

# Der Simulator

## Die technische Dokumentation zum Computerprogramm «Simulator»

*Das Computerprogramm «Simulator», dessen Einsatz in einem separaten Handbuch beschrieben wird, ermöglicht die Simulation einfacher dynamischer Systeme und das Experimentieren mit ihnen. Der Code ist öffentlich auf GitHub zugänglich, in VB.NET geschrieben, mit ausführlichen Kommentaren versehen und kann nach Bedarf erweitert werden. Dazu ist die kostenlose Community-Version von Microsoft Visual Studio nötig, und zwar mindestens in der Version 17.9. Diese setzt auf dem Microsoft Framework 8.0 auf.*

*Dieses Dokument beschreibt die technische Struktur und die Architektur für den Simulator. Die mathematischen Grundlagen findet man im Dokument «Dynamische Systeme».*

*Version 6.0 - 01.10.2024*

### Inhalt

Einführung .....	3
1. Basiskonzepte .....	4
1.1 Bezeichnungen und Programmierstandards .....	4
1.2 Versionierung .....	4
1.3 Zustandsraum eines dynamischen Systems .....	5
1.4 GeneralClassLibrary .....	6
1.5 ClsGraphicTool .....	6
1.6 ClsDiagramAreaSelector .....	7
1.7 ClsLanguageManager und Lokalisierung .....	8
1.8 Parameter Kategorien .....	9
1.9 User Interface: Darstellung und Funktion .....	10
2. Architektur .....	10
2.1 Konzept .....	10
2.2 User Interface: Logik .....	11
2.3 Graphik im User Interface .....	12
2.4 General Forms .....	14
2.5 Form Controller .....	15
2.6 Interface DS und DS Abstract .....	15
2.7 Der typische Ladevorgang .....	16
2.8 Auswahl eines dynamischen Systems durch den Benutzer .....	17
3. Steuerung der Iteration .....	20
3.1 Start, Unterbruch, Stopp .....	20

3.2	Verschiedene Steuerungsebenen der Iteration .....	23
3.3	Asynchrone Iterationen und Performance.....	25
3.4	Definition der Startparameter durch den Benutzer.....	27
4.	Implementierung des Billards.....	30
4.1	Implementierung des elliptischen Billards.....	31
4.2	Implementierung des Ovalen Billards .....	33
4.3	Implementierung des Stadium Billards .....	33
4.4	Implementierung des C-Diagrammes .....	33
5.	Implementierung der Wachstumsmodelle.....	34
5.1	N:M Beziehung zwischen Wachstumsmodellen und Darstellungsformen .....	34
5.2	Implementierung des Feigenbaum Diagrammes .....	36
6.	Implementierung der komplexen Iteration .....	36
6.1	Implementierung der Newton Iteration.....	36
6.2	Implementierung der Julia- und Mandelbrotmenge .....	40
7.	Implementierung Mechanik .....	41
7.1	Implementierung der numerischen Methoden .....	41
7.2	Implementierung der verschiedenen Pendel.....	42
8.	Implementierung eigener Systeme im «Simulator».....	44

## Einführung

Der «Simulator» ermöglicht die Simulation von einfachen dynamischen Systemen. Die mathematischen Grundlagen dazu und die Konzepte für die Implementierung sind im Dokument «Dynamische Systeme» beschrieben. Im «Benutzerhandbuch» wird die Bedienung des «Simulator» erklärt.

Der Code des Programmes ist in GitHub veröffentlicht und als Open Source zugänglich. Er ist in VB.NET verfasst und ausführlich (auf Englisch) kommentiert. Die Entwicklungsumgebung ist die Community Version von Microsoft Visual Studio 2022. Sie ist gratis verfügbar und einfach zu installieren. Nötig ist mindestens die Version 17.9. Diese setzt auf dem Microsoft Framework 8.0 auf, welches ebenfalls heruntergeladen und installiert werden muss, falls es nicht schon vorhanden ist.

Die aktuelle Version des «Simulator» ist 6.0.0 publiziert am 1.10.2024.

Der GitHub Link ist folgender:

<https://github.com/HermannBiner/Simulator>

Dieses Dokument enthält die technische Dokumentation zum «Simulator» und Hinweise, falls man eigene dynamische Systeme im «Simulator» entwickeln will.

# 1. Basiskonzepte

## 1.1 Bezeichnungen und Programmierstandards

Grundsätzlich beginnen alle Bezeichnungen und jedes neue Wort darin mit Grossbuchstaben.

Der Typ eines Objektes wird in üblicher Weise mit drei Buchstaben abgekürzt, z.B.

- Cls für eine Klasse, z.B. *ClsGraphicTool*
- Frm für eine Windows-Form, z.B. *FrmPendulum*
- Txt für ein Textfeld, z.b. *TxtParameter*

Usw.

Eine Ausnahme ist ein Interface. Dieses wird lediglich mit einem vorangestellten «I» gekennzeichnet. Z.B: *IBilliard*.

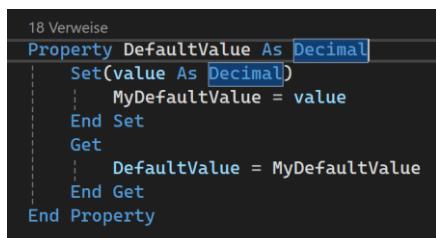
Abstrakte Klassen werden mit einem nachgestellten «Abstract» versehen. Z.B:  
*ClsGrowthModelAbstract*.

Variable vom Typ Boolean haben in der Regel das Präfix «Is». Also z.B. *IsFormLoaded*.

Grundsätzlich sind alle Bezeichnungen auf Englisch. Ebenso sämtliche Kommentare im Code.

Die Übergabe von Parametern zwischen den Klassen geschieht grundsätzlich durch *Properties*. Das ermöglicht, dass bei der Übergabe weiterer Code ausgeführt oder ein Check durchgeführt werden kann. Wenn ein Parameter mit dem Namen *Example* übergeben wird, dann wird er innerhalb der Klasse mit «*MyExample*» bezeichnet. Damit werden Parameter, welche ausserhalb der Klasse zur Verfügung stehen, unterschieden von den privaten einer Klasse.

Beispiel: *ClsGeneralParameter*



The screenshot shows a code editor with a dark theme. A tooltip or callout box highlights a property definition. The code is as follows:

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

Zusätzlich ist jeder Parameter und jede Methode einer Klasse mit *Private*, *Protected*, *Public* versehen.

Implizite Typenkonvertierung ist nicht erlaubt. Wegen der Rechengenauigkeit werden meist die Zahlentypen *Decimal* und zusätzlich auch *Integer* verwendet. Manchmal (bei grossen Zahlwerten) auch *Double*.

## 1.2 Versionierung

Die Nummern *Hauptversion.Nebenversion.Patch* werden wie folgt nachgeführt:

*Hauptversion*

Diese wird erhöht, wenn eine neue Kategorie dynamischer Systeme implementiert wird. Die Kategorien in Hauptversion 6 sind z.B. Billard, Wachstumsmodelle, Mechanik, Komplexe Iteration. Eine neue Kategorie führt zu einem neuen Menüpunkt im Hauptmenü. Der aktuelle Stand «6» bei der Hauptversion ist historisch bedingt, auf Grund von grösseren architektonischen Vereinheitlichungen oder tiefgreifenden Performance Optimierungen.

### Nebenversion

Diese wird erhöht, wenn innerhalb einer bestehenden Kategorie ein weiteres dynamisches System implementiert wurde oder wenn für ein bestehendes dynamisches System eine neue Darstellung zur Verfügung steht.

### Patch

Wird bei Bug-Fixing und allgemeinen Optimierungen erhöht.

Wenn die Hauptversionsnummer erhöht wird, starten die übrigen Nummern wieder bei 0.

## 1.3 Zustandsraum eines dynamischen Systems

Ein *dynamisches System* ist ein Tripel  $(T, X, f)$ . Die Menge  $T$  ist der *Zeitraum*. Im «Simulator» werden nur diskrete dynamische Systeme betrachtet. Deshalb ist immer  $T = \mathbb{N}$ .

Die Menge  $X$  ist der *Zustandsraum*. Bei den im «Simulator» implementierten Systemen ist der Zustandsraum eine Teilmenge von  $\mathbb{R}, \mathbb{R}^n$  oder  $\mathbb{C}, \bar{\mathbb{C}}$ . Der aktuelle Systemzustand wird durch entsprechende Parameter definiert.

Beispiele:

- Billard:  $(t, \alpha) \in [a, b] \times ]0, \pi[ \subset \mathbb{R}^2$
- Wachstumsmodelle:  $x \in [a, b] \subset \mathbb{R}$
- Iteration im Komplexen:  $z \in \mathbb{C}, \bar{\mathbb{C}}$
- Mechanik:  $n$ -Tupel  $\subset \mathbb{R}^n$

Das dynamische System «kennt» seine Parameter. Damit ein Parameter definiert ist, braucht es:

- Eine ID
- Einen Namen
- Der Definitionsbereich, in welchem der Parameter zugelassen ist
- Eine zusätzliche Typisierung
- Einen Default-Startwert, welcher gesetzt wird, bevor der Benutzer diesen ändert

Das ist in der Klasse

*ClsGeneralParameter*

bereit gestellt.

Die Definition befindet sich ebenfalls in *ClsGeneralParameter*.

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

*Variable* ist ein «normaler» Werteparameter des dynamischen Systems. Er definiert den aktuellen Zustand eines Systems und ändert sich im Laufe der Iteration (z.B. die Auslenkungswinkel des Doppelpendels).

