum.ResetIteration

    MyProtocol.Items.Clear()

    BmpGraphics.Clear(Color.White)
    BmpPhaseportraitGraphics.Clear(Color.White)

    MyPicDiagram.Refresh()
    MyPicPhaseportrait.Refresh()

    PrepareDiagram()

    MyIterationStatus = ClsDynamics.EnIterationStatus.Stopped

    MyIsStartParameter1Set = False
    MyIsStartParameter2Set = False

End Sub
```

If the user wants to start a new iteration with the same dynamic system and the same user data, he can provide this with *ResetIteration*.

However, sometimes you also want to set the user data to the default. This is done using the *ResetIteration* method followed by *SetDefaultUserData*. This can affect both the controller and the dynamic system. Example code in *ClsPendulumController*:

```
2 Verweise
Public Sub SetDefaultUserData()
    With DS
        .SetDefaultUserData()
        .PrepareDiagram()

        Dim ConstantsDimension As Integer
        If .CalculationConstants IsNot Nothing Then
            ConstantsDimension = .CalculationConstants.Dimension
            For i = 0 To ConstantsDimension
                MyForm.GrpStartParameter.Controls.Item("TxtP" & (i + 1).ToString).Text =
                    .CalculationConstants.Component(i).ToString
            Next
        Else
            ConstantsDimension = -1
        End If

        For i = 0 To .CalculationVariables.Dimension
            MyForm.GrpStartParameter.Controls.Item("TxtP" & (i + 2 + ConstantsDimension).ToString).Text =
                .CalculationVariables.Component(i).ToString
        Next
    End With

End Sub
```

And in DS *ClsDoublePendulum* (Overrides *SetSefaultUserData* from *ClsPendulumAbstract*):

```
4 Verweise
Protected Overrides Sub SetDefaultUserData()

    'Standardvalues
    With MyCalculationConstants
        .Component(0) = ValueParameter(0).DefaultValue  'L1
        .Component(1) = ValueParameter(1).DefaultValue  'L2
    End With

    With MyCalculationVariables
        .Component(0) = ValueParameter(2).DefaultValue 'Phi1
        u1 = .Component(0)
        v1 = 0
        .Component(1) = ValueParameter(3).DefaultValue 'Phi2
        u2 = .Component(1)
        v2 = 0
    End With

    SetStartEnergyRange()
    SetPosition()

End Sub
```

3.2 Different control levels of iteration

Depending on the dynamic system, an iteration has 2-3 control levels. At the lowest level, there is always a single iteration step:

IterationStep

This method is always implemented at the lowest level in the specific dynamic system.

The loop is then implemented one level higher by many such individual steps:

IterationLoop

It is possible that the loop is limited by a certain number of steps. For example, in the *ClsIterationController*, where you can perform one or 10 iteration steps, or generate an entire diagram of the length *PicDiagram.Width.*

It is also possible that the loop runs until the user presses a stop button, e.g. in the *ClsPendulumController*.

Sometimes these two levels are not enough; a third is needed:

PerformIteration

While *IterationStep* is always implemented at DS or DSAbstract level, *IterationLoop* and *PerformIteration* are always implemented at Form Controller level.

Example:

When generating the Feigenbaum diagram, the diagram must be run through at the highest level in the *PerformIteraton* method for each pixel point in the x-direction and thus for a fixed parameter a.

For each such value of a, a certain number of iteration steps must then be carried out in *IterationLoop* so that the function values occurring for this a can be plotted in the y direction. *IterationStep* is executed for each such iteration step.

Example code in *ClsIterationController*:

```
1 Verweis
Private Sub PerformIteration()

    'In the direction of the x-axis, we work with pixel coordinates
    Dim p As Integer

    For p = 1 To MyForm.PicDiagram.Width

        'for each p, the according parametervalue a is calculated
        'and then, the iteration runs until RuntimeUntilCycle
        'finally, the iteration cycle is drawn
        IterationLoop(p)

    Next

    'Draw Splitpoints
    If MyForm.ChkSplitPoints.Checked Then
        DrawSplitPoints()
    End If

End Sub
```

And one logical level deeper:

```
1 Verweis
Private Sub IterationLoop(p As Integer)
    If IterationStatus = ClsDynamics.EnIterationStatus.Ready Then
        'Initialize
        'enough but not bigger than the y-axis allows
        LengthOfCycle = 4 * MyForm.PicDiagram.Height
        'To draw the cycle
        CyclePoint = New ClsMathpoint(DS.ParameterA, x)
    End If
    IterationStatus = ClsDynamics.EnIterationStatus.Running
    'Calculate the parameter a for the iteration depending on p
    DS.ParameterA = ActualParameterRange.A + (ActualParameterRange.IntervalWidth * p / MyForm.PicDiagram.Width)
    CyclePoint.X = DS.ParameterA
    'Initialize Iteration
    'The startvalue x for the iteration should be the same for all values of a
    x = DS.CriticalPoint
    n = 1
    Do
        x = DS.FN(x)
        n += 1
    Loop Until (n > RunTimeUntilCycle - 1)
    n = RunTimeUntilCycle
    CyclePoint.Y = x
    Do
        BmpGraphics.DrawPoint(CyclePoint, SetColor(n), 1)
        x = DS.FN(x)
        CyclePoint.Y = Math.Max(ActualValueRange.A, Math.Min(x, ActualValueRange.B))
        n += 1
    Loop Until (n > RunTimeUntilCycle + LengthOfCycle)
    MyForm.PicDiagram.Refresh()
End Sub
```

And finally at the lowest level and in the specific DS (here in *ClsLogisticGrowth*):

```
2 Verweise
Protected Overrides Function F(x As Decimal) As Decimal

    'This is the original iteration function
    Return MyParameterA * x * (1 - x)

End Function
```

Here the *IterationStep* is referred to as the function *F(x).*

One question in this context is on which level the graphics *PicDiagram* or *BmpDiagram* should be drawn. This can vary depending on the dynamic system and the respective shape. In billiards, for example, any number of billiard balls can be placed on the table. In this case, it also makes sense for each ball to draw "its" path in the graphic. In this case, the graphic is accessed at *IterationStep* level. With the histogram, on the other hand, many iteration steps should be carried out and then a "sum" of the hits should be displayed in a small interval in the graphic. In this case, the graphic is accessed at *IterationLoop* level.

3.3 Asynchronous iterations and performance

The laptop on which the simulator was developed is a Lenovo with a 13th Gen Intel(R) Core(TM) i7-1370P 1.90 GHz processor. This uses several parallel cores. If a method is programmed asynchronously "outside" the main thread, it does not have to wait for any I/O operations or other events. The computing time can be used efficiently. This means, for example, that the computing time for generating a set of Julia (specifically the "seahorse") has been reduced from around 2 minutes to around 4.6 seconds.

The corresponding code is (in *ClsJuliaController*) at startup:

```
'SECTOR ITERATION
1 Verweis
Public Async Sub StartIteration()

    If IterationStatus = ClsDynamics.EnIterationStatus.Stopped Then

        'the iteration was stopped or reset
        'and should start from the beginning
        If IsUserDataOK() And IsCParameterOK() Then

            DiagramAreaSelector.IsActivated = False
            MyForm.BtnStart.Text = LM.GetString("Continue")
```

And further down in *StartIteration*:

```
    If IterationStatus = ClsDynamics.EnIterationStatus.Ready _
            Or IterationStatus = ClsDynamics.EnIterationStatus.Interrupted Then
        IterationStatus = ClsDynamics.EnIterationStatus.Running
        With MyForm
            .BtnStart.Enabled = False
            .BtnReset.Enabled = False
            .ChkProtocol.Enabled = False
            .BtnDefault.Enabled = False
        End With

        Await PerformIteration()
    End If

    If IterationStatus = ClsDynamics.EnIterationStatus.Stopped Then
        With MyForm
            .BtnStart.Text = LM.GetString("Start")
            .BtnStart.Enabled = True
            .BtnReset.Enabled = True
            .BtnDefault.Enabled = True
        End With
        DiagramAreaSelector.IsActivated = True
    End If
End Sub
```

The *PerformIteration* method is then implemented as a task:

```
1 Verweis
Public Async Function PerformIteration() As Task

    'This algorithm goes through the CPlane in a spiral starting in the midpoint
    If ExaminatedPoints = 0 Then
        p = CInt(MyForm.PicDiagram.Width / 2)
        q = CInt(MyForm.PicDiagram.Height / 2)

        PixelPoint = New Point

        With PixelPoint
            .X = p
            .Y = q
        End With

        DS.IterationStep(PixelPoint)

        Steps = 1
        Watch.Start()

    End If


    Do
        ExaminatedPoints += 1

        IterationLoop()
```

And transfers control to the *IterationLoop*:

```
    Do
        ExaminatedPoints += 1

        IterationLoop()


        If p >= MyForm.PicDiagram.Width Or q >= MyForm.PicDiagram.Height Then

            IterationStatus = ClsDynamics.EnIterationStatus.Stopped
            Watch.Stop()
            MyForm.PicDiagram.Refresh()

        End If

        If ExaminatedPoints Mod 100 = 0 Then
            MyForm.TxtSteps.Text = Steps.ToString
            MyForm.TxtTime.Text = Watch.Elapsed.ToString
            Await Task.Delay(1)
        End If

    Loop Until IterationStatus = ClsDynamics.EnIterationStatus.Interrupted _
        Or IterationStatus = ClsDynamics.EnIterationStatus.Stopped

End Function
```

Which is available as a "normal" method.

The *Await Task.Delay* instruction enables the main thread to react to the pressing of the stop button, for example.

What also leads to a significant reduction in performance is the updating of logs or entries in value lists. This is therefore optional. With the Newton iteration for the third roots of unity, the generation of the basins with a log takes over three minutes. Without a protocol, it takes less than 5 seconds!

3.4 Definition of the start parameters by the user

The user can define start parameters by making entries in the corresponding TextBoxes. A *BtnTakeOver* button transfers these entries directly to the dynamic system and, depending on the type of system, it is placed directly on the *PicDiagram* in this start position (e.g. for the pendulum). The parameters are transferred to the dynamic system again at the latest when the iteration is started.

If it makes sense in the context, an alternative is that the user can define the start position by holding down the left mouse button. This is the case with the pendulum, numerical methods and billiards. The corresponding text fields with the start parameters are then tracked directly and the dynamic system is placed in the *PicDiagram*.

If the iteration is in the *Stopped* state, pressing the left mouse button sets the *IsMouseDown* parameter to *true*. This enables the position of the dynamic system to be moved to the position of the mouse when the mouse is moved, provided that it has not already been positioned. This means that the *IsStartParameterSet* parameter is still set to *false*. The rules of the dynamic system are taken into account. For example, in billiards, the positioned billiard ball must always lie on the edge of the billiard table. If the mouse button is released, *IsStartParameterSet = true* and *IsMouseDown = false*). The *IsStartParameterSet* parameter is only set to *false* again when a new dynamic system is selected or when the iteration is reset (*ResetIteration*).

The following is a typical code example from *ClsPendulumController, IPendulum, ClsPendulumAbstract* and *ClsCombinedPendulum*

ClsPendulumController

```
1 Verweis
Public Sub MouseDown(e As MouseEventArgs)

    If IterationStatus = ClsDynamics.EnIterationStatus.Stopped Then
        If Not (DS.IsStartparameter1Set And DS.IsStartparameter2Set) Then

            MyForm.Cursor = Cursors.Hand
            IsMouseDown = True

            'Now, Moving the Mouse moves the active Pendulum
            MouseMoving(e)

        End If
    End If

End Sub
```

```
2 Verweise
Public Sub MouseMoving(e As MouseEventArgs)

    If IsMouseDown Then
        'Because the Cursor is "Hand", the Mouse Position is adjusted a bit
        Dim Mouseposition As New Point With {
        .X = e.X + 2,
        .Y = e.Y
    }

        Dim i As Integer

        If DS IsNot Nothing Then

            With DS
                If Not .IsStartparameter1Set Then

                    'The actual Position of the Mouse sets Parameter1
                    .SetAndDrawStartparameter1(Mouseposition)

                ElseIf Not .IsStartparameter2Set Then

                    'The actual Position of the Mouse sets Parameter2
                    .SetAndDrawStartparameter2(Mouseposition)

                End If

                Dim ConstantsDimension As Integer
```

```
                Dim ConstantsDimension As Integer

                If .CalculationConstants IsNot Nothing Then
                    ConstantsDimension = .CalculationConstants.Dimension
                    For i = 0 To .CalculationConstants.Dimension
                        MyForm.GrpStartParameter.Controls.Item("TxtP" & (i + 1).ToString).Text =
                            .CalculationConstants.Component(i).ToString
                    Next
                Else
                    ConstantsDimension = -1
                End If

                For i = 0 To .CalculationVariables.Dimension
                    MyForm.GrpStartParameter.Controls.Item("TxtP" & (i + 2 + ConstantsDimension).ToString).Text =
                        .CalculationVariables.Component(i).ToString
                Next

            End With

        End If
    End If
End Sub
```

```
1 Verweis
Public Sub MouseUp()
    'Has only an effect, if the Mouse was down
    If IsMouseDown Then

        With DS
            If Not .IsStartparameter1Set Then

                'The setting of Parameter1 i now blocked
                .IsStartparameter1Set = True

            ElseIf Not .IsStartparameter2Set Then

                'nothing
                'Startparameter2 is fixed when starting

            End If
        End With

        'The Mouse gets its normal behaviour again
        MyForm.Cursor = Cursors.Arrow
        IsMouseDown = False

