Zeus Math Library

Zeus Framework

Autor	Benjamin Hadorn; Martin Abbühl
E-Mail	b_hadorn@bluewin.ch
Ablage/Website	http://www.xatlantis.ch/zeusmath.html
Datum	01.06.08
Version	0.6.3

Inhaltsverzeichnis

1 Einleitung

Die Zeus-Math-Bibliothek ist eine umfangreiche Sammlung von mathematischen Funktionen und Klassen. Wegen der Grösse der Sammlung wurden die Klassen aus der Zeus-Base-Bibliothek ausgelagert. Diese Bibliothek baut aber auf der Zeus-Base-Bibliothek auf.

Die Bibliothek besteht aus folgenden Paketen:

- Allgemeine mathematische Klassen
  - Komplexe Zahlen,
  - Matrizen und Vektoren
  - Permutationen
  - Graphen-Theorie
  - Gleichungssysteme
- Vektor-Geometrie
  - Geraden
  - Ebenen
- Logik und Fuzzy Logik
- Berechnungen
  - Fourier Transformation
  - Integration und Differenzierung
- Künstliche Intelligenz
  - Genetische Algorithmen
  - Neuronale Netzwerke
- Stochastik

1.1 Verwendung der Bibliothek

Die Zeus-Math-Bibliothek ist wie die Zeus-Base-Bibliothek eine Dynamic Link Library, welche bei Prozessstart vorhanden sein muss und nicht nur optional zur Laufzeit geladen werden kann. Diese Art der Bindung ermöglicht die Erzeugung von Objekten auf dem Stack, wie beispielsweise Objekte der Klasse `TString` aus ZeusBase.dll. Im Zeus-Math gibt es dementsprechend Stack-Objekte für Klassen aus der Vektorgeometrie, was beispielsweise die direkte Addition von Vektoren ermöglicht – via Operatoren-Überladung.

2 Allgemeine mathematische Klassen

Die allgemeinen Klassen der Zeus-Math Bibliothek befinden sich in dem System-Verzeichnis.

2.1 Zahlen und Datentypen

Das Zeus-Math bietet verschiedene Zahlen und Datentypen an. Komplexe Zahlen und reelle Werte bilden dabei die Basis.

API	
`Float`	Gibt einen reellen Wert an. Dieser Datentyp wird durch das Zeus-Base definiert und ist für eine 32bit Plattform ein `double` gleichzusetzen.
`Int`	Gibt einen ganzzahligen Wert an. Dieser Datentyp wird wie der Float durch das Zeus-Base definiert und entspricht auf einer 32bit Plattform einem `long`.
`TFloat`	Die Klasse bietet eine umfassende Sammlung von Methoden zum Prüfen und Verwalten eines reellen Wertes. Mehr dazu ist im Dokument [1] zur Zeus-Base Bibliothek zu lesen `zeusbase/System/Float.h`
`TInt`	Die Klasse bietet Methoden zum Verwalten eines ganzzahligen Wertes an. Sie ist eben falls in [1] beschrieben. `zeusbase/System/Int.h`
`TComplex`	Komplexe Zahlen und deren Operatoren sind in dieser Klasse implementiert. `zeusmath/System/Complex.h`
`TAngle`	Die Klasse kapselt einen Winkelwert und implementiert Umsetzungen in verschiedene Winkelsysteme. `zeusmath/System/Angle.h`

2.1.1 Komplexe Zahlen

Komplexe Zahlen bestehen aus einem reellen und einem imaginären Teil bei der kartesischen Darstellung oder einem Argument und dem absoluten Wert bei der Polar-

Darstellung. Die Klasse `TComplex` unterstützt beide Darstellungen, intern werden aber nur Imaginär- und Realteil gespeichert.

API	`zeusmath/Systen/Complex.h`

	`void assign(const TComplex& rInpar);`
	`const TComplex& operator=(const TComplex& rX);`
	
	Einen komplexen Wert zuweisen.

	`bool equals(const TComplex& rInpar,`
	`  Float fPrecision = FLOAT_PRECISION) const;`
	`bool operator==(const TComplex& z) const;`
	
	Prüfen ob zwei komplexe Zahlen gleich sind. Mit equals kann zusätzlich eine Präzisionsangabe gemacht werden.

	`const Float& getReal() const;`
	`const Float& getIm() const;`
	
	Gibt den imaginären, bzw. den reellen Teil der komplexen Zahl zurück.

	`const Float getAbsValue() const;`
	`const Float getArgument() const;`
	
	Gibt den absoluten Wert oder das Argument der komplexen Zahl zurück. Dies entspricht der Polar-Darstellung

	`bool isReal() const;` Die komplexe Zahl besitzt nur einen reellen Teil und ist somit eine reelle Zahl
	`bool isImaginary() const;` Die Zahl besitzt nur einen imaginären Teil.
	`bool isComplex() const;` Ist das Gegenteil von `isReal()`.

Es existieren verschiedene mathematische Operatoren zu den komplexen Zahlen. Hier einige Beispiele:

Operationen mit komplexen Zahlen

```
#include <zeusmath/System/Complex.h>

USING_NAMESPACE_Zeus

TComplex c1(1,1);
TComplex c2(1); //real number

TComplex c3 = c1 * c2 + 6;
c3 *= TComplex(1,9, true); //values are in polar form
```

Tabelle 1: Beispiele von Operationen mit komplexen Zahlen.

Die Klasse bietet auch einige mathematische Funktionen als statische Methoden an:

API	zeusmath/Systen/Complex.h
	`static TComplex log(const TComplex& z);` Berechnet den Logarithmus zur Basis 10 (log Funktion der API).
	`static TComplex log(const TComplex& z, const Float& fBasis);` Berechnet den Logarithmus zu einer beliebigen Basis..
	`static TComplex sqrt(const TComplex& z);` Berechnet das Quadrat einer komplexen Zahl.
	`static TComplex pow(const TComplex& zBase, const TComplex& zExponent);` Berechnet eine beliebige Potenz einer komplexen Zahl.

2.1.2 Winkelsysteme

Folgende Winkelsysteme stehen durch die Klasse `TAngle` zur Verfügung:

- Radiant (0 bis 2 π)
- Grad (0 bis 360)

Folgende Methoden stehen durch die implementierte Schnittstelle `IAngle` zur Verfügung:

API	zeusmath/Systen/Angle.h
	`Float MQUALIFIER getRadians() const;` Gibt den Winkel als Radiant zurück.
	`Float MQUALIFIER getDegrees() const;` Gibt den Winkel als Grad zurück.

	`void MQUALIFIER setRadians(const Float& fAngle);`
	`void MQUALIFIER setDegrees(const Float& fAngle);`
	
	Einen Wert als Radiant bzw. Grad zuweisen.

	`void MQUALIFIER assign(const IAngle& Angle);` Einen Winkel zuweisen.
	`bool MQUALIFIER isEqual(const IAngle& Angle) const` Prüfen, ob zwei Winkel gleich sind.

API	zeusmath/Systen/Angle.h

	`Float MQUALIFIER cos() const;`
	`Float MQUALIFIER sin() const;`
	`Float MQUALIFIER tan() const;`
	
	Winkel in Kosinus, Sinus oder Tangens umrechnen.

2.2 Konstanten

Die Klasse `TConstants` besitzt statische Konstanten, die für mathematische Berechnungen von Bedeutung sind.

API	zeusmath/Systen/Constants.h
	`const Float TConstants::Pi;` Entspricht der Zahl π (PI)
	`const Float TConstants::e;` Entspricht der eulerschen Zahl e.
	`const Float TConstants::RadiansToDegrees;` Umrechnungsfaktor von Radiant nach Grad.
	`const Float TConstants::DegreesToRadians;` Umrechnungsfaktor von Grad nach Radiant.

Die Konstanten werden nach Bedarf erweitert und ergänzt.

2.3 Matrizen

Folgende Klassen dienen zur Darstellung der Matrix:

API	
`TMatrix`	Bildet eine n*m Matrix ab `zeusmath/System/Matrix.h`

API	
`TSquareMatrix3`	Bildet eine 3*3 Matrix ab. Diese Klasse ist optimiert für geometrische Berechnung im 3D Raum. `zeusmath/System/SquareMatrix3.h`
`IMatrix`	Schnittstelle einer n*m Matrix `zeusmath/System/Interfaces/IMatrix.hpp`
`ISquareMatrix3`	Schnittstelle einer quadratischen 3x3 Matrix. `zeusmath/System/Interfaces/ISquareMatrix3.hpp`

2.3.1 Allgemeine Matrix

Die Klasse `TMatrix` ist ein Wert-Typ und kann auf dem Stack erzeugt werden. Die Matrix Klasse bietet viele Methoden. Folgende Methoden sind in der Schnittstelle `IMatrix` enthalten, welche von der Klasse implementiert werden:

API `zeusmath/Systen/Matrix.h`

`void MQUALIFIER assign(const IMatrix& rMatrix);`
`const TMatrix& operator=(const TMatrix& rMatrix);`
 	
Einer Matrix kann eine andere Matrix zugewiesen werden. Dazu gibt es sowohl die Methode assign als auch ein Zuweisungsoperator

`bool MQUALIFIER equals(const IMatrix& rMatrix,`
`  Float fPrecision = FLOAT_PRECISION) const;`
`bool operator==(const IMatrix& rMatrix) const;`
 	
Vergleicht eine Matrix mit einer anderen. Diese Funktionalität wird sowohl durch die Methode equals als auch durch einen Vergleichsoperator zur Verfügung gestellt.