*DS* ist der Parametertyp, welcher die besondere Rolle für den Übergang zu chaotischem Verhalten und damit zur wesentlichen Eigenschaft des dynamischen Systems spielt, z.B. der Parameter *a* beim logistischen Wachstum. Dieser Parameter spielt eine wichtige Rolle in der Formel des

Bewegungsgesetzes des dynamischen Systems und damit bei entsprechenden Diagrammen (z.B. FeigenbaumDiagramm oder CDiagramm), wo sie in Richtung der x-Achse abgebildet werden.

*Constant* ist ein Parameter, der beim Start Iteration definiert werden kann, sich aber anschliessend während der Iteration nicht mehr ändert (z.B. die Längen des Doppelpendels).

In einer Klasse, welche in dynamisches System darstellt, müssen diese Parameterdefinitionen gemacht werden. Das führt z.B. zu folgendem Code (hier in der Klasse *ClsDoublePendulum* in der Methode *New*):

```
'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

Diese Library enthält viele einfache Hilfsklassen, wie z.B.

- *ClsInterval*
- *ClsGeneralParameter*
- *ClsMathPoint* (ein Punkt im  $\mathbb{R}^2$  in mathematischen Koordinaten, im Gegensatz zu einem *Point* aus der .NET Bibliothek, welcher immer in Pixelkoordinaten vorliegt)
- *ClsValuePair* (von der Struktur her genau wie ein *ClsMathPoint*, hat aber eine andere Rolle und repräsentiert ein Werteparameter-Paar und keinen Punkt im  $\mathbb{R}^2$ )
- *ClsNTupel* (hier kann die Dimension N beim Instanzieren des Tupels definiert werden).
- *ClsMathHelperDS* (diese enthalten mathematische Hilfsfunktionen, welche auf das jeweilige dynamische System zugeschnitten sind, aber im Prinzip überall zur Verfügung stehen). Was diese Hilfsklassen «können», sieht man am besten im Code selbst, welcher entsprechend kommentiert ist.
- *ClsSystemBrushes* (für farbige Darstellungen, bei denen die Farbe von einer Anzahl von Iterationsschritten abhängt (z.B. bei der Iteration im Komplexen), werden hier die Standard-Systemfarben definiert)
- *ClsCheckXY* (hier stehen verschiedene Checks zur Verfügung, z.B: ob ein Wert numerisch ist, ob ein Wertepaar ein Intervall darstellt). Eine spezielle Rolle spielt
- *ClsCheckUserData* (hier wird ein einzelner Wert übergeben und gecheckt, ob er ein zugelassenes Format hat und ob er im erlaubten Definitionsbereich des Parameters ist. Es kann auch geprüft werden, ob ein Intervall in diesem erlaubten Definitionsbereich liegt.)

Weitere Klassen in dieser Bibliothek werden weiter unten erläutert.

## 1.5 ClsGraphicTool

Die Bewegung eines dynamischen Systems wird im «Simulator» in eine PictureBox *PicDiagram* oder einer BitMap *BmpDiagram* abgebildet. Wir werden deren unterschiedliche Rolle später sehen. Hier geht es um die Umrechnung zwischen Pixel- und mathematischen Koordinaten.

Damit das nicht für jedes dynamische System immer wieder einzeln gemacht werden muss, erledigt das die Klasse *ClsGraphicTool*. Sie spielt bei allen graphischen Darstellungen die Hauptrolle. Iher wird übergeben:

- Die PictureBox oder BitMap in welche gezeichnet werden soll
- Die Intervalle für die mathematischen Koordinaten

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

Da die Grösse bzw. die Anzahl der Pixel der PictureBox oder BitMap damit bekannt sind, kann damit die Klasse *ClsGraphicTool* die entsprechenden Umrechnungen selbst vornehmen.

Sie stellt dann alle benötigten Zeichnungsoptionen zur Verfügung, wie zeichnen eines Punktes, einer Linie, einer Ellipse, eines Rechtecks usw.

## 1.6 ClsDiagramAreaSelector

Nachdem ein Diagramm gezeichnet wurde, soll der Benutzer in bestimmten Fällen (z.B. Feigenbaum-Diagramm) die Möglichkeit haben, im Diagramm ein Rechteck zur weiteren Untersuchung auszuwählen. Damit werden die Größen der entsprechenden mathematischen Intervalle neu und entsprechend der Auswahl des Benutzers neu gesetzt.

Das heisst, dass in diesem Kontext immer unterschieden werden muss, welches die in *ClsGeneralParameter* fix definierten und erlaubten Definitionsbereiche sind (diese werden der Klasse *ClsDiagramAreaSelector* als *XRange* und *YRange* übergeben) und welches die vom Benutzer ausgewählten Teilintervalle sind (das sind *UserXRange* und *UserYRange*).

Der typische Ablauf ist dann folgender:

- Die Klasse *ClsDiagramAreaSelector* wird instanziert und es wird als Standard gesetzt:  
*UserXRange* = *XRange*, *UserYRange* = *YRange*
- Gleichzeitig wird der Klasse das *PicDiagram* übergeben, in welchem die Benutzauswahl stattfinden soll
- Da die Grösse dieses Diagrammes sowie die Ranges bekannt sind, kann die Klasse Umrechnungen von Pixel- und mathematischen Koordinaten selbst erledigen.
- Wenn der Benutzer nun einen Teilbereich des Diagrammes auswählt, dann macht er das mit gedrückter Maustaste. Dabei wird der Klasse die jeweilige Mausposition übergeben. Die Klasse zeichnet dann im Diagramm ein entsprechendes Rechteck. Sie rechnet die zugehörigen Pixelkoordinaten in mathematische Koordinaten um und setzt im Benutzerfenster die entsprechenden Intervalle *UserXRange*, *UserYRange* in den zugehörigen Textboxen *TxtXMin*, *TxtXMax*, *TxtYMin*, *TxtYMax*. Die Referenz auf diese Felder wird der Klasse bei der Instanzierung übergeben. Siehe den entsprechenden Code in *ClsDiagramAreaSelector*.
- Nun wird das dynamische System mit den variablen Parametern nicht mehr im Bereich *XRange* x *YRange* rechnen, sondern in *UserXRange* x *UserYRange*. Wenn der Benutzer anschliessend nochmals eine weitere Auswahl trifft, ist diese dann relativ zu diesem neuen Bereich.

Beispielcode dazu findet man u.a. in *ClsFeigenbaumController*. Bei der Instanzierung:

```
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
```

Anschliessend muss das dynamische System ebenfalls unterscheiden zwischen den in *ClsGeneralParameter* definierten Ranges (diese werden als *DS.FormulaParameter.Range* und *DS.ValueParameter.Range* bezeichnet) und den vom User definierten Ranges entsprechend seiner Auswahl. Beim Formula-Parameter wird diese mit *ActualParameterRange* und beim Value-Parameter mit *ActualValueRange* bezeichnet.

Beim Start einer Iteration im Auswahlfenster werden die vorhin übergebenen Werte in den Textfeldern, welche *ClsDiagramAreaSelector* gesetzt hat, diesen Ranges übergeben. Wichtig ist, dass auch die mathematischen Intervalle, welche von *ClsGraphicTool* gehalten werden, entsprechend angepasst werden. Das führt dann z.B. zu folgendem Codeausschnitt:

```
1 Verweis
Public Sub StartIteration()