    End If
End Sub
```

IPendulum

```
6 Verweise
Sub SetAndDrawStartparameter1(Mouseposition As Point)

6 Verweise
Sub SetAndDrawStartparameter2(Mouseposition As Point)
```

ClsPendulumAbstract

```
6 Verweise
Public MustOverride Sub SetAndDrawStartparameter1(Mouseposition As Point) _
    Implements IPendulum.SetAndDrawStartparameter1
'OK

6 Verweise
Public MustOverride Sub SetAndDrawStartparameter2(Mouseposition As Point) _
    Implements IPendulum.SetAndDrawStartparameter2
'OK
```

ClsCombinedPendulum

```
'SECTOR SETSTARTPARAMETER
4 Verweise
Public Overrides Sub SetAndDrawStartparameter1(Mouseposition As Point)

    Dim ActualPosition As ClsMathpoint = PicGraphics.PixelToMathpoint(Mouseposition)

    With ActualPosition

        .Y = .Y - Y0

        'Phi
        Dim Phi As Decimal = MathHelper.GetAngle(.X, .Y)
        Phi = MathHelper.AngleInMinusPiAndPi(Phi)

        'Lmax must be adapted depending on Phi
        MyValueParameterDefinition.Item(0) = New ClsGeneralParameter(1, "L",
                    New ClsInterval(CDec(0.2), CDec(0.95 + Y0 * Math.Cos(Phi))),
                    ClsGeneralParameter.TypeOfParameterEnum.Variable)

        'L should be in [MyValueParameters.Item(0).Range.A, MyValueParameters.Item(0).Range.B]
        Dim LocL As Decimal
        LocL = CDec(Math.Sqrt(.X * .X + .Y * .Y))
        LocL = Math.Max(MyValueParameterDefinition.Item(0).Range.A, LocL)
        LocL = Math.Min(LocL, CDec(0.95 + Y0 * Math.Cos(Phi)))


        'Set parameters
        MyCalculationVariables.Component(0) = LocL
        MyCalculationVariables.Component(1) = Phi
```

```
        'Set parameters
        MyCalculationVariables.Component(0) = LocL
        MyCalculationVariables.Component(1) = Phi

        SetStartEnergyRange()

        SetPosition()

        'Draw pendulum
        DrawPendulums()

        'Runge Kutta
        u1 = LocL
        v1 = 0
        u2 = Phi
        v2 = 0

    End With

End Sub

4 Verweise
Public Overrides Sub SetAndDrawStartparameter2(Mouseposition As Point)

    'nothing - the position is set on the first step
    'just implementing IPendulum
End Sub
```

4. Implementation of the Billiard

The *ClsBilliardTableController* class establishes the connection between the *FrmBilliardTable* user interface and the lower levels of the architecture. The *IBilliardTable* interface is located in between. It is implemented by the abstract class *IBilliardTableAbstract*. The concrete realization of a billiard inherits from this class and must overwrite certain of its methods.

The billiard table or the concretizations of the class *ClsBilliardTableAbstract* provide the billiard table. It can draw itself and generate a billiard ball that "fits" it. It then maintains a collection of all balls in *MyBilliardballCollection*.

The IBilliardBall interface is available for the billiard balls. It is implemented by the abstract class *ClsBilliardBallAbstract*. Concrete billiard balls inherit from this class and overwrite certain methods in it. In particular, a billiard ball performs:

- Start parameter and its start position
- Current parameters, in particular *t* and the impact angle α

It overwrites the following methods of *ClsBilliardBallAbstract*:

```
13 Verweise
MustOverride Property Startparameter As Decimal Implements IBilliardball.Startparameter

9 Verweise
MustOverride WriteOnly Property Startangle As Decimal Implements IBilliardball.Startangle

8 Verweise
Public MustOverride Sub IterationStep() Implements IBilliardball.IterationStep

7 Verweise
Public MustOverride Function SetAndDrawUserStartposition(Mouseposition As Point, IsDefinitive As Boolean) As Decimal _
    Implements IBilliardball.SetAndDrawUserStartposition

7 Verweise
Public MustOverride Function SetAndDrawUserEndposition(Mouseposition As Point, IsDefinitive As Boolean) As Decimal _
    Implements IBilliardball.SetAndDrawUserEndposition

6 Verweise
Public MustOverride Function GetNextValuePair(ActualPoint As ClsValuePair) As ClsValuePair _
    Implements IBilliardball.GetNextValuePair

6 Verweise
Public MustOverride Sub DrawFirstUserStartposition() _
    Implements IBilliardball.DrawFirstUserStartposition

12 Verweise
Public MustOverride Function CalculateAlfa(t As Decimal, phi As Decimal) As Decimal _
    Implements IBilliardball.CalculateAlfa
```

The parameter *C*, which defines the shape of the billiard table depending on the billiard, can be changed by the user in the trackbar *TrbParameterC*. The table is then immediately redrawn. This parameter is then decisive for displaying the transition from "well-behaved" behaviour to chaotic behaviour in the C diagram. It is a parameter of type *DS*.

4.1 Implementation of Elliptical Billiards

As angles between a vector and the positive x-axis always must be determined, this is supported by the *CalculateAngleOfDirection (DeltaX, DeltaY)* function in the *ClsMathHelperBilliard* class. The coordinates of the vector are transferred. The return value is an angle in $[0, 2\pi[$. The function is used to determine the angles φ, ψ, ϑ and the parameter *t*. See the corresponding chapter in the mathematical documentation.

The different ball class depending on the billiard (for elliptical billiards the class *ClsEllipseBilliardball)* contains the general logic for the ball movement. The ball path is continuously drawn in the bitmap *BmpDiagam*, which is the image of the PictureBox *PicDiagram*. The current position of the ball is drawn in *PicDiagram*. By refreshing the Picture Box, only this position of the ball is visible, while the bitmap including the ball path is visible at the current position. To support this, the sphere needs the references to *PicDiagram, BmpDiagram* and the associated graphic tools *PicGraphics, BmpGraphics*. Since the sphere works in mathematical coordinates and only the *ClsGraphicTool* class provides the conversion to pixel units, the latter also need the mathematical value ranges for x and y: *MathXInterval, MathYInterval*. This is the standard square [-1,1] x [-1,1].

When the iteration is started, the *ClsEllipseBilliardball* moves independently within the ellipse according to the mathematical algorithms described in the mathematical documentation. From the last impact point and the current impact direction, the ball first calculates the next impact point. It then calculates the tangent angle at the next impact point and from this the direction for the next impact.

From the equations

$$\varphi_{n+1} = \psi_{n+1} + \alpha_{n+1}$$

And

$$\alpha_{n+1} = \psi_{n+1} - \varphi_n$$

Results directly for the direction of the ball after the nth impact:

$$\varphi_{n+1} = 2\psi_{n+1} - \varphi_n$$

The "Simulator" works directly with this formula but logs the angle α_n together with the parameter t_n in the phase portrait. If *IsProtocol* is activated, these parameters are also written to the *MyValueProtocol* list box.

As the ball itself knows the rules for its movement, this makes it possible to instantiate several balls and run their movements in parallel. The balls are differentiated by their colours and five different colours can be selected. The ball speed can be changed and affects all balls.

The following code snippet shows the process during iteration:

```
Public Overrides Sub IterationStep()

    'Startpoint of the actual part of the Orbit
    Dim Startpoint As New ClsMathpoint
    'Parameter of the next Endpoint of the actual part of the Orbit
    Dim NextT As Decimal
    'and the according EndPoint
    Dim Endpoint As New ClsMathpoint

    'MyT is the Parameter ot the StartPoint of the actual part of the Orbit
    Startpoint.X = CDec(MyA * Math.Cos(T))
    Startpoint.Y = CDec(MyB * Math.Sin(T))

    'NextT is the Parameter of the EndPoint of the actual part of the Orbit
    NextT = ParameterOfNextHitPoint(T, Phi)
    Endpoint.X = CDec(MyA * Math.Cos(NextT))
    Endpoint.Y = CDec(MyB * Math.Sin(NextT))

    'The Ball moves between these Points
    MoveOnSegment(Startpoint, Endpoint)

    'The EndPoint is then the StartPoint of the following part of the Orbit
    T = NextT
    Startpoint.X = Endpoint.X
    Startpoint.Y = Endpoint.Y

    'in addition, we calculate the angle of the following movement
    Phi = CalculateNextPhi(T, Phi)

End Sub
```

The designations correspond to the mathematical documentation.

The specific calculations for the elliptical billiard ball can be found in:

- *ParameterOfNextHitpoint*
- *MoveOnSegment*
- *CalculateNextPhi* (this is also where the drawing in the diagrams takes place)
- *CalculatePsi*
- *CalculateAlfa*

The *GetNextValuePair* function is intended for the C diagram.

4.2 Implementation of the Oval Billiard

The only difference to elliptical billiards is the different mathematical formulas and the impact algorithms. When calculating the next impact point, a distinction must be made as to whether the ball hits the half ellipse on the left side or whether it hits the half of the ball on the right side.

4.3 Implementation of the Stadium Billiard

The only difference to elliptical billiards is the different mathematical formulas and the impact algorithms. The calculation of the next shot point is much more complex, however, because of the case distinctions that are necessary due to the different sections of the billiard table. There are four of them, depending on whether the next shot takes place in the left or right semicircle or on the lower or upper straight section of the billiard table. For a kick point to be determined at all, the first kick angle must not be 0 or π or

4.4 Implementation of the C-Diagram

The C-diagram shows different values of the DS parameter *C* along the x-axis. For each such parameter value, the billiard table takes on a certain shape. Of particular interest here is the oval billiard. Now, for each such C value, a billiard ball is always launched from the same starting point and at the same angle of reflection. For each shot, a pair of parameters (t, α) is returned for each shot. This is done by the method:

```
4 Verweise
Public Overrides Function GetNextValuePair(ActualPoint As ClsValuePair) As ClsValuePair

    T = ActualPoint.X
    Dim alfa As Decimal = ActualPoint.Y

    'first, we calculate the angle between tangent in the hit point
    'and the positive x-axis
    Dim psi As Decimal = Mathhelper.AngleInNullTwoPi(CalculatePsi(T))

    'Now the angle between the next moving-direction and the positive x-axis is:
    Phi = psi + alfa

    'Parameter of the next Enpoint of the actual part of the Orbit
    Dim NextT As Decimal = ParameterOfNextHitPoint(T, Phi)

    'in addition, we calculate the angle of the following movement
    Phi = CalculateNextPhi(NextT, Phi)

    alfa = CalculateAlfa(NextT, Phi)

    Dim NextPoint As New ClsValuePair(NextT, alfa)

    Return NextPoint

End Function
```

Before starting, the user has selected a parameter that they want to observe. As a result, the values of this parameter are plotted in the direction of the y-axis of the C diagram.

This also means that in this case the iteration takes place in three stages:

- *PerformIteration*: Go through all possible C-values on the y-axis
- *IterationLoop*: Perform a number of iteration steps for each C value and plot the result in the C diagram
- *IterationStep*: Perform a single step and return the next pair of parameters

The *ClsCDiagramController* also has an instance of the *ClsDiagramAreaSelector* so that you can zoom into the C diagram.

5. Implementation of Growth Models

5.1 N:M Relationship between growth models and forms of representation

The special thing about the growth models is that, on the one hand, various such models are available. Current:

- *ClsLogisticGrowth*
- *ClsParabola*
- *ClsTentMap*
- *ClsMandelbrotReal*

The mathematical description of these models can be found in the mathematical documentation.

In addition, several forms are available for the presentation of different aspects of these models:

- *FrmIteration*: Representation of the actual iteration and the function graph. Support for a predefined protocol and for transitivity.
- *FrmHistogram*: Representation of the histogram in the chaotic case
- *FrmSensitivity*: Support for the investigation of sensitivity
- *FrmTwoDimensions*: For display in two dimensions
- *FrmFeigenbaum*: For displaying the fig tree diagram in the chaotic case

This means that there is an n:m relationship between representation forms and growth models. This is also the reason why the control of the iteration must take place in the respective form controller (and cannot be moved to a lower level, e.g. *ClsGrowthModelAbstract*).

Each form therefore has "its" controller. All of this access the *IIteration* interface. This is implemented by the abstract class *ClsGrowthModelAbstract*. The concrete classes for the various models inherit from this class and overwrite certain methods in it:

```
5 Verweise
Protected MustOverride Sub InitializeIterator()