`bool MQUALIFIER calcInverse(IMatrix& rInverseMatrix) const;` Berechnet die inverse Matrix und gibt diese zurück.
`void MQUALIFIER calcTransposed(IMatrix& rTransposedMatrix) const;` Transponiert eine Matrix und gibt das Resultat zurück.
`Float MQUALIFIER getCell(Int iRow, Int iColumn, bool* pbError = NULL) const;` Gibt eine Zelle der Matrix zurück.

`bool MQUALIFIER getRow(Int iRow, IVector& rVector) const;`
`bool MQUALIFIER getColumn(Int iColumn, IVector& rVector) const;`
`bool MQUALIFIER getDiagonalVector(IVector& rDiagVector) const;`
 	
Gibt den Zeilen-, den Spalten- oder der diagonale Vektor einer Matrix zurück.

API	zeusmath/Systen/Matrix.h
	`void MQUALIFIER initAsIdentityMatrix();` Initialisiert die Matrix als Identitätsmatrix.
	`void MQUALIFIER initAsZeroMatrix();` Initialisiert die Matrix als Null-Matrix
	`void MQUALIFIER addConstant(Float fConstant)` Addiert jede Zelle mit einer Konstanten.
	`void MQUALIFIER multiplyConstant(Float fConstant);` Multipliziert jede Zelle mit einer Konstanten.
	`Retval MQUALIFIER setDimensions(Int iRows, Int iColumns);` Verändert die Grösse der Matrix. Der Inhalt wird gelöscht.
	`bool MQUALIFIER isSquaredMatrix() const;` Prüft ob die Matrix quadratisch ist.
	`bool MQUALIFIER isInvertible() const;` Prüft ob die Matrix invertiert werden kann
	`bool MQUALIFIER isIdentityMatrix() const;` Prüft ob die Matrix der Einheitsmatrix entspricht.
	`bool MQUALIFIER isZeroMatrix() const;` Prüft ob die Matrix eine Null-Matrix ist.

Folgendes Beispiel zeigt die Verwendung von Matrizen:

Operationen mit Matrizen

```
#include <zeusmath/System/Matrix.h>
#include <zeusmath/System/Vector.h>

USING_NAMESPACE_Zeus

TMatrix m1(L"[1,2,3],[-1,2,-3],[8,1,0]");
TVector v1(L"1,2,3");

TVector v2 = m1 * v1;

m1.transpose();
m1 =+ 5;

TVector v3 = m1 * v2;
```

Tabelle 2: Beispiel einer Multiplikation von Vektoren mit Matrizen.

Hilfsklassen dienen dem der Berechnung und dem Strukturieren von anderen Klassen.

2.3.2 3x3 Matrix

Die 3x3 Matrix ist optimiert für den 3D Raum. Die interne Datenstruktur ist optimal für Geschwindigkeit ausgelegt. Es wird deshalb empfohlen diese Klasse für geometrische Berechnungen zu verwenden (Siehe 3DVector im Kapitel zu Geometrischen Klassen).

Die Klasse implementiert neben der IMatrix- auch die ISquareMatrix3-Schnittstelle.

API	`zeusmath/Systen/SquaredMatrix3.h`

	`void MQUALIFIER setValues(`
	`  const Float& f11, const Float& f12, const Float& f13,`
	`  const Float& f21, const Float& f22, const Float& f23,`
	`  const Float& f31, const Float& f32, const Float& f33);`
	
	Setzt alle 9 Werte der Matrix.

	`void MQUALIFIER initAsRotationXMatrix(const IAngle& rAngle);`
	`void MQUALIFIER initAsRotationYMatrix(const IAngle& rAngle);`
	`void MQUALIFIER initAsRotationZMatrix(const IAngle& rAngle);`
	
	Initialisiert die Matrix als Rotationsmatrix um die X-Achse, Y-Achse oder Z-Achse.

2.4 Vektoren

Folgende Vektorklassen sind im System-Paket enthalten:

API	
`IVector`	Schnittstelle eines allgemeinen Vektors `zeusmath/System/Interfaces/IVector.hpp`
`TVector`	Diese Klasse implementiert einen allgemeinen Vektor. `zeusmath/System/Vector.h`
`TVector2D`	Vereinfachter Vektor für den 2D Raum. Siehe Kapitel zu geometrischen Klassen `zeusmath/Geometry/Vector2D.h`

API	
`TVector3D`	Vereinfachter Vektor für den 3D Raum. Siehe Kapitel zu geometrischen Klassen. `zeusmath/Geometry/Vector2D.h`

In diesem Kapitel betrachten wir nur den allgemeinen Vektor. Die vereinfachten Vektoren für den 2D und 3D Raum werden im Kapitel für geometrische Klassen näher beschrieben.

2.4.1 Allgemeiner Vektor

Die Klasse `TVector` implementiert einen allgemeinen Vektor mit variabler Anzahl von Komponenten (Dimensionen).

Folgende Methoden werden durch die Schnittstelle IVector vorgegeben:

API `zeusmath/Systen/Vector.h`

`void MQUALIFIER assign(const IVector& rVector);`
`void MQUALIFIER assignValues(const Float* apValues, Int iSize);`
 	
Einen Vektor zuweisen. Die Komponenten und Dimensionen werden übernommen.

`bool MQUALIFIER equals(const IVector& rVector,`
`Float fPrecision = FLOAT_PRECISION) const;`
 	
Prüft ob zwei Vektoren gleich sind. Zum Vergleichen kann eine Präzision angegeben werden.

`Float MQUALIFIER getComponent(Int iIndex) const;` Gibt eine Komponente einer bestimmten Dimension zurück.
`Int MQUALIFIER getDimension() const;` Gibt die Anzahl Dimensionen zurück.

`Float MQUALIFIER getNorm() const;`
`Float MQUALIFIER getLength() const;`
 	
Gibt die Länge des Vektors zurück.

`void MQUALIFIER setDimension(Int iValue);` Verändert die Dimensionen. Diese Methode löscht alle Komponenten (werden auf Null gesetzt).

`void MQUALIFIER add(const IVector& rVector);`
`void MQUALIFIER subtract(const IVector& rVector);`
 	
Addiert bzw. Subtrahiert einen Vektor.

API	zeusmath/Systen/Vector.h
	`Float MQUALIFIER calcScalarProduct(const IVector& rVector) const;` Berechnet das Skalarprodukt zweier Vektoren

	`void MQUALIFIER multiply(const Float& f);`
	`void MQUALIFIER divide(const Float& f);`
	
	Multipliziert bzw. Dividiert den Vektor mit einem Skalar.

2.5 Lösen von Gleichungen und Gleichungssystemen

Folgende Gleichungssysteme können mit dem mathematischen Framework gelöst werden:

- Lineare Gleichungssysteme (2,3 und beliebige Anzahl Variablen)
- Quadratische Gleichungen
- Finden einer Nullstelle einer beliebigen Funktion.

Dazu wurden 2 Klassen entwickelt:

API	
`TEquationsSolver`	Hauptklasse zum Lösen von Gleichungen und Gleichungssystemen `zeusmath/System/EquationsSolver.h`
`TLinearAlgebraHelper`	Diese Hilfsklasse implementiert die Berechnung von Determinanten zum Lösen von linearen Gleichungssystemen. `zeusmath/System/LinearAlgebraHelper.h`

Die Hauptklasse ist der TEquationsSolver. Folgende Methoden stehen zur Verfügung:

API	zeusmath/Systen/EquationsSolver.h

	`static bool solveLinearEquations(const TMatrix& rAVars,`
	`const TVector& rDVars,`
	`TVector& rResults);`
	
	Löst ein beliebiges lineares Gleichungssystem.

API	zeusmath/Systen/EquationsSolver.h
	`static bool solveLinearEquations2( const Float& a1, const Float& b1, const Float& c1, const Float& a2, const Float& b2, const Float& c2, Float& x, Float& y);` Löst ein Gleichungssystem mit 2 Variablen. Es existiert auch eine Variante speziell für 3 Variablen. Diese Varianten sind optimiert und sollten eingesetzt werden, wenn 2 bzw. 3 Variablen gefunden werden sollten..
	`static Int solveQuadraticEquationReal( const Float& a, const Float& b, const Float& c, Float& x1, Float& x2);` Löst eine quadratische Gleichung. Als Rückgabe werden die Anzahl Lösungen angegeben.
	`static bool findRoot(RealRealFunction pFunction, const Float& fInitialValue, Float& fRoot, Int iMaxIterations = 100000);` Findet eine Nullstelle einer beliebigen Funktion. Die Nullstelle wird iterativ gesucht.

Alle Gleichungen müssen normiert werden, das heisst, sie müssen mit Null gleichgesetzt werden.

2.5.1 Lineare Gleichungssystemen

Bei der allgemeinen Variante zum Lösen von linearen Gleichungssystemen, müssen Matrix und Vektor wie folgt abgefüllt werden:

Das Gleichungssystem

$$\left|\begin{array}{l} a_{11}x_1+a_{12}x_2+...+a_{1n}x_n+d_1=0 \\ a_{21}x_1+a_{22}x_2+...+a_{2n}x_n+d_2=0 \\ \quad ... \\ a_{m1}x_1+a_{m2}x_2+...+a_{mn}x_n+d_m=0 \end{array}\right|$$

wird in eine Matrix A und einen Vektor D gespeichert. Die Matrix A speichert alle Werte a_{11} bis a_{mn}, wobei m die Anzahl Zeilen und n die Anzahl Spalten der Matrix entsprechen. Die Spalte *i* der Matrix A enthält also die Werte der Variable x_i. Der Vektor D speichert die Werte d_1 bis d_m.