    ResetIteration()
    If IterationStatus = ClsDynamics.EIterationStatus.Stopped Then
        If IsUserDataOK() Then
            With MyForm
                DiagramAreaSelector.IsActive = False
                IterationStatus = ClsDynamics.EIterationStatus.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
```

Der DiagramAreaSelector darf natürlich während einer laufenden Iteration nicht aktiv sein.

## 1.7 ClsLanguageManager und Lokalisierung

Damit sämtliche Anzeigen in der gewählten Sprache (momentan D/E) erscheinen, wird die Klasse *ClslanguageManager* eingesetzt. Sie wird als Singleton implementiert und steht dann global zur Verfügung.

Die Klasse hat die Methode *GetString(«StrID»)*. Der String, welcher übergeben wird, muss eine eindeutige ID sein, damit der entsprechende Text in der definierten Sprache zurückgegeben werden. Die Einträge dazu sind in Resourcen Dateien

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

aufgelistet. Wenn kein Eintrag gefunden wird, wird *StrID* zurückgegeben.

Ein Spezialfall liegt vor, wenn der Benutzer ein eigenes dynamisches System implementiert. Das wird später detaillierter erläutert. Betreffend die sprachliche Bezeichnung gilt folgendes.

Angenommen der Benutzer erstellt eine neue Klasse *ClslMyDynamicSystem*. Diese muss ein Interface für die entsprechende Klasse von dynamischen Systemen implementieren (z.B. *IBilliard*). Beim Laden des Benutzerfensters werden *By Reflection* alle Klassen geladen, welche dieses Interface implementieren. Dabei «taucht» nun auch die Klasse *ClslMyDynamicSystem* auf. Der Language Manager sucht dann in den Resourcen Dateien nach einem Eintrag mit diesem Klassennamen *ClslMyDynamicSystem*. Wenn dort ein Eintrag gefunden wird, dann wird in der Auswahl-Combo des Benutzerfensters das von ihm programmierte System unter diesem Namen angezeigt. Wenn kein Eintrag vorliegt, erscheint das System unter der Bezeichnung *ClslMyDynamicSystem*.

Wenn eines Tages weitere Sprachen zur Verfügung stehen sollen, muss man lediglich die im Language Manager gehaltene Enumeration

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

Durch die weitere Sprache erweitert werden und dann muss eine weitere Resourcen-Datei für diese Sprache erstellt werden. Dabei werden einfach die bereits vorliegenden Einträge mit denselben ID's in die neue Sprache übersetzt.

## 1.8 Parameter Kategorien

Es ist nützlich, zwischen folgenden Kategorien von Parametern zu unterscheiden.

### *Spezifizierende Parameter (DS Parameter)*

Das sind Parameter, welche die Art des dynamischen Systems spezifizieren. Zum Beispiel muss bei der Newton Iteration noch der Exponent der zu untersuchenden Einheitswurzeln festgelegt werden. Oder beim logistischen Wachstum der *FormulaParameter* a.

Eine Änderung dieser Parameter kommt der Wahl eines neuen dynamischen Systems gleich.

### *Startparameter*

Diese legen den Zustand des dynamischen Systems beim Start fest. Das sind *ValueParameter*. Beim ersten Start werden diese auf die Default Werte gesetzt, welche in *ClslGeneralParameter* definiert sind. Der Benutzer kann diese Startwerte verändern, ohne dass dies einem Wechsel des dynamischen

Systems gleich kommt. Wenn wieder die Default Werte gesetzt werden sollen, geschieht dies mit der Methode *SetDefaultUserData*, welche überall so genannt wird.

### Darstellungsparameter

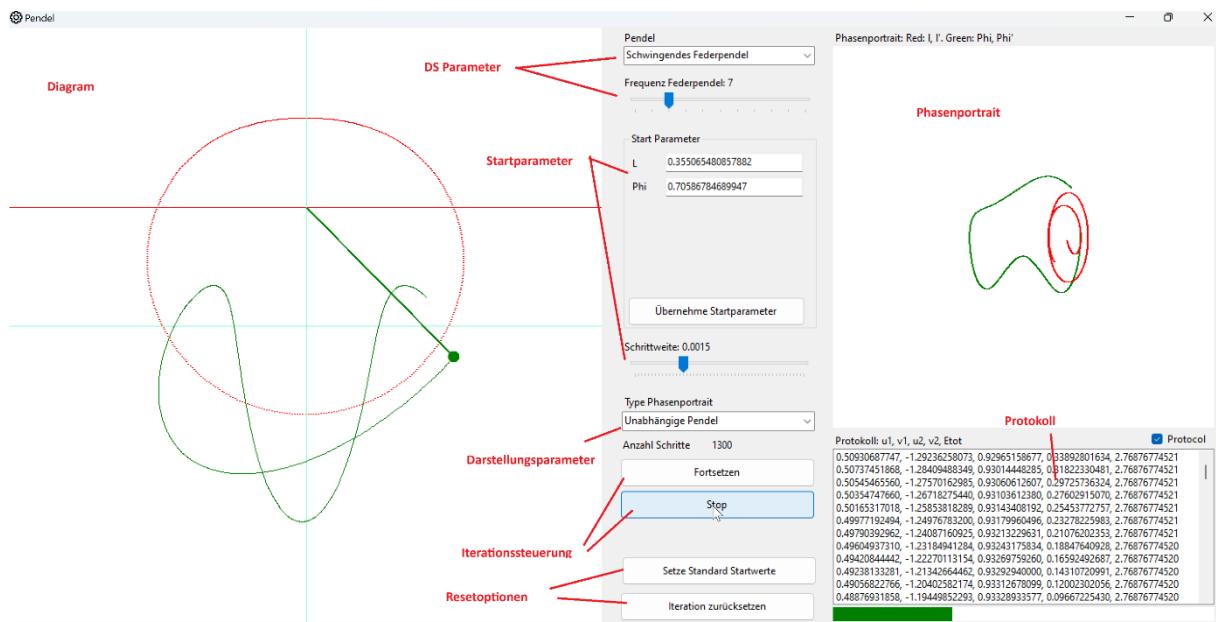
Das sind Parameter, welche sich auf die Art der Darstellung beziehen. Z.B: soll eine Darstellung farbig erscheinen, oder ein Protokoll soll ein-/ausgeblendet werden. Oder es soll die Geschwindigkeit von Billardbällen eingestellt werden.

Diese Parameter sind ebenfalls beim Start auf einen Default gesetzt. Sie bleiben aber stehen, wenn die Methode *SetDefaultUserData* aufgerufen wird.

Siehe dazu auch den Abschnitt «Form Controller» weiter unten.

## 1.9 User Interface: Darstellung und Funktion

Ein typisches Benutzerfenster hat dann diesen Aufbau:



Typischer Aufbau eines Benutzerfensters

Wichtig ist die Rolle der Startparameter. Diese können auf verschiedene Arten gesetzt werden:

- Setze Standard Startwerte
- Manuelle Einträge
- Setzen der Startposition mit der Maus

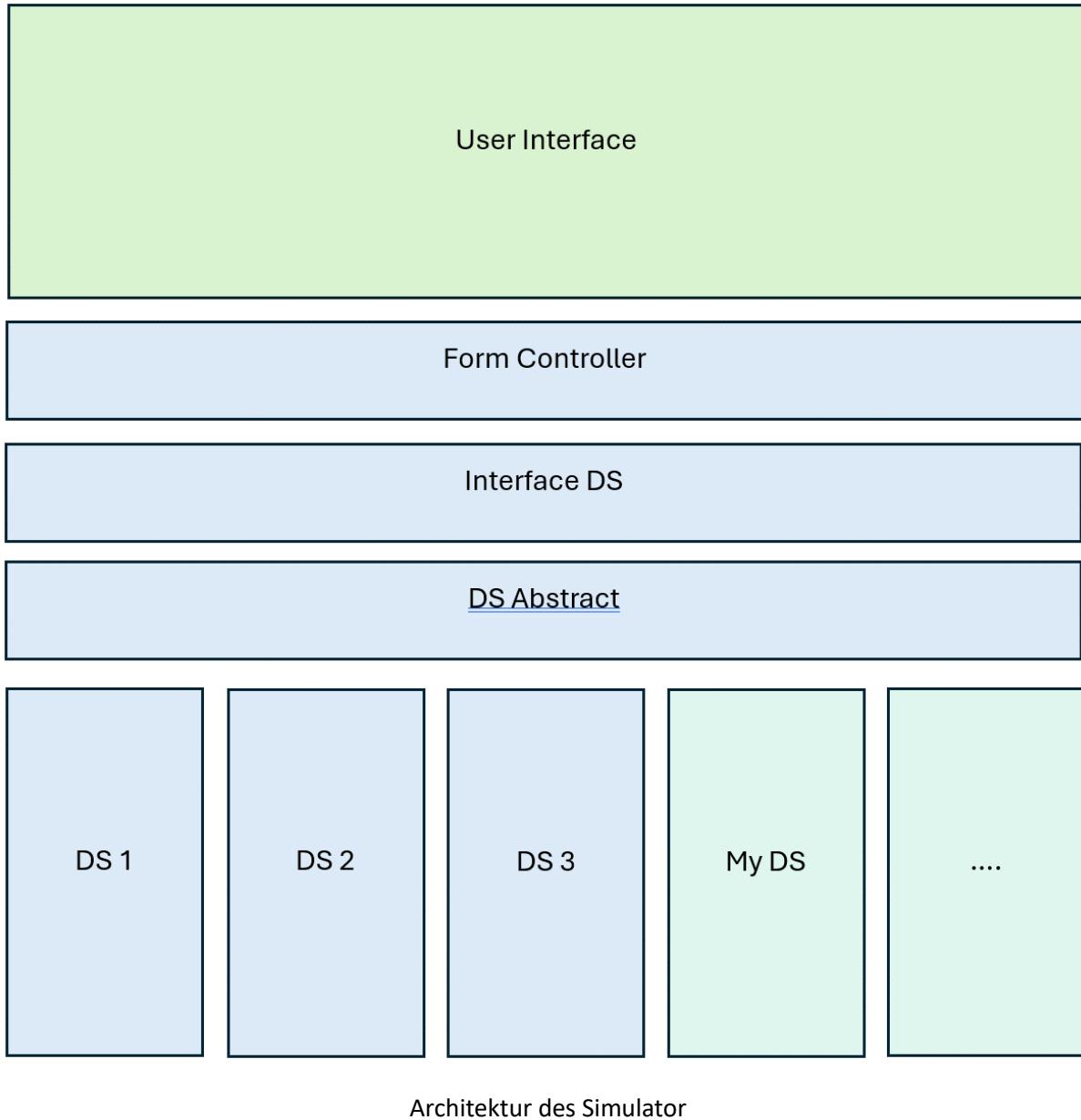
Beim Starten der Iteration werden diese Werte auf das dynamische System übertragen.

Die Taste «Übernehme Startparameter» hat die Funktion, dass bei einer manuellen Eingabe derselben die Startposition des dynamischen Systems im Diagramm entsprechend angepasst wird.

## 2. Architektur

### 2.1 Konzept

Die Zuständigkeiten im «Simulator» sind entsprechend untenstehendem Schema verteilt:



## 2.2 User Interface: Logik

Das User Interface ist eine WindowsForm und hält nur die Logik, welche unmittelbar mit der Steuerung der Controls in der Form zu tun hat. Dazu gehört auch deren Beschriftung, welche in jeder Form in der Methode

*InitializeLanguage*

Implementiert ist.

Damit beim Laden nicht unerwünschte «Events» erzeugt werden, wie z.B. *TextChanged*, führt jede Form eine Boolean-Variable *IsFormLoaded*. Diese wird am Anfang des Ladevorganges auf *false* gesetzt und nach Beendigung des Ladevorgangs auf *true*. Bei allen Events der Form wird geprüft, ob der Ladevorgang abgeschlossen ist.

Zusätzlich muss die Form beim Laden «ihren» Form Controller instanzieren, welcher im nächsten Abschnitt beschrieben wird.

Das führt dann zu folgendem typischen Code in der Form:

```
0 Verweise
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
```

Und

```
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("ExaminedValueRange")
    LblParameterRange.Text = LM.GetString("ExaminedParameterRange")
    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
```

Der Formcontroller wird in jeder Form mit *FC* bezeichnet.

## 2.3 Graphik im User Interface

Bei der Darstellung der Bewegung eines dynamischen Systems soll einerseits der aktuelle Systemzustand dargestellt werden (dieser ändert laufend) und andererseits sie «Spur» der Bewegung

aufgezeichnet werden. Die Lösung dafür besteht darin, dass der aktuelle Systemzustand in der PictureBox *PicDiagram* gezeichnet wird und die permanente Spur in einer Bitmap *BmpDiagram*. Beide müssen dieselbe Grösse haben. Die *Image* Eigenschaft von *PicDiagram* verweist dann auf *BmpDiagram*, sodass immer beide gleichzeitig dargestellt werden.

Für das Zeichnen in *PicDiagram* ist dann eine Instanz des ClsGraphicTool *PicGraphics* zuständig und für das Zeichnen in *BmpDiagram* die Instanz *BmpGraphics*.

Das führt dann zu folgendem Code Beispiel bei *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

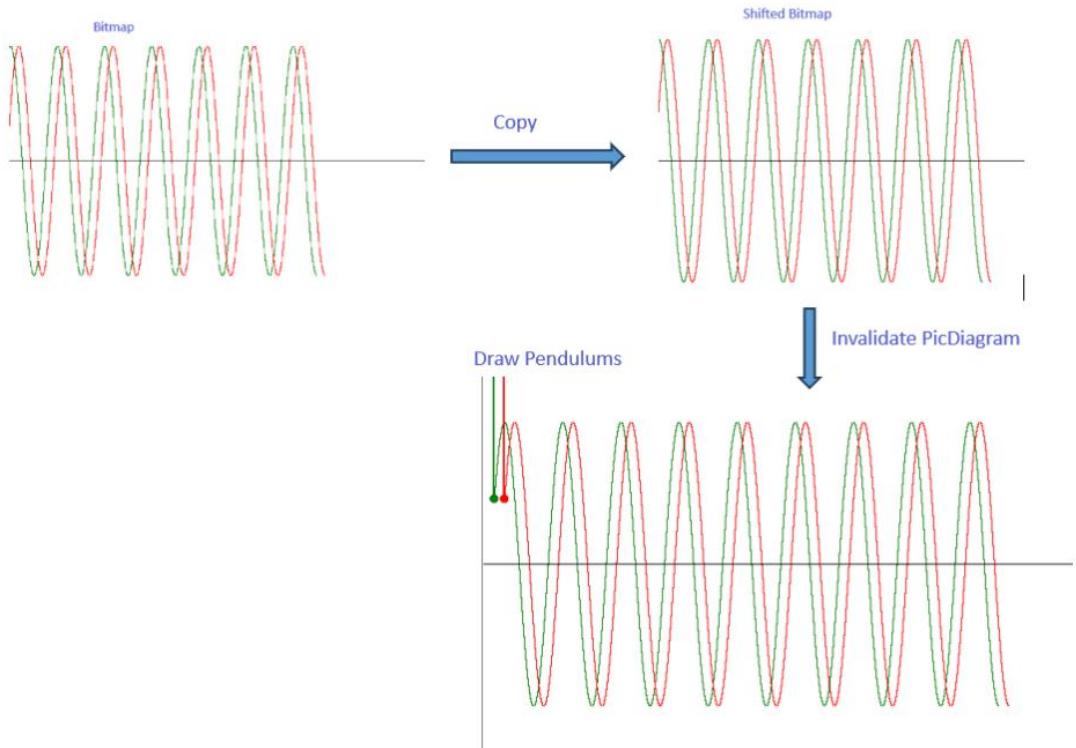
        'Graphics
        'The MathInterval is the same for X and Y
        PicGraphics = New ClsGraphicTool(MyPicDiagram, MathInterval, MathInterval)
        BmpGraphics = New ClsGraphicTool(BmpDiagram, MathInterval, MathInterval)

    End Set
End Property
```

Wenn der mathematische Definitionsbereich ändert (z.B. durch den *ClsDiagramAreaSelector*) müssen ebenfalls die entsprechenden Parameter von *PicGraphics*, *BmpGraphics* geändert werden.

Bei der Untersuchung der numerischen Methoden im Ordner *NumericMethodClassLibrary* muss die Bewegung eines Pendels auf der Zeitachse, welche sich nach rechts schiebt, dargestellt werden. Das geschieht durch Kopieren der *BmpDiagram* mit Verschiebung um einen Schritt nach rechts und die Kopie wird wieder in die *BmpDiagram* eingefügt.

Die Idee ist folgende:



Die aktuelle Bitmap wird kopiert und dann geleert. Dann wird die Kopie der Bitmap wieder in die ursprüngliche Bitmap eingefügt, aber mit einem 1-Pixel Shift nach rechts. Durch *PicDiagram.Refresh* die die geshiftete Bitmap in das PicDiagram übernommen. Anschliessend wird noch die aktuelle Position der Pendel in das PicDiagram eingezeichnet.

Da führt zu folgendem 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

Unter *GeneralForms* sind die allgemeinen Windows Formen abgelegt:

- *FrmMain* ist die Form, welche beim Programmstart geladen wird. Sie enthält alle Menüpunkte.
- *FrmTest* ist eine Form, welche standardmäßig nicht sichtbar ist. Sie dient bei der Entwicklung neuer dynamisches Systeme oder neuen Darstellungen von diesen zu Testzwecken.

- *FrmInfo* zeigt dem Benutzer die Informationen zur aktuellen Version des «Simulator» an.

## 2.5 Form Controller

Ein Form Controller (z.B. *FrmBilliardController*) enthält die Steuerungslogik für die Form. Er hat eine Referenz auf die Form (siehe *Load* Ereignis in den Codebeispielen) und damit auch auf alle Elemente der Form.

Der Form Controller sorgt für den typischen Ablauf beim Laden einer Form und reagiert auf Benutzeraktionen. Der Ladevorgang ist im folgenden Abschnitt erläutert.

Der Form Controller steuert auch den Ablauf der Iteration. Dies wird in einem folgenden Abschnitt erläutert.

## 2.6 Interface DS und DS Abstract

Das Interface DS stellt die Verbindung zwischen Form Controller und dem dynamischen System dar. Es liefert alle Uservorgaben «nach unten» und alle relevanten Daten für den Form Controller «nach oben». Ferner stellt es die Methoden für die Iteration und meist auch für das Setzen der Startposition zur Verfügung.

Beispiel: *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

```

Die spezifischen dynamischen Systeme einer Kategorie implementieren «ihr» Interface. Bei allen ist aber gemeinsamer Code vorhanden, mindestens das Setzen der Parameter durch die Properties Schnittstelle. Zudem gibt es gemeinsame Objekte wie z.B. der Language Manager.

Beispiel: Code Ausschnitt aus *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 Der typische Ladevorgang

Die verschiedenen dynamischen Systeme, welche in einer Form angeboten werden, sind in einer ComboBox aufgelistet. Diese muss beim Laden der Form gefüllt werden. Die Form ruft eine entsprechende Routine im Form Controller auf, z.B.:

```

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

```

Und im Formcontroller hat man dann:

```

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 called with 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

```

Wie man sieht, wird hier die Technik der «Reflection» verwendet.

Der Form Controller hat dann eine Referenz auf die Form: *MyForm*. Er kennt dann auch das ausgewählte dynamische System, welches in jedem Form Controller als *DS* bezeichnet wird. Je nach Zusammenhang «braucht» der Form Controller auch einen *DiagrammAreaSelector*. Beispiel:

```

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 Auswahl eines dynamischen Systems durch den Benutzer

Beim Laden der Form wird ein dynamisches System als Standard gesetzt. Das führt zum selben Prozess, wie wenn der Benutzer ein dynamisches System selbst auswählt.

Die entsprechend aufgerufenen Methoden des Controllers sind:

- *SetDS*: Hier wird das dynamische System ausgewählt und instanziert. Beispiel:

```

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

```

Jedes Mal, wenn das dynamische System neu gewählt wird, muss es anschliessend zusammen mit weiteren Objekten der Form (z.B. *ClsDiagramAreaSelector*) initialisiert werden. Das geschieht mit der Methode *InitializeMe*. Beispiel *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
        .ProtocolList = 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

```

Da das dynamische System neu geladen wurde, müssen die Standard Benutzerdaten als Startdaten für das dynamische System gesetzt werden. Das geschieht in der Methode *SetDefaultUserData*. Beispiel 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
```

Da möglicherweise vorher eine Iteration durchgeführt wurde, muss diese zurückgesetzt werden. Das geschieht in der Methode *ResetIteration*. Diese wird oft an das dynamische System weitergereicht, je nachdem, auf welcher Ebene die Kontrolle der Iteration erfolgt. Beispiel in *ClsJuliaIterationController*:

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

Und dann im *DS* (hier 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.EIterationStatus.Stopped

End Sub
```

Diese Methoden

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

Werden in allen Form Controller gleich benannt und nach demselben Prozess abgearbeitet.

### 3. Steuerung der Iteration

#### 3.1 Start, Unterbruch, Stopp

Jeder Form Controller verwaltet den Zustand der Iteration. Die möglichen Zustände werden in der Klasse *ClsDynamics* als Enumeration aufgeführt.