5 Verweise
Protected MustOverride Function F(x As Decimal) As Decimal

6 Verweise
Protected MustOverride Function IterationToTentmap(x As Decimal) As Decimal

6 Verweise
Protected MustOverride Function TentmapToIteration(u As Decimal) As Decimal
```

There are quite a few methods. Mainly the actual function rule, as well as the diffeomorphism to the tent mapping and its inverse.

To ensure that the designations are clear, the variables for the growth models (except for the tent illustration) are consistently labelled with (x, y) throughout. The variables for the tent mapping, on the other hand, are labelled with (u, v). This corresponds to the designations in the mathematical documentation.

For all calculations, we work in a coordinate system that is defined by a value range for the x-coordinate and y-coordinate.

$$x \in [x_{min}, x_{max}], y \in [y_{min}, y_{max}]$$

A point (x, y) in this coordinate system is represented by an object of the *ClsMathPoint* class.

To find a start value for a given protocol or a given target value, the corresponding conjugation transformations for tent mapping are used: First calculate the corresponding starting value for the tent mapping as described in the corresponding previous section on tent mapping. Then the initial value for the logistic growth or the normalized parabola is obtained by applying the corresponding conjugation transformation described in the sections on these iterations.

These are the methods

- *IterationToTentmap*
- *TentmapToIteration*

The *FrmFeigenbaum* supports the investigation of any cycles depending on *a* parameter *a* for the iteration. Here too, the *FrmFeigenbaum* uses the methods and properties of the *IIterator* interface. Classes that implement this interface then contain the specifics of the respective iteration.

The user can manually enter the parameter range in which *a* moves. They can also enter a value range for the x-values, which defines the range in which the x-values are considered. This leads to a scaling of the x-axis.

Both can also be done by selecting with the left mouse button pressed. This allows you to examine interesting areas in the fig tree diagram more closely. A detailed description can be found in the "Manual".

5.2 Implementation of the Feigenbaum Diagram

As with the C-diagram in billiards, the parameter *a* in the growth models determines whether the system behaves chaotically or not. The Feigenbaum diagram shows the transition from order to chaos. All possible parameter values of *a* are run through in the x-axis direction. For each such *a*, the iteration is always started with the same starting value $x1$. Then the iteration is allowed to run for a while until it has settled on an attractive cycle, if one exists at all. The iteration values and thus this cycle is then plotted in the y-direction in the diagram.

This also means that there are three levels of iteration:

- *PerformIteration*: Go through all possible a-values on the y-axis
- *IterationLoop*: Perform a number of iteration steps for each a-value and then plot the possibly existing cycle in the Feigenbaum diagram
- *IterationStep*: Perform a single step and return the next pair of parameters

The *ClsMandelbrotReal* model plays a special role. It is only used for representation in the Feigenbaum diagram so that it can be compared with the Mandelbrot set, which is described in the

section on complex iteration. Since the definition range of this model depends on the parameter a, the other forms of representation do not really make sense.

6. Implementation of the Complex Iteration

6.1 Implementation of the Newton iteration

In contrast to the previous dynamic systems, the trajectories of iteration points are not plotted here, but the catchment areas or basins of attractive fixed points are examined. As described in the mathematical documentation, zeros of complex polynomials are also (super-)attractive fixed points of the Newton iteration. The latter is implemented for three models:

- roots of unity or zeros of the polynomials $p_n(z) = z^n - 1$
- "Inverted" roots of unity: This is the behaviour of the iteration at the infinitely distant point (see math doc.)
- Zeros of a third-degree polynomial

The iteration process is then always the same: you go through all the points in the complex plane. For each point you iterate until it is clear against which fixed point the iteration converges, or whether it does not converge if you exceed a certain number of iteration steps. Details can be found in the math. Documentation.

The *ClsNewtonIterationController* ensures the connection from the user interface to the lower levels and controls the iteration process, among other things. It is based on the *INewton* interface. This is implemented by the *ClsNewtonAbstract* class. The concrete classes (i.e. *ClsUnitRootCollection* and *ClsPolynom3*) inherit from this class and overwrite the explicit formulas of the iteration:

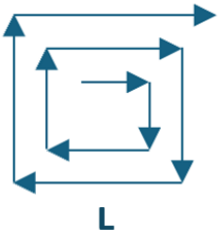
```
9 Verweise
MustOverride ReadOnly Property IsShowBasin As Boolean Implements INewton.IsShowBasin

4 Verweise
Public MustOverride Function StopCondition(Z As ClsComplexNumber) As Boolean

4 Verweise
Public MustOverride Function Newton(Z As ClsComplexNumber) As ClsComplexNumber

6 Verweise
Protected MustOverride Sub PrepareUnitRoots() Implements INewton.PrepareUnitRoots

8 Verweise
Public MustOverride Function Denominator(Z As ClsComplexNumber) As ClsComplexNumber
```

Here too, iteration takes place on three levels:

- *PerformIteration*: Determine a "corner" to be examined in the complex plane (see explanation below). These are two sides of a rectangle in the complex plane.
- *IterationLoop*: Examine each point on these rectangle sides by selecting the point *z* as the start value for the Newton iteration.
- *IterationStep*: Perform as many iteration steps for this starting value *z* until it becomes clear to which zero the iteration converges (if at all). Criteria for this are derived in the mathematical documentation. The starting point is then given the same color as "its" zero.

PerformIteration and *IterationLoop* are located in *ClsNewtonIterationController*. *IterationStep* and *ClsNewtonAbstract*.

The idea is that you do not go through the section of the complex plane line by line from right to left and top to bottom, because the interesting sections are in the middle of the diagram. The following algorithm was therefore chosen:



L

The iteration starts in the middle and then moves one step to the right and downwards. Then the length of the rectangle is increased by +1 and you first go to the left and then up. Then the length of the rectangle is increased again by +1 and you move to the right and then down. And so on.

Two consecutive rectangle pages (= an *ExamineCorner*) are one step in *PerformIteration*. Here is a code snippet from this method.

```
Do
    ExaminatedCorner += 1

    IterationLoop()

    If p >= MyForm.PicDiagram.Width Or q >= MyForm.PicDiagram.Height Then

        IterationStatus = ClsDynamics.EnIterationStatus.Stopped
        Watch.Stop()
        MyForm.PicDiagram.Refresh()

    End If

    If ExaminatedCorner Mod 100 = 0 Then
        MyForm.TxtSteps.Text = Steps.ToString
        MyForm.TxtTime.Text = Watch.Elapsed.ToString
        Await Task.Delay(1)
    End If

Loop Until IterationStatus = ClsDynamics.EnIterationStatus.Interrupted _
    Or IterationStatus = ClsDynamics.EnIterationStatus.Stopped
```

An *iteration loop* is the processing of two consecutive rectangle sides. This means that you go through each point of a rectangle side and select it as the starting point for the iteration. Here is a code snippet from this method.

```
1 Verweis
Private Sub IterationLoop()

    If ExaminatedCorner Mod 2 = 0 Then
        Sig = -1
    Else
        Sig = 1
    End If

    L += 1

    k = 1
    Do While k < L
        p += Sig
        With PixelPoint
            .X = p
            .Y = q
        End With

        'Calculates the color of the PixelPoint
        'and draws it to MapCPlane
        DS.IterationStep(PixelPoint)

        If Steps Mod 10000 = 0 Then
            MyForm.PicDiagram.Refresh()
        End If

        Steps += 1
        k += 1
    Loop
```

An *IterationStep* here is the iteration of a starting point *z* until its convergence is clarified. A code snippet from this method.