Lösen von linearen Gleichungssystemen

```
#include <zeusmath/System/EquationsSolver.h>

USING_NAMESPACE_Zeus

TMatrix mAVar1(L"[1,0,3],[2,4,6],[-8,6,2]");
TVector vDVar1(L"4,8,3");
TVector vResult1(3);
if (TEquationsSolver::solveLinearEquations(mAVar1, vDVar1, vResult1))
{
  //3 Lösungen gefunden
}
else
{
  //keine Lösungen
}
```

Tabelle 3: Zeigt ein Beispiel eines linearen Gleichungssystems mit 3 unbekannten Variablen.

2.5.2 Numerische Approximation

Die Gleichung wird so normiert, dass die eine Seite Null ist. Die andere Seite kann als reellwertige Funktion aufgefasst werden. Dann entspricht eine Nullstelle dieser Funktion einer Lösung der ursprünglichen Gleichung.

`TEquationsSolver::findRoot(..)`. Es muss eine reellwertige Funktion angegeben werden, ein Initialwert, sowie optional eine obere Grenze für die Anzahl Iterationen der zugrunde liegenden Newton'schen Approximation.

2.5.3 Klasse TLinearAlgebraHelper

Diese Hilfsklasse dient zum Ermitteln von Determinanten einer beliebigen quadratischen Matrix.

2.6 Implementation der Graphen-Theorie

Das Zeus-Framework bietet eine umfangreiche Sammlung an Algorithmen zur Graphen-Theorie. Ein Graph ist eine Verallgemeinerung von Listen und Bäumen. Folgende Liste definiert einige deutsche und englische Begriffe:

- Graph: Ein Graph besteht aus Knöten (Vertices) und Kanten (Edges).
- Directed Graph: Die Kanten eines gerichteten Graphen zeigen von einem Knoten X zum Knoten Y
- Weighted Graph: Die Kanten sind gewichtet.
- Complete Graph: Alle Knöten sind miteinander verbunden.
- Cycle: Kreis oder Zyklus. Beim Traversieren eines Graphen wird mindestens ein Knoten mehrmals besucht. Zirkulare Beziehung.
- Tree: Ein Baum ist ein gerichteten Graphen, bei dem jeder Knoten ausser die Wurzel einen Vaterknoten besitzt. Die Wurzel (Root) besitzt keinen Vatenknoten.
- Spanning Tree: Ein Baum als Teilmenge aller Kanten eines Graphen. Der minimum-cost spanning tree ist ein Baum mit den billigsten Kanten

Im Zeus-Framework existieren 3 Klassen zur Implementation eines Graphen.

API	
`TGraph`	Heap Klasse welche einen gesamten Graphen enthaltet. `zeusbase/Math/Graph.h`
`TVertice`	Definiert einen Knoten (Vertice) eines Graphen `zeusbase/Math/Vertice.h`
`TEdge`	Definiert eine Kante zwischen zwei Knöten `zeusbase/Math/Vertice.h`

Die Klasse `TGraph` beinhaltet alle Knöten und Kanten. Ein Knoten kann nicht ohne Graphen existieren. Beim Erzeugen des Graphen müssen Sie sich entscheiden ob der Graph `directed` oder `undirected` ist.

Erstellen eines Graphen

```
#include <zeusmath/System/Graph.h>

//Directed graph
TAutoPtr<TGraph> Graph1 = new TGraph(true);

//Undirected graph
TAutoPtr<TGraph> Graph2 = new TGraph(false);
```

Tabelle 4: .Erstellen von gerichteten und ungerichteten Graphen

2.6.1 Erstellen von Knoten und Kanten

Die Klasse TGraph bietet folgende Methoden um Knoten und Kanten zu erstellen, zu ermitteln oder zu löschen:

API	`zeusmath/System/Graph.h`
	`Retval addEdge(Uint uiVertice1, Uint uiVertice2, Float fWeight = 1.0);` Erzeugt eine Kante zwischen einem Knoten 1 und einem Knoten 2. Das Gewicht der Kante kann optional angegeben werden (Default ist 1).
	`Retval addVertice(Uint uiID, IZUnknown* pObject = NULL);` Erzeugt einen Knoten mit einer ID. Der Parameter `pObject` kann optional ein Datenobjekt enthalten.
	`void clearEdges();` Alle Kanten löschen.
	`void clearVertices();` Alle Knoten löschen. Löscht auch alle Kanten.
	`Int getEdgeCount() const;` Gibt die Anzahl Kanten des Graphen zurück.
	`Int getVerticeCount() const;` Gibt die Anzahl Knoten des Graphen zurück.
	`const TEdge& getEdge(Int iIndex) const;` Gibt eine indexierte Kante zurück.
	`Retval getVertice(Int iIndex, TVertice*& rpVertice) const;` Gibt einen indexierten Knoten als Klasse zurück. Die Klasse `TVertice` beinhaltet weitere knotenbezogene Methoden.
	`Retval getVerticeByID(Uint uiID, TVertice*& rpVertice) const;` Sucht einen Knoten nach seiner ID und gibt ihn zurück.

Das folgende Beispiel erstellt einen ungerichteten, gewichteten Graphen:

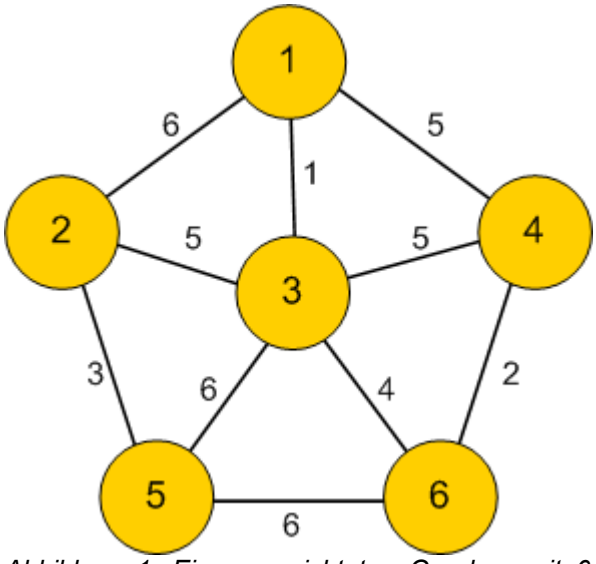


Abbildung 1: Einen gewichteten Graphen mit 6 Knoten

Der Graph wird im Programmcode wie folgt gebildet:

Erstellen eines Graphen

```
#include <zeusmath/System/Graph.h>

//Creates a undirected, weighted graph
TAutoPtr<TGraph> Graph = new TGraph(false);
Graph->addVertice(1);
Graph->addVertice(2);
Graph->addVertice(3);
Graph->addVertice(4);
Graph->addVertice(5);
Graph->addVertice(6);

Graph->addEdge(1,2, 6);
Graph->addEdge(1,3, 1);
Graph->addEdge(1,4, 5);

Graph->addEdge(2,3, 5);
Graph->addEdge(2,5, 3);

Graph->addEdge(3,4, 5);
Graph->addEdge(3,5, 6);
Graph->addEdge(3,6, 4);

Graph->addEdge(4,6, 2);
Graph->addEdge(5,6, 6);
```

Tabelle 5: .Beispiel zeigt das Erstellen eines ungerichteten, gewichteten Graphen

2.6.2 Methoden eines Knoten

Die Klasse `TVertice` sollte nicht direkt erzeugt werden. Durch den Aufruf entsprechender Methoden erzeugt die Graph-Klasse die Knoten und fügt diese in eine interne Liste. Dennoch ist die Klasse `TVertice` von Interesse. Die Knoten-Objekte können durch Methoden die Methoden `getVertice()` und `getVerticeByID()` ermittelt werden.

Sie stellt folgende Methoden zur Verfügung:

API	`zeusmath/System/Vertice.h`
	`Uint getID() const;` Gibt die ID des Knoten zurück.
	`Retval getObject(IZUnknown*& rpObject) const;` Gibt das interne Objekt des Knoten zurück.
	`void getEdgeList(IList<TEdge>& rList) const;` Gibt eine Liste aller Kanten zurück, die von diesem Knoten ausgehen.
	`void getNextVertices(IList<Uint>& rList) const;` Gibt eine Liste aller benachbarten Knoten zurück.

Objekte des Typs TVertice sind Heap-Objekte und müssen mit release() freigegeben werden

2.6.3 Erstellen eines „minimum-cost spanning tree“

Aus gewichteten Graphen kann ein Baum erstellt werden, welcher alle Knoten verbindet und dabei das billigste Netz darstellt. Dieser Baum nennt man auch den minimum-cost spanning tree eines Graphen. Im Zeus-Framework wird dazu der Algorithmus von Prim (Prioritätssuche) angewandt:

Durch den Aufruf der Methode `getMinimumCostTree()` wird ein neuer Graph erzeugt, welcher den Baum abbildet. Als Übergabeparameter kann ein Knoten angegeben werden. Dieser Knoten bildet dann die Wurzel des Baums. Wird kein Knoten angegeben, wird der erst erstellte Knoten als Wurzel angenommen.