```
99+ Verweise
✓Public Class ClsDynamics

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

End Class
```

Bevor eine Iteration zum ersten Mal gestartet wurde, ist sie im Zustand

*Stopped*

Wenn sie in diesem Zustand gestartet wird, dann muss zuerst gecheckt werden, ob die Benutzereingaben innerhalb der erlaubten Bereiche sind, Das geschieht in der Funktion

*IsUserDataOK*

Welche true oder false zurückgibt.

Wenn die Benutzereingaben OK sind, dann wird der Status der Iteration auf

*Ready*

gesetzt.

In diesem Zustand startet die Iteration und erhält den Status

*Running*

Anschliessend durchläuft sie entweder eine vorgegebene Anzahl Schritte und wird dann beendet, Dann erhält sie wieder den Zustand *Stopped*. Oder sie läuft, bis sie der Benutzer durch die Stopp-Taste unterbricht. Dann erhält sie den Zustand

*Interrupted*

Das führt zu folgendem Code (z.B: 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

```

Und

```

1 Verweis
Public Sub StopIteration()
    'the iteration was running and is interrupted
    IterationStatus = ClsDynamics.EnIterationStatus.Interrupted
    'the iteration is stopped 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

```

Wird die Iteration zurückgesetzt durch *ResetIteration*, werden alle Grafiken und Status- bzw. Protokollfelder geleert. Die Benutzerdaten bleiben aber stehen. Das betrifft sowohl den Controller wie u.U. auch das dynamische System. Deshalb haben beide eine Reset Iteration-Methode. Ein typischer Codeausschnitt ist folgender (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

```

Und im 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.EIterationStatus.Stopped

    MyIsStartParameter1Set = False
    MyIsStartParameter2Set = False

End Sub

```

Wenn der Benutzer eine neue Iteration mit demselben dynamischen System und denselben Benutzerdaten starten will, dann kann er dies mit *ResetIteration* bereitstellen.

Manchmal möchte man aber auch die Benutzerdaten auf den Default stellen. Das geschieht mit der Methode *ResetIteration* gefolgt von *SetDefaultUserData*. Diese kann sowohl den Controller wie auch das dynamische System betreffen. Beispielcode im *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

```

Und im DS *ClsDoublePendulum* (Overrides *SetDefaultUserData* von *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 Verschiedene Steuerungsebenen der Iteration

Eine Iteration hat je nach dem dynamischen System 2-3 Steuerungsebenen. Auf tiefster Stufe hat man immer einen einzelnen Iterationsschritt:

#### *IterationStep*

Diese Methode ist immer auf tiefster Ebene im spezifischen dynamischen System implementiert.

Eine Stufe höher ist dann der Loop durch viele solche Einzelschritte implementiert:

#### *IterationLoop*

Es kann sein, dass der Loop durch eine bestimmte Anzahl Schritte begrenzt ist. Zum Beispiel im *ClsIterationController*, wo man einen oder 10 Iterationsschritte durchführen kann, oder auch ein ganzes Diagramm der Länge *PicDiagram.Width* erzeugt.

Es kann auch sein, dass der Loop läuft, bis der Benutzer eine Stop-Taste drückt, wie z.B. im *ClSPendulumController*.

Manchmal genügen diese zwei Ebenen nicht, sondern es braucht eine dritte:

#### *PerformIteration*

Während *IterationStep* immer auf Ebene DS oder DSAbstract implementiert ist, sind *IterationLoop* bzw. *PerformIteration* immer beim Form Controller implementiert.

Beispiel:

Bei der Erzeugung des Feigenbaum-Diagrammes muss auf höchster Ebene in der Methode *PerformIteration* das Diagramm für jeden Pixelpunkt in x-Richtung und damit für einen festen Parameter a durchgegangen werden.

Für jeden solchen Wert von a müssen dann eine gewisse Anzahl Iterationsschritte in *IterationLoop* durchgeführt werden, damit man für dieses a die auftretenden Funktionswerte in y-Richtung plotten kann. Bei jedem solchen Iterationsschritt wird *IterationStep* ausgeführt.

Beispielcode 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
```

Und eine logische Ebene tiefer:

```

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

```

Und schliesslich auf tiefster Ebene und im spezifischen DS (hier 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

```

Hier wird der *IterationStep* als Funktion *F(x)* bezeichnet.

Eine Frage in diesem Zusammenhang ist, auf welcher Ebene in die Grafiken *PicDiagram* bzw. *BmpDiagram* gezeichnet werden soll. Das kann je nach dem dynamischen System und der jeweiligen Form verschieden sein. Beim Billard zum Beispiel können beliebig viele Billardbälle auf dem Tisch platziert werden. Dann ist es auch sinnvoll, dass jeder Ball «seine» Bahn in die Grafik zeichnet. Hier erfolgt der Zugriff auf die Grafik also auf Ebene *IterationStep*. Beim Histogramm hingegen sollen viele Iterationsschritte durchgeführt werden und dann eine «Summe» der Hits in einem kleinen Intervall in der Grafik dargestellt werden. Hier erfolgt der Zugriff auf die Grafik also auf Ebene *IterationLoop*.

### 3.3 Asynchrone Iterationen und Performance

Der Laptop, auf welchem der Simulator entwickelt wurde, ist ein Lenovo mit einem 13th Gen Intel(R) Core(TM) i7-1370P 1.90 GHz als Prozessor. Dieser nutzt mehrere parallele Kerne. Wenn man nun eine Methode asynchron «ausserhalb» des Hauptthreads programmiert, muss diese nicht auf allfällige I/O Operationen oder andere Ereignisse warten. Die Rechenzeit kann effizient genutzt werden. Damit wurde z.B. die Rechenzeit für die Generierung einer Juliamenge (konkret dem «Seepferdchen») von rund 2 Minuten auf rund 4.6 Sekunden gesenkt.

Der entsprechende Code ist (in *ClsJuliaController*) beim Start:

```

'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.IsActive = False
            MyForm.BtnStart.Text = LM.GetString("Continue")

```

Und weiter unten 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.IsActive = True
End If
End Sub

```

Die Methode *PerformIteration* ist dann als Task implementiert:

```

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)
    End If

    Do
        ExaminatedPoints += 1
        IterationLoop()

```

Und übergibt die Kontrolle dem *IterationLoop*:

```

Do
    ExaminedPoints += 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 ExaminedPoints Mod 100 = 0 Then      I
        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

```

Welcher als «normale» Methode bereitsteht.

Die `Await Task.Delay` Anweisung ermöglicht, dass der HauptThread z.B. auf die Betätigung der Stopp-Taste reagieren kann.

Was zu einer wesentlichen Verminderung der Performance führt, ist auch das Nachführen von Protokollen oder Einträgen in Wertelisten. Deshalb ist dies optional. Bei der Newton Iteration für die dritten Einheitswurzeln beträgt die Generierung der Bassins mit Protokoll über drei Minuten. Ohne Protokoll dauert sie weniger als 5 Sekunden!

### 3.4 Definition der Startparameter durch den Benutzer

Der User kann Startparameter festlegen, indem er in den entsprechenden TextBoxen Einträge macht. Durch eine Taste `BtnTakeOver` werden diese Einträge direkt an das dynamische System übergeben und je nach Art desselben, wird es direkt auf dem `PicDiagram` in dieser Startposition platziert (z.B. beim Pendel). Spätestens beim Start der Iteration, werden die Parameter nochmals an das dynamische System übergeben.

Wenn es im Kontext Sinn macht, ist eine Alternative, dass der Benutzer die Startposition mit gedrückter linker Maustaste festlegen kann. Das ist der Fall beim Pendel, den Numerischen Methoden und dem Billard. Die entsprechenden Textfelder mit den Startparametern werden dann direkt nachgeführt und das dynamische System im `PicDiagram` platziert.

Wenn die Iteration im Status `Stopped` ist, dann setzt das Drücken der linken Maustaste den Parameter `IsMouseDown` auf `true`. Damit wird aktiviert, dass nun bei der Mausbewegung die Position des dynamischen Systems gleichzeitig an die Position der Maus geschoben wird, vorausgesetzt, dass es nicht schon vorher positioniert wurde. Das heisst, dass der Parameter `IsStartParameterSet` noch auf `false` ist. Dabei werden die Regeln des dynamischen Systems berücksichtigt. Z.B. muss beim Billard der positionierte Billardball immer auf dem Rand des Billardtisches liegen. Wird die Maustaste losgelassen, dann wird `IsStartParameterSet = true` und `IsMouseDown = false`). Der Parameter `IsStartParameterSet` wird erst wieder auf `false` gesetzt, wenn ein neues dynamisches System ausgewählt wird oder wenn die Iteration zurückgesetzt wird (`ResetIteration`).

Nachfolgend ein typisches Codebeispiel aus *ClsPendulumController*, *IPendulum*, *ClsPendulumAbstract* und *ClsCombinedPendulum*

### *ClsPendulumController*

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

    If IterationStatus = ClsDynamics.EIterationStatus.Stopped Then
        If Not (DS.IsStartparameter1Set And DS.IsStartparameter2Set) Then
            MyForm.Cursor = Cursors.Hand I
            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
            End With
        End If
    End Sub
```

```
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 is 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
    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. Implementierung des Billards

Die Klasse *ClsBilliardTableController* stellt die Verbindung zwischen dem User Interface *FrmBilliardTable* und den unteren Ebenen der Architektur her. Dazwischen befindet sich das Interface *IBilliardTable*. Es wird durch die abstrakte Klasse *IBilliardTableAbstract* implementiert. Die konkrete Realisierung eines Billards erbt von dieser Klasse und muss gewisse ihrer Methoden überschreiben.

Der Billardtisch bzw. die Konkretisierungen der Klasse *ClsBilliardTableAbstract* stellen den Billardtisch zur Verfügung. Er kann sich selbst zeichnen und einen Billardball generieren, welcher «zu ihm passt». Er führt dann eine Collection aller Bälle in *MyBilliardballCollection*.