```
5 Verweise
Public Sub IterationStep(Startpoint As Point) Implements INewton.IterationStep

    'Transform the PixelPoint to a Complex Number
    Dim MathStartpoint As ClsComplexNumber

    With BmpGraphics.PixelToMathpoint(Startpoint)
        'Saved for debugging
        MathStartpoint = New ClsComplexNumber(.X, .Y)
    End With

    Dim Zi As New ClsComplexNumber(MathStartpoint.X, MathStartpoint.Y)

    Dim MyBrush As Brush = Brushes.White

    If Zi.AbsoluteValue > 0 Then

        'Protocol of the startpoint as Pixel and as Mathpoint
        'MyProtocol.Items.Add(Startpoint.X.ToString & ", " & Startpoint.Y.ToString & ", " &
        'Zi.X.ToString("N5") & ", " & Zi.Y.ToString("N5"))

        Dim i As Integer = 1


        Do While i <= IterationDeepness And Not StopCondition(Zi)

            i += 1

            'Calculate Next Z
            Zi = Newton(Zi)
```

```
            'Calculate Next Z
            Zi = Newton(Zi)

        Loop

        If (i > IterationDeepness) Or (Denominator(Zi).AbsoluteValue = 0) Then

            'the point doesn't converge to a root
            MyBrush = Brushes.Black
        Else
            'the basin defines the type of color
            'and i influences its brightness
            MyBrush = GetBasin(Zi, i)
        End If
    Else

        MyBrush = Brushes.Black
    End If

    BmpGraphics.DrawPoint(Startpoint, MyBrush, 1)

    'Protocol of the PixelStartpoint and the Endpoint as Mathpoint
    If MyIsProtocol Then
        MyProtocolList.Items.Add(MathStartpoint.X.ToString("N5") & ", " &
                                 MathStartpoint.Y.ToString("N5") &
            ", " & Zi.X.ToString("N5") & ", " & Zi.Y.ToString("N5"))
    End If
End Sub
```

The *GetBasin* method is responsible for colouring the basins of the zeros. It ensures that the colouring also depends on the number of steps until convergence to zero is ensured. The corresponding calculations were optimized experimentally. Here is an excerpt of the relevant code:

```
    If MyN = 2 Then
        MyColorDeepness = 30
    Else
        MyColorDeepness = 5 * MyN * MyN - 15 * MyN + 30
    End If

    FinalBrightness = (1 - Steps / MyColorDeepness) * 1.1

    'but the maximum is 1
    FinalBrightness = Math.Min(FinalBrightness, 1)
End If

For Each Root As ClsUnitRoot In UnitRootCollection
    Difference = Z.Add(Root.Stretch(-1)).AbsoluteValue
    If Difference < Temp Then
        Temp = Difference
        RootBrush = Root.GetColor(FinalBrightness)
    End If
Next

Return RootBrush
```

The colour of the *brush* returned by *Root.GetColor* has been systematically defined: The red/blue/green parts of the colour are graded according to the index of the unit root and this depends on the exponent *n.* These colours are chosen so that, for example, the 12th roots of unity have colours that match each other in order.

The *ClsComplexNumber* class implements certain operations with complex numbers, such as their multiplication, addition or the formation of their nth power, so that it is easier to calculate with

complex numbers without always having to resort to their real and imaginary parts. This allows algebraic expressions to be coded using a simple type of parser. For example, the code for the Newton iteration of the roots of unity:

$$N_p(z) := \frac{n-1}{n} z + \frac{1}{n z^{n-1}}$$

Coded:

Denominator = Z.Power(n - 1).Invers.Stretch(1/n)

Newton = Z.Stretch((n-1)/n).Add(Denominator)

6.2 Implementation of the Julia and Mandelbrot set

The controller for the *FrmJulia* is the *ClsJuliaIterationConrtoller*. It communicates with the lower layers via the *IJulia* interface. This is implemented by the abstract class *ClsJuliaAbstract*. The concrete realizations of the Julia set and Mandelbrot set inherit from this class and only overwrite one specific method: *IterationStep*.

The generation of the Julia and Mandelbrot sets follows the same scheme as the Newton iteration. The algorithm to examine the section of the complex plane is the same: The start is in the centre of the diagram and then it is "rolled up" in a spiral.

The available operations of the *ClsComplexNumber* class are also essential here for a simple code when calculating with complex numbers.

The colouring of the Julia or Mandelbrot set is special here. The colour depends on how fast a starting point moves towards ∞ strives. In other words, after how many iteration steps it is clear that it is doing so. Here is a code excerpt from *ClsJuliaPN*, i.e. the iterated polynomial for the Julia set with the exponent *n*:

```
If MyIsUseSystemColors Then
    MyBrush = StandardColors.GetSystemBrush(Steps)
    ColorIndex = 1
Else

    ColorIndex = Steps / MaxSteps

    'to keep the brightness higher
    ColorIndex = Math.Min(1, ColorIndex * 2)
    ColorIndex = Math.Pow(ColorIndex, 1 / 5)

    MyBrush = New SolidBrush(Color.FromArgb(255, CInt(255 * ColorIndex * MyRedPercent),
          CInt(255 * ColorIndex * MyGreenPercent), CInt(255 * ColorIndex * MyBluePercent)))
End If
```

You can see that the "system colours" are provided in the *ClsSystemBrushes* class, because *StandardColors* is an instance of this class. These system colours have also been optimized experimentally.

7. Implementation Mechanics

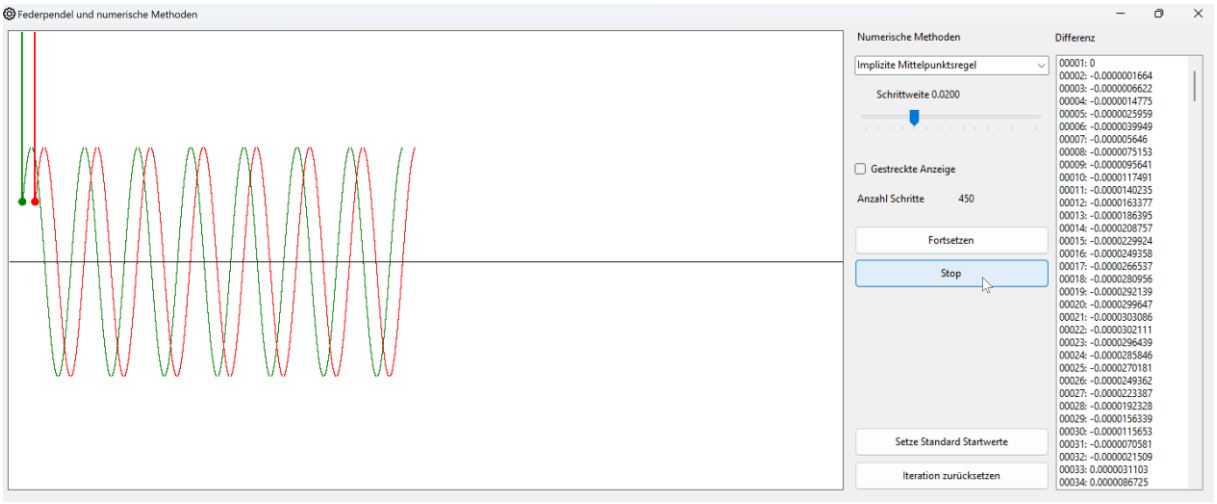
7.1 Implementation of the numerical methods

The architecture of the "Numeric Methods" area is based on the same principles as the previous implementations. The *ClsNumericMethodController* class contains the logic for the user interface and establishes the connection to the lower layers. It accesses the *INumericMethod* interface. This is

implemented by *ClsNumericMethodAbstract*. Concrete implementations then inherit from this class and only overwrite the *Iteration* method.

This method is then essentially a numerical procedure for approximating the spring pendulum for each realization.

Since the movement of a real spring pendulum is compared with the approximation, the challenge of synchronization arises here:



The real spring pendulum should oscillate at the same speed as the approximation, which is calculated using a numerical method. This is ensured in the *ClsNumericMethodController* by the *SetStepWidthAB* method. It calculates the step width of the numerical method and synchronizes it with the step width of the "time" in the real spring pendulum. The diagram can also be stretched.

Here are excerpts from the code:

```
3 Verweise
Public Sub SetStepWidthAB()

    'concernes the number of approximation steps for the PendulumB.Y-value
    'the Default is 1, but the final value is calculated here
    'and is depending of StepWidthB
    Dim NumberOfApproxStepsB As Integer

    'In case of the non-stretched mode
    'the stepwidth for the approximation of PendulumB.Y
    'should be a whole-number divisor of StepWidthA
    'see preliminarey note of the Sub IterationLoop

    'the effect is, that if PendulumA increases by StepWidthA
    'the iteration of PendulumB is repeated so many times
    'that NumberOfApproxStepsB x StepWidthB = StepWidthA
    'thereby both pendulums are synchronized

    Dim LocStepWidth As Decimal
    Dim StretchFactor As Integer = MyForm.TrbStepWidth.Value

    If StretchFactor > 5 Then
        LocStepWidth = CDec(Math.Max(0.01 - 0.001 * (StretchFactor - 6), 0.001))
    Else
        LocStepWidth = CDec(0.1 - 0.02 * (StretchFactor - 1))
    End If
```

```
    If MyForm.ChkStretched.Checked Then

        'in that case, the stepwidth of both pendulums are the same
        'and therefore also the NumberOfApproxStepsB is = 1
        'like NumberOfApproxStepsA
        StepWidthA = LocStepWidth
        StepWidthB = StepWidthA
        NumberOfApproxStepsB = 1

    Else
        'StepWidthA is set as Standard = 0.1
        StepWidthA = CDec(0.1)

        'and StepWidthB is equal locStepWidth and adapted
        'so that StepWidthB x NumberOfApproxStepsB = StepWidthA
        NumberOfApproxStepsB = CInt(Math.Round(StepWidthA / LocStepWidth))
        StepWidthB = StepWidthA / NumberOfApproxStepsB
    End If

    DSA.h = StepWidthA
    DSA.NumberOfApproxSteps = 1

    DSB.h = StepWidthB
    DSB.NumberOfApproxSteps = NumberOfApproxStepsB

    'Set Stepwidth
    MyForm.LblStepWidth.Text = LM.GetString("StepWidth") & " " & StepWidthB.ToString("0.0000")

End Sub
```

When displayed in the diagram, the bitmap *BmpDiagram* is copied, moved to the right and pasted back into *BmpDiagram* (see also section "Graphics in the user interface").

7.2 Implementation of the pendulums

The *ClsPendulumController* class contains the logic of the user interface and establishes the connection downwards. It accesses the *IPendulum* interface, which is implemented by the *ClsPendulumAbstract* class. Concrete implementations of the various pendulums then inherit from this class and overwrite certain methods in it.

```
13 Verweise
Public MustOverride Function GetAddParameterValue(TbrValue As Integer) As Decimal _
    Implements IPendulum.GetAddParameterValue
6 Verweise
Public MustOverride Sub SetAndDrawStartparameter1(Mouseposition As Point) _
    Implements IPendulum.SetAndDrawStartparameter1
6 Verweise
Public MustOverride Sub SetAndDrawStartparameter2(Mouseposition As Point) _
    Implements IPendulum.SetAndDrawStartparameter2
6 Verweise
Public MustOverride Sub IterationStep(IsTestMode As Boolean) Implements IPendulum.IterationStep
12 Verweise
Protected MustOverride Sub DrawPendulums()
19 Verweise
Protected MustOverride Sub SetPosition()
14 Verweise
Protected MustOverride Sub SetStartEnergyRange()
4 Verweise
Protected MustOverride Sub SetPhasePortraitParameters()
4 Verweise
Protected MustOverride Sub SetAdditionalParameters()
8 Verweise
Protected MustOverride Function GetEnergy() As Decimal Implements IPendulum.GetEnergy
6 Verweise
Protected MustOverride Sub SetDefaultUserData() Implements IPendulum.SetDefaultUserData
4 Verweise
Protected MustOverride Sub DrawCoordinateSystem()
```

The management of the parameters is special here. It is not known how many are required to describe the movement of a pendulum system. A maximum of 6 such parameters are available. This would be sufficient for a triple-coupled pendulum. As these depend on the specific pendulum system, they are defined in the *New* Method of the respective system. Here is an example from *ClsDoublePendulum*.

```
Public Sub New()

    Y0 = 0

    MyLabelProtocol = LM.GetString("Parameterlist") & ": u1, v1, u2, v2, Etot"

    MyValueParameterDefinition = New List(Of ClsGeneralParameter)

    'Inizialize all parameters
    'Tag is the Number of the Label in the Pendulum Form
    'L1
    ValueParameter(0) = New ClsGeneralParameter(1, "L1", New ClsInterval(CDec(0.1), CDec(0.85)),
                                            ClsGeneralParameter.TypeOfParameterEnum.Constant, CDec(0.7))
    MyValueParameterDefinition.Add(ValueParameter(0))

    'L2
    ValueParameter(1) = New ClsGeneralParameter(2, "L2", New ClsInterval(CDec(0.1), CDec(0.85)),
                                            ClsGeneralParameter.TypeOfParameterEnum.Constant, CDec(0.2))
    MyValueParameterDefinition.Add(ValueParameter(1))

    'Phi1
    ValueParameter(2) = New ClsGeneralParameter(3, "Phi 1", New ClsInterval(-CDec(Math.PI), CDec(Math.PI)),
                                            ClsGeneralParameter.TypeOfParameterEnum.Variable, CDec(Math.PI / 4))
    MyValueParameterDefinition.Add(ValueParameter(2))