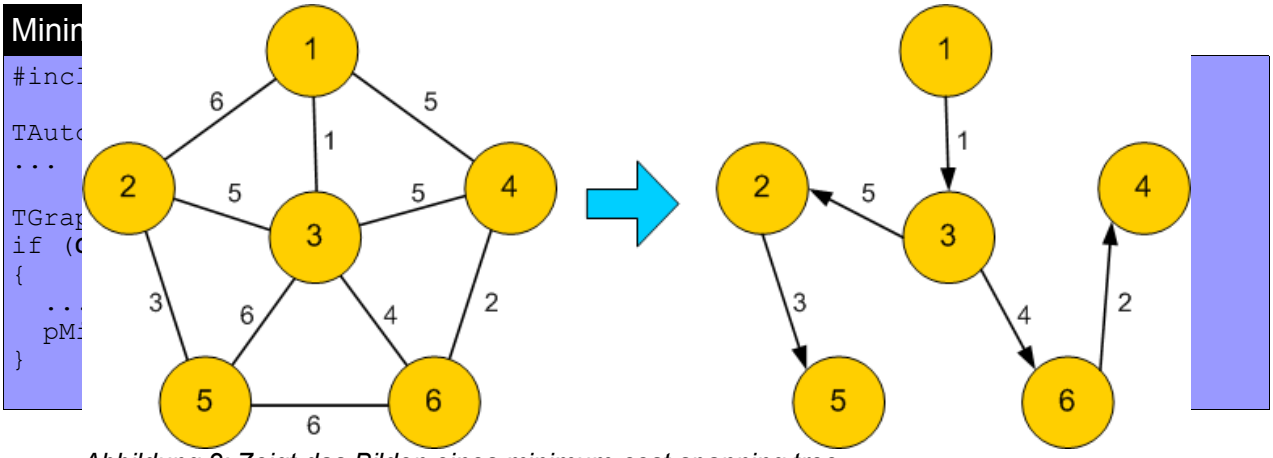


Abbildung 2: Zeigt das Bilden eines minimum-cost spanning tree

Tabelle 6: .Beispiel zeigt das Erstellen eines ungerichteten, gewichteten Graphen

2.6.4 Prüfen ob ein Graph Zyklen enthält

Gerichtete Graphen können Zyklen enthalten. Ein Zyklus ist eine Endlosschlaufe beim Traversieren eines Graphen. Bei ungerichteten Graphen existiert für jede Kante einen Zyklus, da sich die Knoten gegenseitig referenzieren. Es ist deshalb uninteressant zu prüfen, ob ein ungerichteter Graph einen Zyklus enthält.

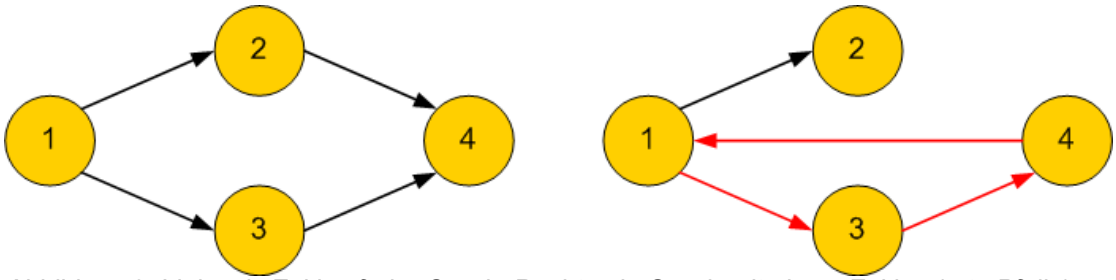


Abbildung 3: Links ein Zyklus freier Graph. Rechts ein Graph mit einem Zyklus (rote Pfeile)

Mit der Methode `hasCycle()` kann geprüft werden, ob ein Graph mindestens einen Zyklus enthält.

Auf Zyklen prüfen

```
#include <zeusmath/System/Graph.h>

TAutoPtr<TGraph> Graph = new TGraph(false);
...

if (Graph->hasCycle())
{
  ...
}
```

Tabelle 7: .Prüfen ob ein Graph einen Zyklus beinhaltet.

2.6.5 Graph Iteratoren

Zum Traversieren des Graphen existieren verschiedene Iteratoren, den Depth-First (Tiefensuche) und den Breath-First (Breitensuche) Algorithmus.

API	`zeusmath/System/Graph.h`

	`TGraphIterator* getDepthFirstIterator();`
	`const TGraphIterator* getConstDepthFirstIterator() const;`
	
	Gibt den Depth-First Iterator zurück. Muss mit `release()` freigegeben werden.

	`TGraphIterator* getBreathFirstIterator();`
	`const TGraphIterator* getConstBreathFirstIterator() const;`
	
	Gibt den Breath-First Iterator zurück. Muss mit `release()` freigegeben werden.

Der Iterator besitzt folgende Methoden

API	`zeusmath/System/GraphIterator.h`
	`void startWith(Uint uiVertice) const;` Starte den Iterator von einem bestimmten Knoten aus. Standardmässig wird beim erst eingefügten Knoten gestartet.
	`void reset();` Setzt den Iterator an den Anfang zurück.

	`Retval getNextVertice(TVertice*& rpVertice);`
	`Retval getNextVerticeConst(TVertice*& rpVertice) const;`
	
	Gibt den nächsten Knoten zurück.

2.6.5.1 Depth-First

Der Depth-First Iterator sucht zuerst in der Tiefe nach Knoten. Folgendes Bild zeigt die Funktionsweise des Algorithmus. In den Kasten steht jeweils die Traversierungs-Reihenfolge. Rechts steht die Ausgangs-Reihenfolge der Knoten.

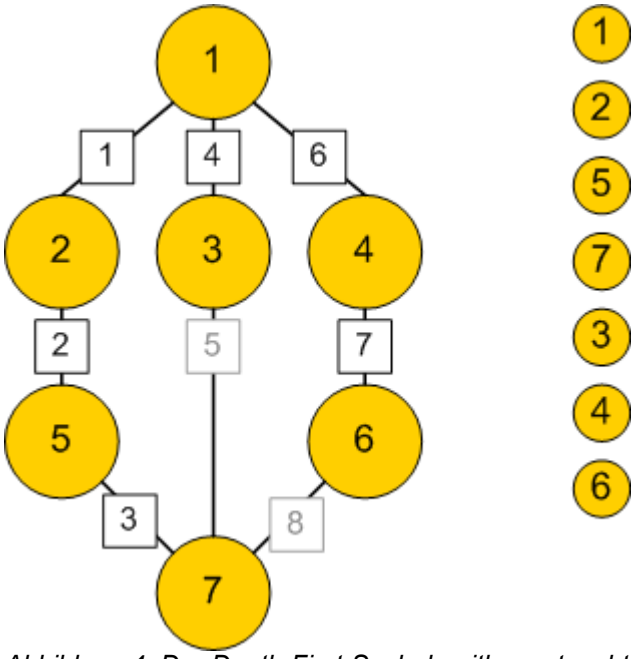


Abbildung 4: Der Depth-First Suchalgorithmus taucht zuerst in die Tiefe. Jeder Knoten wird dabei genau einmal traversiert.

Tiefensuche

```
#include <zeusmath/System/Graph.h>

TAutoPtr<TGraph> Graph = new TGraph(false);
...

TGraphIterator* pIt = Graph->getDepthFirstIterator();

TVertice* pVertice = NULL;
while(pIt->getNextVertice(pVertice) == RET_NOERROR)
{
  ...
   pVertice->release();
}
pIt->release();
```

Tabelle 8: .Zeigt die Verwendung des Depth-First Iterators

2.6.5.2 Breath-First

Der Breath-First Iterator sucht zuerst in der Breite nach Knoten. Folgendes Bild zeigt die Funktionsweise des Algorithmus. In den Kasten steht jeweils die Traversierungs-Reihenfolge. Rechts steht die Ausgangs-Reihenfolge der Knoten.

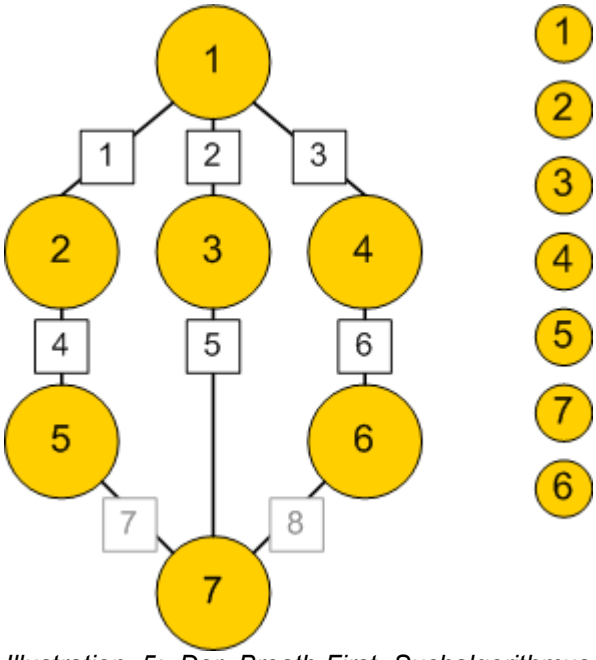


Illustration 5: Der Breath-First Suchalgorithmus sucht zuerst in die Breite ab. Jeder Knoten wird dabei genau einmal traversiert.