Für die Billardbälle steht das Interface *IBilliardBall* zur Verfügung. Es wird von der abstrakten Klasse *ClsBilliardBallAbstract* implementiert. Konkrete Billardbälle erben von dieser Klasse und überschreiben gewisse Methoden darin. Ein Billardball führt insbesondere:

- Startparameter und seine Startposition
- Aktuelle Parameter, insbesondere  $t$  und den Stosswinkel  $\alpha$

Er überschreibt folgende Methoden von *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

```

Der Parameter  $C$ , welcher je nach Billard die Form des Billardtisches definiert, kann durch den Benutzer in der Trackbar *TrbParameterC* verändert werden. Der Tisch wird dann unmittelbar neu gezeichnet. Dieser Parameter ist dann entscheidend, um den Übergang von «bravem» Verhalten zu chaotischem Verhalten im C-Diagramm darzustellen. Es ist ein Parameter vom Typ *DS*.

## 4.1 Implementierung des elliptischen Billards

Da immer wieder Winkel zwischen einem Vektor und der positiven x-Achse zu bestimmen sind, wird dies unterstützt durch die Funktion *CalculateAngleOfDirection (DeltaX, DeltaY)* in der Klasse *ClsMathHelperBilliard*. Übergeben werden dabei die Koordinaten des Vektors. Die Rückgabe ist ein Winkel in  $[0, 2\pi[$ . Verwendet wird die Funktion insbesondere zur Bestimmung der Winkel  $\varphi, \psi, \vartheta$  sowie des Parameters  $t$ . Siehe entsprechendes Kapitel der mathematischen Dokumentation.

Die je nach Billard verschiedene Kugelklasse (beim elliptischen Billard die Klasse *ClsEllipseBilliardball*) enthält die allgemeine Logik für die Kugelbewegung. Die Kugelbahn wird laufend in die Bitmap *BmpDiagram* gezeichnet, welche das Image der PictureBox *PicDiagram* ist. Die aktuelle Position der Kugel wird in *PicDiagram* gezeichnet. Durch das Refreshing der Picture Box ist jeweils nur diese Position der Kugel sichtbar, während die Bitmap inklusive Kugelbahn auf dem aktuellen Stand sichtbar ist. Um das zu unterstützen, braucht die Kugel die Referenzen auf *PicDiagram*, *BmpDiagram* und den zugehörigen Grafiktools *PicGraphics*, *BmpGraphics*. Da die Kugel in mathematischen Koordinaten arbeitet und erst die Klasse *ClsGraphicTool* die Umrechnung in Pixeleinheiten besorgen, brauchen

letztere auch die Angabe der mathematischen Wertebereiche für x und y: *MathXInterval*, *MathYInterval*. Das ist das Standardquadrat [-1,1] x [-1,1].

Wenn die Iteration gestartet wird, bewegt sich die *ClsEllipseBilliardball* selbständig innerhalb der Ellipse gemäss den mathematischen Algorithmen, welche in der mathematischen Dokumentation beschrieben sind. Aus dem letzten Stosspunkt und der aktuellen Stossrichtung berechnet die Kugel zuerst den nächsten Stosspunkt. Anschliessend den Tangentenwinkel im nächsten Stosspunkt und daraus resultiert die Richtung für den nächsten Stoss.

Aus den Gleichungen

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

Und

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

Ergibt sich für die Richtung der Kugel nach dem n-ten Stoss direkt:

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

Der «Simulator» arbeitet direkt mit dieser Formel, protokolliert aber den Winkel  $\alpha_n$  zusammen mit dem Parameter  $t_n$  im Phasenportrait. Falls *IsProtocol* aktiviert ist, werden diese Parameter zusätzlich in die ListBox *MyValueProtocol* geschrieben.

Da die Kugel die Regeln für ihre Bewegung selbst kennt, ermöglicht dies, mehrere Kugeln zu instanziieren und deren Bewegung parallel laufen zu lassen. Die Kugeln werden durch ihre Farben unterschieden und es sind fünf verschiedene Farben wählbar. Die Kugelgeschwindigkeit kann dabei verändert werden und wirkt sich auf alle Kugeln aus.

Der folgende Codeausschnitt zeigt den Ablauf während der 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 of 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
```

Die Bezeichnungen stimmen mit der mathematischen Dokumentation überein.

Die spezifischen Berechnungen für den elliptischen Billardball befinden sich in:

- *ParameterOfNextHitpoint*
- *MoveOnSegment*
- *CalculateNextPhi* (hier erfolgt auch das Zeichnen in die Diagramme)
- *CalculatePsi*
- *CalculateAlfa*

Die Funktion *GetNextValuePair* ist für das C-Diagramm bestimmt.

## 4.2 Implementierung des Ovalen Billards

Der Unterschied zum elliptischen Billard besteht lediglich in anderen mathematischen Formeln und der Stossalgorithmen. Beim Berechnen des nächsten Stosspunktes muss die Fallunterscheidung getroffen werden, ob der Ball auf der linken Seite an die halbe Ellipse stösst, oder ob er auf der rechten Seite auf die Kugelhälfte stösst.

## 4.3 Implementierung des Stadium Billards

Der Unterschied zum elliptischen Billard besteht lediglich in anderen mathematischen Formeln und der Stossalgorithmen. Die Berechnung des nächsten Stosspunktes ist aber deutlich aufwendiger, wegen der Fallunterscheidungen die wegen der verschiedenen Abschnitte des Billardtisches nötig sind. Es sind deren vier, je nachdem ob der nächste Stoss im linken oder rechten Halbkreis oder auf dem unteren oder oberen Geradenstück des Billardtisches stattfindet. Damit überhaupt ein Stosspunkt ermittelt werden kann, darf der erste Stosswinkel nicht 0 oder  $\pi$  sein.

## 4.4 Implementierung des C-Diagrammes

Das C-Diagramm zeigt in Richtung x-Achse verschiedene Werte des DS-Parameter  $C$ . Für jeden solchen Parameterwert nimmt der Billardtisch eine gewisse Form an. Vor allem interessant ist hier das Ovale Billard. Nun wird für jeden solchen  $C$ -Wert eine Billardkugel immer vom selben Startpunkt aus und unter demselben Reflexionswinkel gestartet. Bei jedem Stoss wird ein Parameterpaar  $(t, \alpha)$

zurückgegeben. Dies geschieht durch die Methode:

```
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 Endpoint 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
```

Der Benutzer hat vor dem Start einen Parameter ausgewählt, den er beobachten will. Das führt dazu, dass die Werte von diesem Parameter in Richtung y-Achse des C-Diagramms geplottet werden.

Das heisst auch, dass in diesem Fall die Iteration auf drei Stufen erfolgt:

- *PerformIteration*: Gehe alle möglichen C-Werte auf der y-Achse durch
- *IterationLoop*: Führe für jeden C-Wert eine Anzahl Iterationsschritte durch und plotte das Ergebnis im C-Diagramm
- *IterationStep*: Führe einen einzelnen Stoss durch und gibt das nächste Parameterpaar zurück

Damit in das C-Diagramm hineingezoomt werden kann, hat der *ClsCDiagramController* auch eine Instanz des *ClsDiagramAreaSelector*.

## 5. Implementierung der Wachstumsmodelle

### 5.1 N:M Beziehung zwischen Wachstumsmodellen und Darstellungsformen

Das spezielle an den Wachstumsmodellen ist, dass einerseits verschiedene solche Modelle zur Verfügung stehen. Aktuell:

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

Die mathematische Beschreibung dieser Modelle findet man in der mathematischen Dokumentation.

Zusätzlich stehe für die Darstellung verschiedener Aspekte dieser Modelle mehrere Formen zur Verfügung:

- *FrmIteration*: Darstellung der eigentlichen Iteration und dem Funktionsgraphen. Unterstützung für ein vorgegebenes Protokoll und für die Transitivität.

- *FrmHistogramm*: Darstellung des Histogrammes im chaotischen Fall
- *FrmSensitivity*: Unterstützung für die Untersuchung der Sensitivität
- *FrmTwoDimensions*: Für die Darstellung in zwei Dimensionen
- *FrmFeigenbaum*: Für die Darstellung des Feigenbaum Diagrammes im chaotischen Fall

Das heisst, dass zwischen Darstellungsformen und Wachstumsmodellen eine n:m Beziehung besteht.

Das ist auch der Grund, wieso die Kontrolle der Iteration im jeweiligen Form Controller stattfinden muss (und nicht auf eine tiefere Ebene, z.B. *ClsgrowthModelAbstract*) geschoben werden kann.

Jede Form hat also «ihren» Controller. Alle diese greifen auf das Interface *IIteration* zu. Dieses wird von der abstrakten Klasse *ClsgrowthModelAbstract* implementiert. Die konkreten Klassen für die verschiedenen Modelle erben von dieser Klasse und überschreiben gewisse Methoden darin:

```
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
```

Das sind recht wenige Methoden. Hauptsächlich die eigentliche Funktionsvorschrift, sowie der Diffeomorphismus zur Zeltabbildung und seine Inverse.

Damit die Bezeichnungen klar sind, werden die Variablen für die Wachstumsmodelle (ausser die Zeltabbildung) durchgehend mit  $(x, y)$  bezeichnet. Die Variablen für die Zeltabbildung dagegen mit  $(u, v)$ . Das entspricht den Bezeichnungen in der mathematischen Dokumentation.

Bei sämtlichen Berechnungen arbeiten wir in einem Koordinatensystem, welches durch einen Wertebereich für die x-Koordinate und y-Koordinate definiert ist.

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

Ein Punkt  $(x, y)$  in diesem Koordinatensystem wird durch ein Objekt der Klasse *ClsmathPoint* repräsentiert.