    'Phi2
    ValueParameter(3) = New ClsGeneralParameter(4, "Phi 2", New ClsInterval(-CDec(Math.PI), CDec(Math.PI)),
                                            ClsGeneralParameter.TypeOfParameterEnum.Variable, CDec(Math.PI / 6))
    MyValueParameterDefinition.Add(ValueParameter(3))
```

The GUI is then dynamically adapted when the pendulum system is loaded. Here is the corresponding code in the *InitializeMe* method in *ClsPendulumController*:

```
1 Verweis
Private Sub InitializeMe()
    With DS
        .PicDiagram = MyForm.PicDiagram
        .PicPhaseportrait = MyForm.PicPhasePortrait
        .Protocol = MyForm.LstProtocol
        .LblStepWidth = MyForm.LblStepWidth

        MyForm.TrbAdditionalParameter.Minimum = CInt(.AdditionalParameter.Range.A)
        MyForm.TrbAdditionalParameter.Maximum = CInt(.AdditionalParameter.Range.B)
        MyForm.TrbAdditionalParameter.Value = CInt(.AdditionalParameter.Range.A + 0.5 * _
            .AdditionalParameter.Range.IntervalWidth)

        MyForm.LblAdditionalParameter.Text = .AdditionalParameter.Name & ": " &
            .GetAddParameterValue(MyForm.TrbAdditionalParameter.Value).ToString

        Dim i As Integer
        For i = 1 To 6
            MyForm.GrpStartParameter.Controls.Item("LblP" & i.ToString).Visible = (i <= .ValueParameterDefinition.Count)
            MyForm.GrpStartParameter.Controls.Item("TxtP" & i.ToString).Visible = (i <= .ValueParameterDefinition.Count)
        Next

        Dim LocValueParameter As ClsGeneralParameter
        For Each LocValueParameter In .ValueParameterDefinition
            MyForm.GrpStartParameter.Controls.Item("LblP" & LocValueParameter.ID).Text = LocValueParameter.Name
        Next
```

The following comments on iteration:

- The relevant iteration parameters are based on the formulas of the Runge Kutta method, for example for the double pendulum *u1, v1, u2, v2*. Their start value is also set by *SetAndDrawStartParameter1,2.*
- At the start of an iteration step, *OldPosition = Position* is set. *OldPosition* is therefore the old position of the pendulum before the iteration step.

- For the iteration, a *ClsVector x(3)* is used for the double pendulum based on the mathematical formulas. It plays the role of the individual $\vec{x}_{in}$.
- k11, k12, k13, k14 is managed as *ClsVector(3).* Likewise *k2i, h1i* and *h2i*.
- At the end of each iteration step, the current position of the pendulum is drawn based on *Position* and the pendulum track is drawn in *DrawTrack* based on *OldPosition* and *Position*.

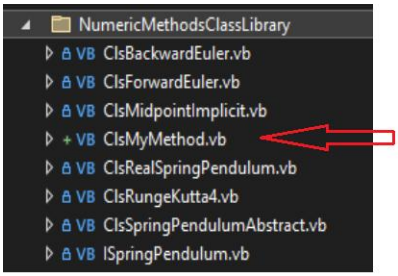
As the individual pendulum systems behave chaotically, they react very sensitively to changes in the starting value and therefore also to the inaccuracies of the numerical approximation method. The behaviour shown in the diagram is therefore not realistic after just a few steps. What can be checked, however, is whether the total energy is retained. The *GetEnergy* method at the level of a specific pendulum system calculates the current energy and compares it with the energy at the start. The energy is then displayed in a bar below the protocol and appears green if the deviation is below 10% of the start energy. Otherwise, it appears red if the energy is too high and purple if the energy is too low.

8. Implementation of own systems in the "Simulator"

If the user of the "Simulator" wants to program their own variants of the existing implemented systems, this is very easy to do. We would like to explain this using the example of our own variant of a numerical method.

1. Step

Create your own *ClsMyMethod* class and add it to the *NumericMethodsClassLibrary* folder:



It is important to adhere to the convention that class names begin with the prefix *Cls*.

2. Step

This class must now inherit the *ClsSpringPendulumAbstract* class. The only part of the code that changes compared to the existing classes is the algorithm of the numerical method:

```
0 Verweise
Public Class ClsMyMethod
    Inherits ClsSpringPendulumAbstract

    Private u As Decimal
    Private v As Decimal


    5 Verweise
    Protected Overrides Sub Iteration()

        Dim i As Integer


        With MyActualParameter
            For i = 1 To MyNumberOfApproxSteps
                .Component(0) += MyH

                'Component(1) holds the y-value
                u = .Component(1)
                v = .Component(2)

                'this is the numerical approximation
                .Component(1) = My own formula For the first component
                .Component(2) = My own formula For the second component
            Next

            'the Conponent(0) holds the "time" t with 2*pi period
            .Component(0) = .Component(0) Mod CDec(2 * Math.PI)

        End With

    End Sub
End Class
```

The code could then look as shown above.

No further steps are necessary. In particular, the combo box in the *FrmNumericMethods* is filled with the selection of numeric methods by *reflection.* No change is therefore necessary. The own method is listed as *MyMethod* in the combo box. This means that the prefix *Cls* is cut out of the name.

3. Step (optional)

If you want to choose different names in German and English than *MyMethod*, you can enter a corresponding key in the resource files *LabelsEN* and *LabelsDE*. This is done as follows:

In LabelsEN:

ClsMyMethod	My own numeric method	Cls

In LabelsDE:

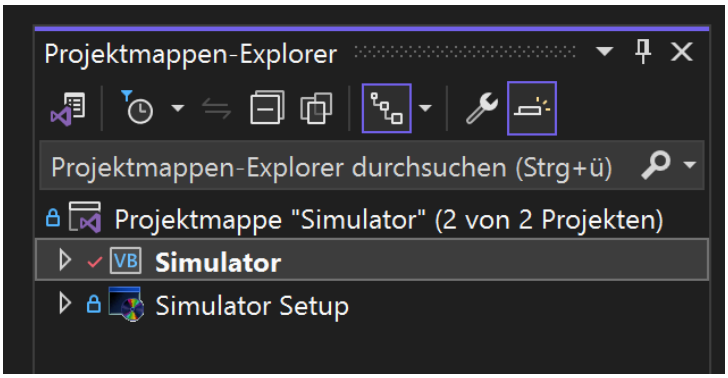
ClsMyMethod	Meine eigene numerische Methode	Cls

In both cases, the key (name of the entry) is the class name, the value is your own selected designation and the comment 'Cls'.

The same applies to other custom extensions for growth models, complex iteration or billiards. Your own implementations must inherit the corresponding abstract class and overwrite certain methods in it.

4. Step (optional)

You can publish the “Simulator” as MSI-file. To generate that, there is a “Simulator Setup” project:


Projektmappen-Explorer
Projektmappen-Explorer durchsuchen (Strg+ü)
Projektmappe "Simulator" (2 von 2 Projekten)
Simulator
Simulator Setup