Breitensuche

```
#include <zeusmath/System/Graph.h>

TAutoPtr<TGraph> Graph = new TGraph(false);
...

TGraphIterator* pIt = Graph->getBreathFirstIterator();

TVertice* pVertice = NULL;
while(pIt->getNextVertice(pVertice) == RET_NOERROR)
{
  ...
   pVertice->release();
}
pIt->release();
```

Tabelle 9: .Zeigt die Verwendung des Breath-First Iterators

2.6.6 Isomorphismus

Isomorphismus ist eine bijektive Prüfung, ob zwei Graphen identisch sind. Dabei ist die Verknüpfung der Knoten relevant, nicht aber deren Bezeichnung.

Folgende Abbildung zeigt einen isomorphen Graphen:

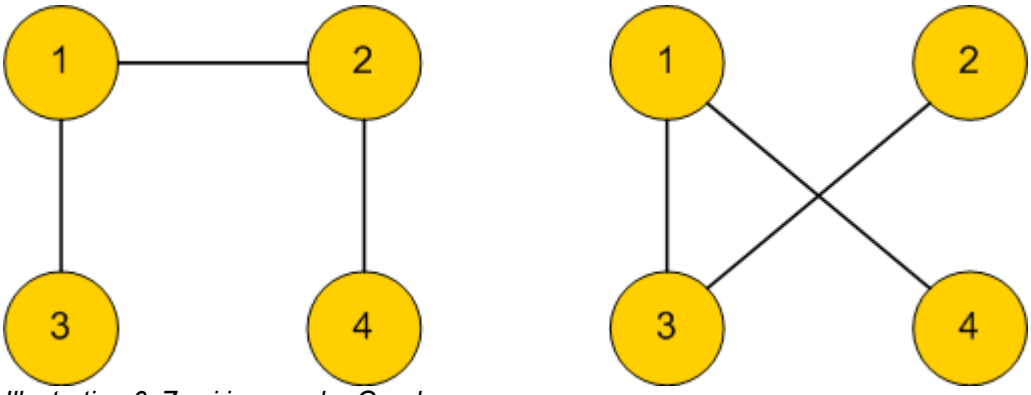


Illustration 6: Zwei isomorphe Graphen

Dabei wird die Abbildung 1→1, 2→3, 3→4 und 4→2 verwendet.

Die Zeitkomplexität um zwei Graphen miteinander zu vergleichen, ist NP (nondeterministic Polynomial). Das heisst, es existiert kein Algorithmus, welcher das Problem in polynomialer Zeit lösen kann.

API	`zeusmath/System/Graph.h`
	`bool equals(const TGraph& rGraph, bool bIsomorphism = false) const;` Prüft, ob der Graph rGraph gleich strukturiert ist wie dieses Graph-Objekt. Ist das Flag `bIsomorphism=true`, dann wird auf isomorph geprüft. Bei false wierden die Graphen exakt verglichen, dass heisst jeder Knoten muss identisch sein.
	`Retval getIsomorphicMatches(const TGraph& rGraphToMatch, TSet<TPair<Uint, Uint> >& rMatches) const;` Gibt die Menge der Knoten-Paare zurück, die zueinander isomorph sind.

Isomorphe Prüfung

```
TAutoPtr<TGraph> pGraph1 = new TGraph(false);
TAutoPtr<TGraph> pGraph2 = new TGraph(false);
...
if (ptrGraph1->equals(*ptrGraph2, true))
{
  //graphs are isomorph
}
```

Tabelle 10: Zeigt eine isomorphe Prüfung zweier Graphen.

2.6.7 Komplementärer Graph

Der komplementäre Graph erhält man durch die Umkehrung der Kanten. Da die Klasse gerichtete und ungerichtete, als auch gewichtete Graphen unterstützt, werden hier vier Variationen der Umkehrung gezeigt:

2.6.7.1 Komplement von ungerichteten Graphen

Sei G=(V,E) ein ungerichteter Graph, dann besteht eine Kante aus einer Menge aus Knöten der Kardinalität 2, also E_x={v, u}. Dann ist der komplementäre Graph wie folgt definiert:

$$G^C = (V, E^C)$$
$$E^C = \{\{u, v\} \mid u \in V, v \in V \wedge \{u, v\} \notin E\}$$

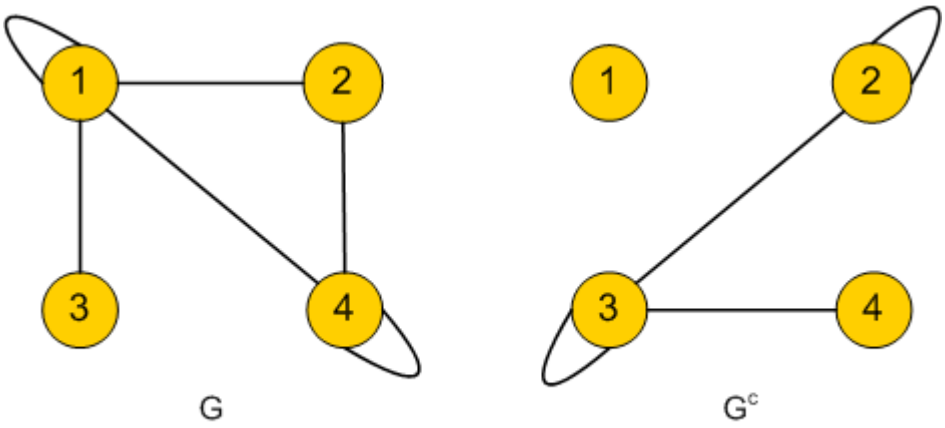


Illustration 7: Ungewichteter Graph und sein komplementärer Graph

Die gewichteten Graphen verlangen aber eine noch präzisere Angabe der Umkehrung von Kanten. Die Kanten werden als eine Struktur betrachtet: Ex=({u,v}, w), wobei w die Gewichtung der Kante ist.

$$E_1^C = \{(\{u,v\},1) \mid u \in V, v \in V \wedge (\{u,v\},1) \notin E\}$$

$$E_2^C = \{(\{u,v\},w^{-1}) \mid u \in V, v \in V \wedge (\{u,v\},w) \in E \wedge w \neq 1\}$$

$$E^C = E_1^c \cup E_2^c$$

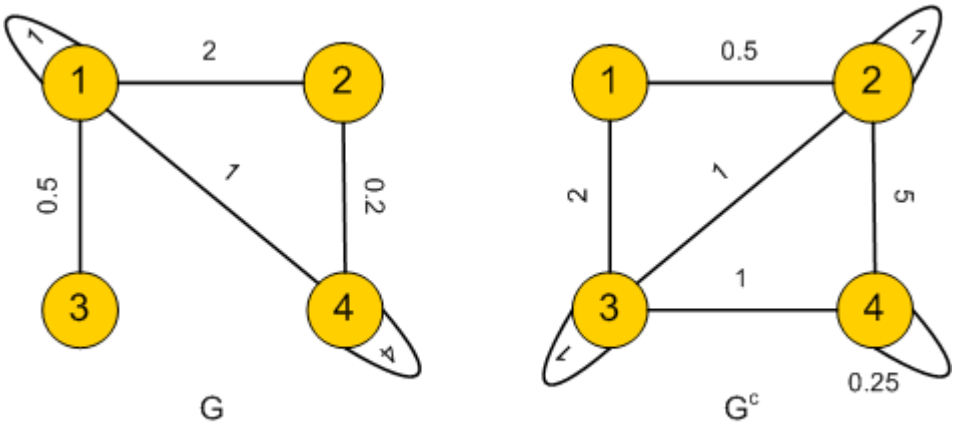


Illustration 8: Umkehrung von gewichteten Graphen

2.6.7.2 Komplement von gerichteten Graphen

Sei G=(V,E) ein gerichteter Graph, dann besteht eine Kante aus einer Struktur von 2 Knöten, E_x=(u,v).