Um einen Startwert für ein vorgegebenes Protokoll oder einen vorgegebenen Zielwert zu finden, werden die entsprechenden Konjugations-Transformationen zur Zeltabbildung benützt: Man berechnet zuerst den entsprechenden Startwert für die Zeltabbildung wie im entsprechenden vorhergehenden Abschnitt über die Zeltabbildung beschrieben. Dann erhält man den Startwert für das logistische Wachstum bzw. die normierte Parabel durch Anwendung der entsprechenden Konjugationstransformation, welche in den Abschnitten über diese Iterationen beschrieben sind.

Das sind die Methoden

- *IterationToTentmap*
- *TentmapToIteration*

Die *FrmFeigenbaum* unterstützt die Untersuchung der allfälligen Zyklen in Abhängigkeit eines Parameters  $a$  für die Iteration. Auch hier verwendet die *FrmFeigenbaum* die Methoden und Eigenschaften des Interfaces *IIteration*. Klassen, welche dieses Interface implementieren, enthalten dann die Spezifika der jeweiligen Iteration.

Der Benutzer kann den Parameterbereich, in welchem sich  $a$  bewegt, manuell eingeben. Ebenso kann er für die x-Werte einen Wertebereich eingeben, der festlegt, in welchem Bereich die x-Werte betrachtet werden. Das führt zu einer Skalierung der x-Achse.

Beides kann auch durch eine Auswahl mit gedrückter linker Maustaste geschehen. So hat man die Möglichkeit, interessante Bereich im Feigenbaum Diagramm näher zu untersuchen. Eine detaillierte Beschreibung findet man im «Handbuch».

## 5.2 Implementierung des Feigenbaum Diagrammes

Wie beim C-Diagramm beim Billard entscheidet der Parameter  $a$  bei den Wachstumsmodellen darüber, ob sich das System chaotisch verhält oder nicht. Im Feigenbaum Diagramm wird der Übergang von Ordnung zum Chaos dargestellt. In Richtung x-Achse werden alle möglichen Parameterwerte von  $a$  durchgegangen. Für jedes solche  $a$  wird die Iteration immer mit demselben Startwert  $x_1$  gestartet. Dann wird die Iteration eine Weile laufen gelassen, bis sie sich auf einen attraktiven Zyklus eingependelt hat, falls überhaupt ein solcher existiert. Anschliessend werden die Iterationswerte und damit dieser Zyklus in y-Richtung ins Diagramm geplottet.

Das bedeutet auch hier, dass man drei Ebenen der Iteration hat:

- *PerformIteration*: Gehe alle möglichen  $a$ -Werte auf der y-Achse durch
- *IterationLoop*: Führe für jeden  $a$ -Wert eine Anzahl Iterationsschritte durch und plotte anschliessend den eventuell vorhandenen Zyklus ins Feigenbaum-Diagramm
- *IterationStep*: Führe einen einzelnen Stoss durch und gibt das nächste Parameterpaar zurück

Eine spezielle Rolle spielt das Modell *ClIsMandelbrotReal*. Es dient nur zur Darstellung im Feigenbaum-Diagramm, damit dieses mit der Mandelbrotmenge verglichen werden kann, welche im Abschnitt über Komplexe Iteration beschrieben wird. Da der Definitionsbereich dieses Modells vom Parameter  $a$  abhängt, machen die übrigen Darstellungsformen nicht wirklich Sinn.

## 6. Implementierung der komplexen Iteration

### 6.1 Implementierung der Newton Iteration

Im Unterschied zu den bisherigen dynamischen Systemen werden hier nicht Bahnen von Iterationspunkten geplottet, sondern man untersucht die Einzugsgebiete bzw. Bassins von attraktiven Fixpunkten. Wie in der mathematischen Dokumentation beschrieben, sind Nullstellen von komplexen Polynomen gleichzeitig (super-)attraktive Fixpunkte der Newton Iteration. Letztere ist für drei Modelle implementiert:

- Einheitswurzeln, bzw. Nullstellen der Polynome  $p_n(z) = z^n - 1$
- «Invertierte» Einheitswurzeln: Das ist das Verhalten der Iteration im unendlich fernen Punkt (siehe math. Dok.)
- Nullstellen eines Polynoms dritten Grades

Der Ablauf der Iteration ist dann immer dieselbe: Man geht alle Punkt in der komplexen Ebene durch. Für jeden Punkt führt man solange Iterationsschritte durch bis klar ist, gegen welchen Fixpunkt die Iteration konvergiert, oder ob sie nicht konvergiert, falls man eine bestimmte Anzahl Iterationsschritte übertrifft. Details dazu findet man in der math. Dokumentation.

Der *ClIsNewtonIterationController* stellt die Verbindung vom User Interface zu den tieferen Ebenen sicher und kontrolliert u.a. den Ablauf der Iteration. Er setzt auf das Interface *INewton* auf. Dieses wird von der Klasse *ClIsNewtonAbstract* implementiert. Die konkreten Klassen (also

*ClsUnitRootCollection*, und *ClsPolynom3*) erben von dieser Klasse und überschreiben die expliziten Formeln der Iteration:

```

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

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

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

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

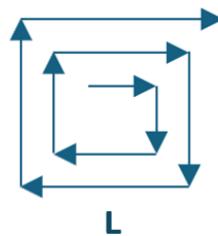
8 Verweise
Public MustOverride Function Denominator(Z As ClsComplexNumber) As ClsComplexNumber
```

Auch hier findet die Iteration auf drei Ebenen statt:

- *PerformIteration*: Bestimme einen zu untersuchenden «Corner» in der komplexen Ebene (siehe Erklärung weiter unten). Das sind zwei Seiten eines Rechtecks in der komplexen Ebene.
- *IterationLoop*: Untersuche jeden Punkt auf diesen Rechteckseiten indem der Punkt z als Startwert für die Newton Iteration gewählt wird.
- *IterationStep*: Führe für diesen Startwert z so viele Iterationsschritte durch, bis klar wird, gegen welche Nullstelle die Iteration konvergiert (wenn überhaupt). Kriterien dafür sind in der mathematischen Dokumentation hergeleitet. Anschliessend erhält der Startpunkt dieselbe Farbe wie die «seine» Nullstelle.

*PerformIteration* und *IterationLoop* befinden sich in *ClsNewtonIterationController*. *IterationStep* in *ClsNewtonAbstract*.

Die Idee ist, dass man den Ausschnitt der komplexen Ebene nicht zeilenweise von rechts nach links und oben nach unten durchgeht, weil die interessanten Ausschnitte in der Mitte des Diagrammes sind. Deshalb wurde folgender Algorithmus gewählt:



Die Iteration startet in der Mitte und geht dann einen Schritt nach rechts und nach unten. Dann wird die Länge des Rechtecks um +1 erhöht und man geht zuerst nach links und dann hinauf. Dann wird die Länge des Rechtecks wieder um +1 erhöht und man geht nach rechts und dann nach unten. Und so weiter.

Zwei aufeinanderfolgende Rechteckseiten (=ein *ExamineCorner*) sind ein Schritt in *PerformIteration*. Hier ein Codeausschnitt aus dieser Methode.

```

Do
    ExaminatedCorner += 1
    IterationLoop()

    If p >= MyForm.PicDiagram.Width Or q >= MyForm.PicDiagram.Height Then
        IterationStatus = ClsDynamics.EIterationStatus.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.EIterationStatus.Interrupted _
    Or IterationStatus = ClsDynamics.EIterationStatus.Stopped

```

Ein *IterationLoop* ist die Abarbeitung von zwei aufeinanderfolgenden Rechteckseiten. Das heisst, man geht jeden Punkt einer Rechteckseite durch und wählt ihn als Startpunkt für die Iteration. Hier ein Codeausschnitt aus dieser Methode.

```

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

```

Ein *IterationStep* ist hier die Iteration eines Startpunkten z bis dessen Konvergenz abgeklärt ist. Ein Codeausschnitt aus dieser Methode.

```

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)
    End Sub

    '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

```

Verantwortlich für das Färben der Bassins der Nullstellen ist die Methode *GetBasin*. Sie sorgt dafür, dass das Einfärben auch abhängig von der Anzahl der Schritte ist, bis die Konvergenz zur nullstelle sichergestellt ist. Die entsprechenden Berechnungen wurden experimentell optimiert. Hier ein Auszug der relevanten Codestelle:

```

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 I

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

```

Die Farbe der *Brush* welche von *Root.GetColor* zurückgegeben wird, wurde systematisch definiert: Die Rot/Blau/Grün Anteile der Farbe sind abgestuft je nach dem Index der Einheitswurzel und dieser hängt vom Exponenten  $n$  ab. Diese Farben sind so gewählt, dass z.B. die 12. Einheitswurzeln Farben besitzen, welche in der Reihenfolge zueinander passen.

Damit man besser mit komplexen Zahlen rechnen kann, ohne dass man immer auf deren Real- und Imaginärteil zurückgreifen muss, implementiert die Klasse *ClsComplexNumber* gewisse Operationen mit komplexen Zahlen wie zum Beispiel deren Multiplikation, Addition oder auch die Bildung ihrer  $n$ -ten Potenz. Das ermöglicht, dass algebraische Ausdrücke durch eine einfache Art Parser codiert werden können. Zum Beispiel ist der Code für die Newton Iterierte der Einheitswurzeln:

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

Codiert:

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

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

## 6.2 Implementierung der Julia- und Mandelbrotmenge

Der Controller für die *FrmJulia* ist der *ClsJuliaIterationController*. Er kommuniziert mit den unteren Schichten via dem interface *IJulia*. Dieses wird von der abstrakten Klasse *ClsJuliaAbstract* implementiert. Die konkreten Realisierungen der Juliamenge und Mandelbrot Menge erben von dieser Klasse und überschreiben nur eine spezifische Methode: *IterationStep*.