$$G^C=(V,E^C)$$
$$E^C=\{(u,v)\mid u\in V, v\in V\wedge(u,v)\notin E\}$$

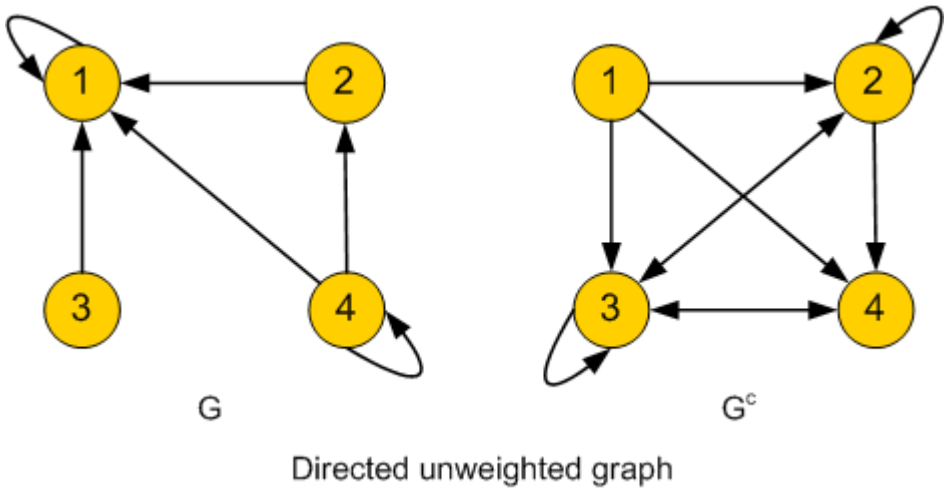


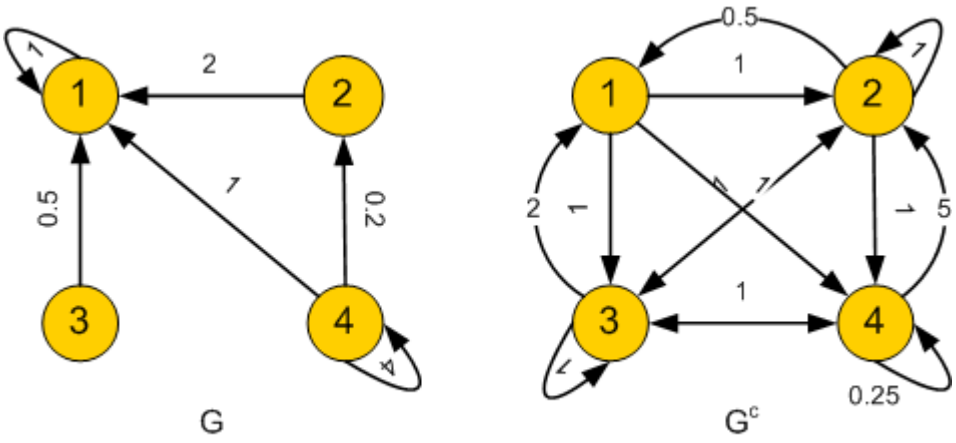
Illustration 9: Ein gerichteter Graph, dessen Kanten keine Gewichtung haben, und sein komplementärer Graph

Auch hier muss nun noch die Gewichtung der Kanten berücksichtigt werden. Dabei wird die Struktur der Kante erweitert: E_x=(u,v,w).

$$E_1^C = \{(u, v, 1) \mid u \in V, v \in V \wedge (u, v, 1) \notin E\}$$

$$E_2^C = \{(u, v, w^{-1}) \mid u \in V, v \in V \wedge (u, v, w) \in E \wedge w \neq 1\}$$

$$E^C = E_1^c \cup E_2^c$$



Directed weighted graph

Illustration 10: Umkehrung eines gerichteten und gewichteten Graphen

Komplementärer Graph

```
#include <zeusbase/Math/Graph.h>
TAutoPtr<TGraph> ptrGraph = new TGraph(false);
//adding vertices and edges
...
TAutoPtr<TGraph> ptrGraphC;
if (ptrGraph->getComplementGraph(ptrGraphC) == RET_NOERROR)
{
  ...
}
```

Tabelle 11: Ermitteln des komplementären Graphen.

3 Stochastik

Das mathematische Framework beinhaltet Klassen für Stochastik. Dabei sind vor allem verschiedene Verteilungen von Interesse:

- allgemeine Verteilungen
- Normalverteilung
- Binomial-Verteilung
- Poisson-Verteilung

Die Verteilungen sind je als Klasse implementiert und bieten eine gemeinsame Schnittstelle an, `IDistribution`.

API	
`IDistribution`	Allgemeine Schnittstelle für Verteilungen `zeusmath/Stochastic/Interfaces/IDistribution.hpp`
`TBinomialDistribution`	Definiert eine Binomial-Verteilung `zeusmath/Stochastic/BinomialDistribution.h`
`TDistribution`	Implementiert eine allgemeine Verteilung von Float-Werten `zeusmath/Stochastic/Distribution.h`
`TNormalDistribution`	Implementiert die Normalverteilung. `zeusmath/Stochastic/NormalDistibution.h`
`TPoissonDistribution`	Implementiert die Poisson-Verteilung. `zeusmath/Stochastic/PoissonDistribution.h`

Die Schnittstelle umfasst folgende Methoden:

API `zeusmath/Stochastic/Interfaces/IDistribution.hpp`
`Float MQUALIFIER getEntropy() const;` Gibt die Entropie der Verteilung an.
`Float MQUALIFIER getExpectedValue() const;` Gibt den Erwartungswert der Verteilung an.

API	`zeusmath/Stochastic/Interfaces/IDistribution.hpp`
	`Float MQUALIFIER getMean() const;` Berechnet den Durchschnittswert der Verteilung.
	`Float MQUALIFIER getMedian() const;` Der Median ist jener Wert, welcher die Fläche der Verteilung exakt halbiert.
	`Float MQUALIFIER getVariance() const;` Varianz der Verteilung.
	`Float MQUALIFIER getMode() const;` Der Modus gibt den Wert mit der höchsten Wahrscheinlichkeit an (Maximum der Verteilung).
	`Float MQUALIFIER getSkewness() const;` Gibt die schräge der Verteilung an.
	`Float MQUALIFIER getCumulativeProb(const Float& rfStart, const Float& rfEnd) const;` Gibt die Fläche zwischen zwei Werten zurück.
	`Float MQUALIFIER getStdDeviation() const;` Berechnet die Standard Abweichung
	`Float MQUALIFIER getProb(const Float& rfX) const;` Gibt die Wahrscheinlichkeit eines Wertes zurück. Diese Methode kann fehlschlagen, wenn die Verteilung auf Samples basiert. Verwenden Sie stattdessen die Methode `getSampleProb()`.
	`Float MQUALIFIER getSampleProb(Int iSample) const;` Gibt die Wahrscheinlichkeit eines Samples an.

4 Künstliche Intelligenz

4.1 Genetische Algorithmen

Das Zeus-Framework stellt eine umfassende Implementation der genetischen Algorithmen (GA) zur Verfügung. Diese ist generisch und kann auf einfache Art und Weise erweitert werden.

Folgende Klassen sind definiert:

API	
`IGAPopulation`	Schnittstelle der Population von Individuen `zeusbase/Math/Interfaces/IGAPopulation.hpp`
`IGAIndividual`	Definiert die Schnittstelle zu einem Individuum `zeusbase/Math/Interfaces/IGAIndividual.hpp`
`IGAChromosome`	Definiert ein Chromosom `zeusbase/Math/Interfaces/IGAChromosome.hpp`
`IGAGene`	Interface für Gene (Genstrukturen) `zeusbase/Math/Interfaces/IGAGene.hpp`
`TGAChromosone`	Sammlung von Genen. Implementiert das Kreuzen von Gen-Strängen. `zeusbase/Math/GAChromosone.h`
`TGAIndividual`	Ein Individuum. `zeusbase/Math/GAIndividual.h`
`TGAPopulation`	Sammlung von Individuen. Durch das Bilden von Generationen werden immer bessere Individuen gezüchtet. `zeusbase/Math/GAPopulation.h`
`TGAGene`	Implementation eines Gens in Form eines Zahlenwertes. Es können Floats oder Integers verwendet werden. `zeusbase/Math/GAGene.h`
`TGAGeneMinMax`	Implementation eines Gens mit Werte-Grenzen. `zeusbase/Math/GAGeneMinMax.h`

4.1.1 Wie funktioniert ein genetischer Algorithmus?

Hier wird nur ansatzweise die Funktionalität und Spezialitäten dieses Algorithmus erklärt. Umfassende Erklärungen finden Sie im Internet oder in Fachliteratur.

Folgender Ablauf ist typisch für genetische Algorithmen:

1. Erstellen einer Population von n-Individuen nach dem Zufallsprinzip.

2. Berechnen der Fitness von jedem Individuum.

3. Einen neue Population erstellen. Dieser Vorgang muss solange wiederholt werden, bis die Population komplett ist:

   a) Zwei Individuen aus der alten Population wählen. Diejenigen mit hoher Fitness sollten eher berücksichtigt werden.

   b) Kreuzen der zwei Individuen. Es entsteht ein neues Individuum.

   c) Je nach Mutationswahrscheinlichkeit wird die Erbinformation zusätzlich verändert.

   d) Das neue Individuum wird in die neue Population eingefügt.

4. Die neue Population wird nun verwendet. Die Schritte 2 bis 4 solange wiederholen, bis das gewünschte Resultat erreicht wurde.

4.1.2 GA Einsetzen mit Zeus-Framework

Damit ein Problem mit GA gelöst werden kann, müssen folgende Teile individuell implementiert werden:

- Das Individuum. Abgeleitet von `TGAIndiviual`

- Je nach Problemstellung muss ein eigenes Kreuzungsverfahren implementiert werden

- Je nach Problemstellung muss das Gen implementiert werden. Abgeleitet von `TGAGene` oder von `TGAGeneMinMax`.

Folgendes Beispiel soll den Einsatz des Zeus-Frameworks erläutern.

4.1.3 Beispiel Goldstein Price Function

Das Beispiel illustriert, wie die Funktion von Goldstein und Price auf einfache Art und Weise minimiert werden kann. Die Formel lautet:

$$z=(1+(x+y+1)^2(19-14x+3x^2-14y+6xy+3y^2))$$
$$(30+(2x-3y)^2(18-32x+12x^2+48y-36xy+27y^2))$$

Diese Funktion soll möglichst einfach und schnell minimiert werden, dass heisst, wir suchen jenen Punkt der das Minimum der Funktion markiert.