Die Generierung des Julia- und Mandelbrotmenge folgt demselben Schema wie die Newton Iteration. Der Algorithmus, um den Ausschnitt der komplexen Ebene zu untersuchen, ist derselbe: Der Start ist in der Mitte des Diagrammes und anschliessend wird dieses spiralförmig «aufgerollt».

Die verfügbaren Operationen der Klasse *ClsComplexNumber* sind auch hier wesentlich für einen einfachen Code beim Rechnen mit komplexen Zahlen.

Speziell hier ist die Farbgebung der Julia- bzw. Mandelbrotmenge. Die Farbe hängt davon ab, wie schnell ein Startpunkt gegen  $\infty$  strebt. Das heisst, nach wie vielen Iterationsschritten ist klar, dass er

das tut. Hier ein Codeausschnitt aus *ClsJuliaPN*, also dem iterierten Polynom für die Juliamenge mit dem Exponenten  $n$ :

```

1      If MyIsUseSystemColors Then
2          MyBrush = StandardColors.GetSystemBrush(Steps)
3          ColorIndex = 1
4      Else
5
6          ColorIndex = Steps / MaxSteps
7
8          'to keep the brightness higher
9          ColorIndex = Math.Min(1, ColorIndex * 2)
10         ColorIndex = Math.Pow(ColorIndex, 1 / 5)
11
12         MyBrush = New SolidBrush(Color.FromArgb(255, CInt(255 * ColorIndex * MyRedPercent),
13                                         CInt(255 * ColorIndex * MyGreenPercent), CInt(255 * ColorIndex * MyBluePercent)))
14     End If

```

Man sieht, dass die «Systemfarben» in der Klasse *ClsSystemBrushes* bereitgestellt werden, denn *StandardColors* ist eine Instanz derselben. Diese Systemfarben wurden ebenfalls experimentell optimiert.

## 7. Implementierung Mechanik

### 7.1 Implementierung der numerischen Methoden

Die Architektur des Bereichs «Numerische Methoden» ist nach denselben Prinzipien aufgebaut, wie die vorhergehenden Implementierungen. Die Klasse *ClsNumericMethodController* enthält die Logik für das User Interface und stellt die Verbindung zu den unteren Schichten her. Sie greift auf das Interface *INumericMethod* zu. Dieses wird implementiert durch *ClsNumericMethodAbstract*. Konkrete Realisierungen erben dann von dieser Klasse und überschreiben nur die Methode *Iteration*.

Diese Methode ist dann bei jeder Realisierung im wesentlichen ein numerisches Verfahren zur Approximation des Federpendels.

Da die Bewegung eines realen Federpendels mit der Approximation verglichen wird, stellt sich hier die Herausforderung der Synchronisierung:



Das reale Federpendel soll gleich schnell schwingen wie die Approximation, welche mit einem numerischen Verfahren berechnet wird. Das wird im *ClsNumericMethodController* durch die Methode *SetStepWidthAB* sicher gestellt. Sie berechnet die Schrittweite des numerischen Verfahrens und

synchronisiert diese mit der Schrittweite der «Zeit» im echten Federpendel. Dabei kann das Diagramm auch gestrechzt werden.

Hier Ausschnitte aus dem 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

```

Bei der Darstellung im Diagramm wird die Bitmap *BmpDiagram* kopiert, nach rechts geschoben und wieder in *BmpDiagram* eingefügt (siehe auch Abschnitt «Graphik im User Interface»).

## 7.2 Implementierung der verschiedenen Pendel

Die Klasse *ClsPendulumController* enthält die Logik des User Interface und stellt die Verbindung nach unten her. Sie greift auf das Interface *IPendulum* zu, welches von der Klasse *ClsPendulumAbstract* implementiert wird. Konkrete Realisierungen der verschiedenen Pendel erben dann von dieser Klasse und überschreiben gewisse Methoden darin.

```

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()

```

Speziell ist hier die Verwaltung der Parameter. Man weiss nicht, wie viele nötig sind, um die Bewegung eines Pendelsystems zu beschreiben. Maximal stehen 6 solcher Parameter zur Verfügung. Das würde für ein dreifach gekoppeltes Pendel reichen. Da diese vom konkreten Pendelsystem abhängen, werden sie in der *New* Methode des jeweiligen Systems definiert. Hier ein Beispiel aus *ClsDoublePendulum*.

```

Public Sub New()
    Yθ = 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))

```

Beim Laden des Pendelsystems wird dann das GUI dynamisch angepasst. Hier der entsprechende Code in der Methode *InitializeMe* 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
    End With

```

Zur Iteration folgende Bemerkungen:

- Die relevanten Iterationsparameter sind in Anlehnung an die Formeln des Runge Kutta-Verfahrens zum Beispiel beim Doppelpendel  $u_1, v_1, u_2, v_2$ . Deren Startwert wird auch durch *SetAndDrawStartParameter1,2* gesetzt.
- Am Beginn eines Iterationsschrittes wird  $OldPosition = Position$  gesetzt.  $OldPosition$  ist also die alte Position des Pendels vor dem Iterationsschritt.
- Für die Iteration werden beim Doppelpendel in Anlehnung an die mathematischen Formeln ein *ClsVector*  $x(3)$  gebraucht. Er spielt die Rolle der einzelnen  $\vec{x}_{in}$ .
- $k_{11}, k_{12}, k_{13}, k_{14}$  wird als *ClsVector(3)* geführt. Ebenso  $k_{21}, h_{11}$  und  $h_{21}$ .
- Am Ende jedes Iterationsschrittes wird die aktuelle Position des Pendels auf Grund von *Position* gezeichnet und die Pendelbahn in *DrawTrack* auf Grund von *OldPosition* und *Position*.

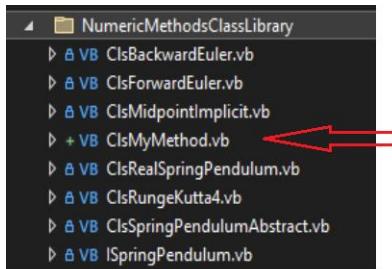
Da sich die einzelnen Pendelsysteme chaotisch verhalten, reagieren sie sehr sensitiv auf Änderungen des Startwertes und damit auch auf die Ungenauigkeiten des numerischen Approximationsverfahrens. Das Verhalten, das im Diagramm gezeigt wird, ist also schon nach wenigen Schritten nicht realistisch. Was aber immerhin überprüft werden kann ist, ob die Gesamtenergie erhalten bleibt. Die Methode *GetEnergy* auf Ebene konkretes Pendelsystem berechnet die aktuelle Energie und vergleicht sie mit der Energie am Start. Die Energie wird dann in einem Balken unterhalb des Protokolls angezeigt und erscheint grün, wenn die Abweichung unterhalb 10% von der Startenergie ist. Sonst erscheint sie bei zu hoher Energie rot und bei zu tiefer Energie in Lila.

## 8. Implementierung eigener Systeme im «Simulator»

Falls der Benutzer des «Simulator» eigener Varianten der bestehenden implementierten Systeme programmieren möchte, ist dies sehr einfach möglich. Wir wollen das am Beispiel einer eigener Variante einer numerischen Methode erläutern.

### 1. Schritt

Erstelle eine eigene Klasse *ClsMyMethod* und füge sie zum Ordner *NumericMethodsClassLibrary* hinzu:



Wichtig dabei ist, sich an die Konvention zu halten, dass Klassennamen mit dem Präfix *Cls* beginnen.

## 2. Schritt

Diese Klasse muss nun die Klasse *ClsSpringPendulumAbstract* übernehmen. Der einzige Codeteil, welcher gegenüber den bestehenden Klassen ändert, ist der Algorithmus der numerischen Methode:

```

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 Component(0) holds the "time" t with 2*pi period
                .Component(0) = .Component(0) Mod CDec(2 * Math.PI)

            End With
        End Sub
    End Class

```

Der Code könnte dann wie oben gezeigt aussehen.

Weitere Schritte sind nicht nötig. Insbesondere wird in der *FrmNumericMethods* die Combobox mit der Auswahl der numerischen Methoden durch *Reflection* gefüllt. Da ist also keine Änderung nötig. In der Combobox wird die eigene Methode als *MyMethod* aufgeführt. Das heißt, aus dem Namen wird das Präfix *Cls* herausgeschnitten.

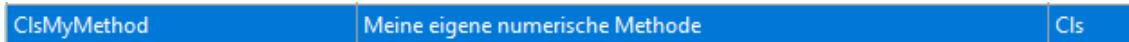
## 3. Schritt (optional)

Wenn man die Bezeichnungen in Deutsch und Englisch anders wählen will als *MyMethod*, kann man einen entsprechenden Schlüssel in den Resourcenfiles *LabelsEN* und *LabelsDE* eintragen. Das geschieht wie folgt:

In LabelsEN:

ClsMyMethod	My own numeric method	Cls
-------------	-----------------------	-----

In LabelsDE:



In beiden Fällen ist der Key (Name des Eintrags) der Klassenname, der Wert die eigene gewählte Bezeichnung und der Kommentar 'Cls'.

Für andere eigene Erweiterungen bei den Wachstumsmodellen, der komplexen Iteration oder dem Billard gilt genau das analoge. Eigene Implementierungen müssen die entsprechende abstrakte Klasse überbauen und gewisse Methoden darin überschreiben.

#### 4. Schritt (optional)

Erstellen eines eigenen MSI-files:

Im Simulator ist ein Setup Projekt vorhanden. Mit «neu erstellen» erzeugt dieses ein MSI-File für die Installation des Simulators.