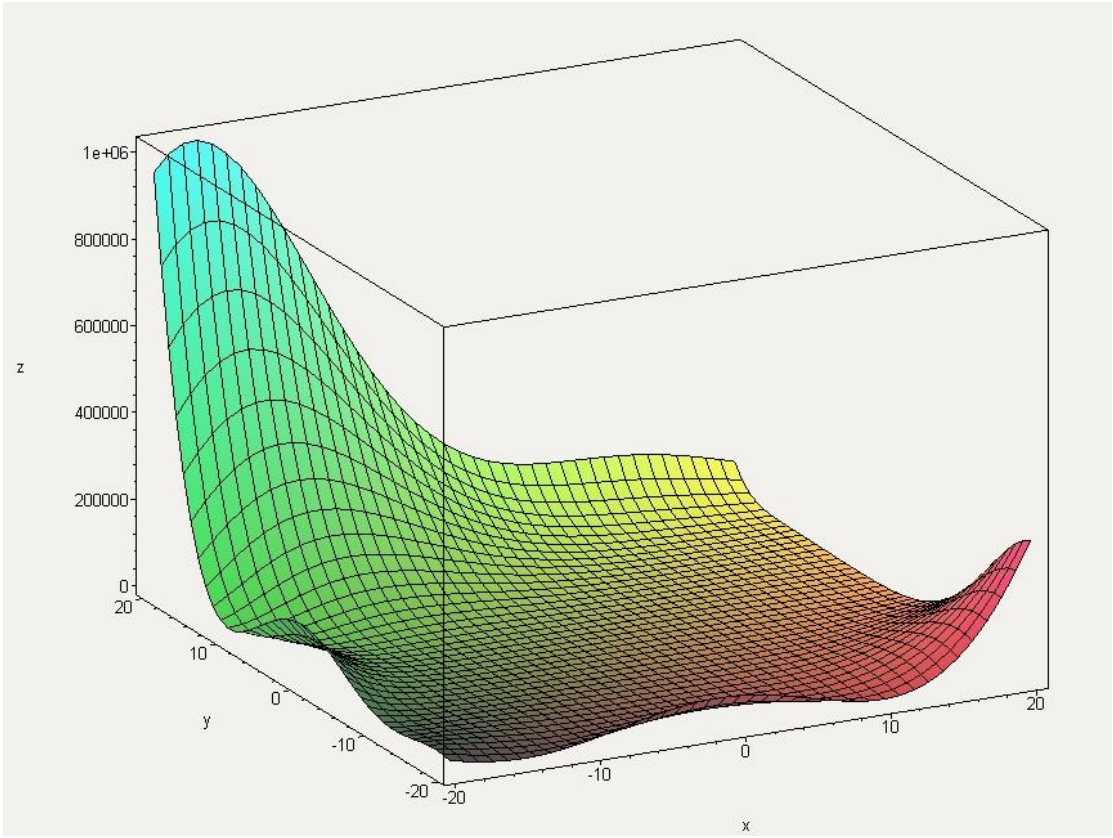


Abbildung 11: Die Goldstein und Price Funktion

4.1.3.1 Das Hauptprogramm

Um das Problem effizient zu lösen, erstellen wir 100 Individuen. 90% der Individuen-Eltern wird gekreuzt und 10% der besten Eltern wird in die neue Population übertragen (`etKeepBestParents`).

Zum Kreuzen selektieren wir zufällig Individuen aus der Population. Jene Individuen mit

der höheren Fitness bevorzugen wir (`etCrossBestPrior`).

Zudem mutieren wir 0.1 Prozent der Chromosomen, die zum Kreuzen verwendet werden.

Erstellen der Population

```
#include <GoldsteinPrice.h>
#include <zeusmath/Intel/GAPopulation.h>

{
...
  // Creates a population.
  TAutoPtr<TGAPopulation> Population =
    new TGAPopulation(
              TGoldsteinPriceIndividual::createExampleIndividuals,
              TGAPopulation::etKeepBestParents,
              TGAPopulation::etCrossBestPrior,
              TGAPopulation::etSmallerBetter,
              90);

  //Sets mutation rate
  Population->setMutationParams(0.1); // 0.1 percent

  //Generate 100 Individuals
  Population->createPopulation(100);
...
```

Tabelle 12: Erstellen einer Population zum Ermitteln des Minima der Goldstein Funktion.

Wir erstellen nun iterativ Generationen, bis wir 40mal keine Verbesserung mehr erzielen können.

Erstellen der Generationen

```
{
...
  // Creates generations. Compare tolerance is 0.01 for doubles.
  Population->makeGenerations(40, 0.01)

  //Return the best individual and print the data
  if (Population->getBestIndividual(pIndividual) == RET_NOERROR)
  {
    printf("Fitness of the best is: %f\n",
      pIndividual->getFitness());
    pIndividual->release();
  }
}
```

Tabelle 13: Erstellen der Generationen. Nach .40 Generationen, bei denen sich die Fitness des besten Individuums unverändert blieb, wird abgebrochen.

4.1.3.2 Das Individuum

Damit die Population Individuen erstellen kann, muss beim Erzeugen der Population die Fabrik-Funktion registriert werden. Diese Erzeugt in unserem Fall die GoldsteinPrice-Individuen.

Das Individuum (Header)

```
#include <zeusmath/Intel/GAIndividual.h>

//-------------------------------------------------------------------
class TGoldsteinPriceIndividual : public TGAIndividual
{
  public:
    TGoldsteinPriceIndividual();

    //Methods of IGAIndividual
    virtual double MQUALIFIER getFitness() const;
    virtual void MQUALIFIER createChromosomes();

    static unsigned long createExampleIndividuals(
                            IGAIndividual*& rpInd,
                            bool bCreateChromosomes);

};

//-------------------------------------------------------------------
//Creates individuals
/*static*/ unsigned long TGoldsteinPrice::createExampleIndividuals(
                                      IGAIndividual*& rpInd,
                                      bool bCreateChromosomes)
{
  rpInd = new TGoldsteinPriceIndividual();

  if (bCreateChromosomes)
  {
    rpInd->createChromosomes();
  }
  return RET_NOERROR;
}
```

Tabelle 14: das `TGoldsteinPriceIndividual` kapselt eine potentielle Lösung des Problems

Wir erstellen Individuen die aus einem Chromosom bestehen. Das Chromosom enthält 2 Gene, die Werte x und y. Die Individuen werden gemessen an der Fitness. Sie ist hoch, wenn ein tiefer Funktionswert der Goldstein-Funktion ermittelt wird.

Die Werte der Gene werden zufällig gewählt. Dabei gelten folgende Grenzwerte:

$$-2.0 \leqslant x \leqslant 2; -2.0 \leqslant y \leqslant 2;$$

Das Chromosom wird nach dem Prinzip der inter recombination gekreuzt.

Erstellen von Chromosomen

```
void MQUALIFIER TGoldsteinPriceIndividual::createChromosomes()
{
  TGAChromosome* pChrom =
    new TGAChromosome(0, TGAChromosome::etInterRecombination);

  double dVal1 = TTime::randomFloat(-2.0, 2.0);
  double dVal2 = TTime::randomFloat(-2.0, 2.0);

  TAutoPtr<TGAGeneMinMax> GeneX
    = new TGAGeneMinMax(dVal1, -2.0, 2.0);
  pChrom->addGene(*GeneX);

  TAutoPtr<TGAGeneMinMax> GeneY
     = new TGAGeneMinMax(dVal2, -2.0, 2.0);
  pChrom->addGene(*GeneY);

  this->m_lstChromosomes.add(pChrom);
}
```

Tabelle 15: .Erzeugen der Chromosomen und Gene.

Schlussendlich müssen wir nur noch die Fitness-Funktion implementieren. Dabei ermitteln wir den Funktionswert der Goldstein-Price-Funktion, indem wir die Gene X und Y als Parameter einfügen und rechnen.

Erstellen von Chromosomen

```
double MQUALIFIER TGoldsteinPriceIndividual::getFitness() const
{
  double dX = 0;
  double dY = 0;
  IGAChromosome* pChromosome = NULL;
  if (this->getChromosome(0, pChromosome) == RET_NOERROR)
  {
    IGAGene* pGene = NULL;
    if (pChromosome->getGene(0, pGene) == RET_NOERROR)
    {
      dX = pGene->getFloatContent();
      pGene->release();
    }

    pGene = NULL;
    if (pChromosome->getGene(1, pGene) == RET_NOERROR)
    {
      dY = pGene->getFloatContent();
      pGene->release();
    }
    pChromosome->release();
  }

  double dVal1 = 1 + pow(dX + dY + 1, 2)*
    (19 - 14*dX + 3*pow(dX,2) - 14*dY + 6*dX*dY + 3*pow(dY,2));
  double dVal2 = 30 + pow(2*dX - 3*dY, 2)*
    (18 - 32*dX + 12*pow(dX, 2) + 48*dY - 36*dX*dY + 27*pow(dY,2));

  return dVal1 * dVal2;
}
```

Tabelle 16: .ermitteln der Fitness eines Goldstein-Price-Individuums

5 Vektorgeometrie

Zeus-Math beinhaltet ein vollständiger Funktionsumfang für 3-dimensionale Vektorgeometrie, bestehend aus den Klassen `TVector3D` und `TVector2D` für Vektoren, `TStraightLine` für Geraden und `TPlane` für Ebenen. Zur Abdeckung der Funktionalität für räumliche Rotationen war eine Matrix-Klasse erforderlich, so dass auch Klassen aus der Linearen Algebra vorhanden sind. Als Anwendung der Vektorgeometrie wurde eine Klasse für Koordinatentransformationen implementiert, welche beliebig durchmischte Sequenzen aus Translationen und Rotationen verwalten kann.

5.1 3D Vektor

Der 3D Vektor ist ein optimierter Vektor. Für generelle Berechnungen im n-dimensionalen Raum ist der allgemeine TVector zu verwenden. Der 3D Vektor ist mit dem allgemeinen Vektor kompatibel und implementiert die IVector-Schnittstelle.

API	
`IVector`	Schnittstelle des allgemeinen Vektors. Siehe Kapitel zu allgemeine mathematische Klassen. `zeusmath/System/Interfaces/IVector.hpp`
`TVector`	Klasse welche einen allgemeinen Vektor implementiert. Siehe Kapitel zu allgemeine mathematische Klassen. `zeusbase/Math/Interfaces/IGAIndividual.hpp`
`IVector3D`	Schnittstelle für geometrischen Vektor im 3D Raum. `zeusmath/Geometry/Interfaces/IVector3D.hpp`
`TVector3D`	Implementation des 3D Vektors. `zeusmath/Geometry/Vector3D.h`

In diesem Kapitel sehen wir uns nur die Methoden des geometrischen Vektors an. Die allgemeinen Vektor-Methoden sind im Kapitel zu Allgemeine mathematische Klassen entnehmen.

Der 3D Vektor besitzt die 3 Komponenten X, Y und Z. Folgende Methoden sind optimiert für den Zugriff und Berechnung:

API	zeusmath/Geometry/Vector3D.h
	`TVector3D getNormalizedVector() const;` Gibt den Einheitsvektor zurück..

	`TVector3D rotatedX(const IAngle& rAngle) const;`
	`TVector3D rotatedY(const IAngle& rAngle) const;`
	`TVector3D rotatedZ(const IAngle& rAngle) const;`
	
	Rotiert den Vektor um die X, Y oder Z Achse und gibt den neuen Vektor zurück..

	`bool MQUALIFIER isParallel(const IVector3D& Vector) const;` Prüft, ob zwei Vektoren parallel sind und in die gleiche Richtung zeigen.
	`bool MQUALIFIER isCollinear(const IVector3D& Vector) const;` Prüft, ob zwei Vektoren parallel sind. Sie können auch in die entgegengesetzte Richtung zeigen.

	`bool MQUALIFIER isUnitVectorX() const;`
	`bool MQUALIFIER isUnitVectorY() const;`
	`bool MQUALIFIER isUnitVectorZ() const;`
	
	Prüft, ob der Vektor ein Einheitsvektor in eine der Achsen X, Y oder Z ist.

	`const Float& MQUALIFIER getX() const;`
	`const Float& MQUALIFIER getY() const;`
	`const Float& MQUALIFIER getZ() const;`
	
	Gibt die X, Y oder Z Komponente des Vektors zurück. Pro Komponente gibt es entsprechende Set-Methoden.

	`void MQUALIFIER calcAngleTo(const IVector3D& rVector, IAngle& rAngle) const;` Berechnet den Winkel zwischen zwei Vektoren.
	`Float MQUALIFIER calcDistanceTo(const IVector3D& rVector) const;` Berechnet die Distanz zwischen 2 Vektoren.

5.1.1 Initialisierungen

Ein Vektor kann als Nullvektor, sowie als Einheitsvektor entlang der X, Y oder Z-Achse initialisiert werden:

Initialisieren

```
#include <zeusmath/Geometry/Vector3D.h>

TVector3D v1;
v1.initAsZeroVector();

TVector v2;
v2.initAsUnitVectorX();

TVector v3;
v3.initAsUnitVectorY();

TVector v4;
v4.initAsUnitVectorZ();
```

Tabelle 17: Initialisieren eines Vektors im 3D Raum

5.1.2 Operationen

Dank Operator-Überladungen kann direkt mit Instanzen von `TVector3D` gerechnet werden:

Operatoren

```
TVector vecA(23, 4, -2);
TVector vecB(-20, -4, 2);
TVector vecSum = vecA + vecB; // vecSum ist jetzt (3, 0, 0).
```

Tabelle 18: Verwenden von Operatoren eines Vektors

Vektoraddition und Subtraktion:
`vec1 + vec2, vec1 - vec2, vec1 += vec2, vec1 -= vec2, IVector::addToThis(..), IVector::subtractFromThis(..)`

Skalarprodukt: `vec1 * vec2, IVector::calcScalarProduct(..)`

Vektorprodukt: `vec1 ^ vec2, IVector::calcVectorProduct(..)`

Multiplikation und Division mit Skalar:
`f1 * vec1, vec1 * f1, IVector::multiply(..), IVector::divide(..)`

5.1.3 Vergleichsmethoden

Gleichheit zweier Vektoren:
`vec1 == vec2, vec1 != vec2, IVector::isEqual(..)`

Parallelität / Gleichrichtung zweier Vektoren:
`vec1 || vec2, IVector::isParallel(..)`

Kollinearität zweier Vektoren: `IVector::isCollinear(..)`

5.1.4 Berechnungsmethoden

Länge / Norm eines Vektors:
`IVector: getNorm(), getLength(), getNormSquared()` resp. `getLengthSquared()` für das Quadrat der Länge

Normalisierung:
`IVector::normalize(), TVector::getNormalizedVector()`

Winkel zwischen zwei Vektoren: `IVector::calcAngleTo(..)`

Distanz zwischen zwei Punkten: `IVector::calcDistanceTo(..)`

Horizontalwinkel lesen und setzen:
`IVector: getHorizontalAngle(..),setHorizontalAngle(..)`

Vertikalwinkel lesen und setzen:
`IVector: getVerticalAngle(..),setVerticalAngle(..)`

5.1.5 Rotationen

Ein `IVector` kann mit direkten Methoden um jede Achse rotiert werden:

x-Achse: `IVector::rotateX(..), TVector::rotatedX(..)`

y-Achse: `IVector::rotateY(..), TVector::rotatedY(..)`

z-Achse: `IVector::rotateZ(..), TVector::rotatedZ(..)`

Horizontalwinkel:
`IVector::rotateHorizontalAngle(..)`,
`TVector::rotatedHorizontalAngle(..)`

Vertikalwinkel:
`IVector::rotateVerticalAngle(..)`,
`TVector::rotatedVerticalAngle(..)`

5.1.6 Spezielle statische Members von TVector

Nullvektor: `TVector::getZeroVector()`

Einheitsvektor x: `TVector::getUnitVectorX()`

Einheitsvektor y: `TVector::getUnitVectorY()`

Einheitsvektor z: `TVector::getUnitVectorZ()`

5.2 Schnittebene und Schnittgeraden

5.2.1 Klasse TStraightLine & Interface IStraightLine

5.2.1.1 Allgemeine Angaben

Include: `<zeusmath/Geometry/StraightLine.h>`

Art der Klasse: Stack-Objekte mit Interface, nicht von `TZObject` abgeleitet

Verwendung: 3-dimensionale Gerade, aufbauend auf `TVector`

5.2.1.2 Konstruktoren

Defaultkonstruktor: Erzeugt eine Gerade, die mit der x-Achse zusammenfällt.

Konstruktor mit 2 Vektoren und einem booleschen Wert: Erzeugt eine Gerade entweder auf Grund der Punkt-Richtungsform im Falle `bConstructionByPointDirectionForm_Or_By2Points = true`, oder aber durch 2 Punkte im Falle das Flag ist `false`.

Kopierkonstruktor: Zur Konstruktion aus einem Objekt der Implementationsklasse

Konstruktor mit `const IStraightLine&`: Zur Konstruktion aus einem Objekt des Interfacetyps

5.2.1.3 Eigenschaften

Initialvektor / Anfangspunkt:
Lesen mit `IStraightLine::getInitialVector()`, Setzen mit `IStraightLine::setInitialVector(..)`

Richtungsvektor: Lesen mit `IStraightLine::getDirectionVector()`, Setzen mit `IStraightLine::setDirectionVector(..)`

Gültigkeit: `IStraightLine::isValid()`

5.2.1.4 Initialisierungen

Als x-Achse: `IStraightLine::initAsAxisX()`, Test mit `isAxisX()`

Als y-Achse: `IStraightLine::initAsAxisY()`, Test mit `isAxisY()`

Als z-Achse: `IStraightLine::initAsAxisZ()`, Test mit `isAxisZ()`

5.2.1.5 Vergleichsmethoden

Gleichheit: `sl1 == sl2`, `sl1 != sl2`, `IStraightLine::isEqual(..)`

Parallelität: `sl1 || sl2`, `IStraightLine::isParallel(..)`

5.2.1.6 Berechnungsmethoden

Punkt & Laufparameter:
[Hinweis zur Punkt-Richtungsform: Bei Laufparameter 0 befindet sich immer der Anfangspunkt (`getInitialVector()`), bei Laufparameter 1 immer der Punkt `getInitialVector() + getDirectionVector()`.]

```
IStraightLine::calcPointFromParameter(..)
IStraightLine::calcParameterFromPoint(..)
```

Hinweis: Ein beliebiger räumlicher Punkt liegt in der Regel nicht auf einer gegebenen Geraden.

Nächstgelegener Punkt zu einer Geraden:
`IStraightLine::calcClosestPoint(..)`

Distanz zu einem gegebenen Punkt:
`IStraightLine::calcDistanceTo(..)`. Falls bLimited true ist, wird die Gerade lediglich als Strecke zwischen Laufparameter 0 und 1 aufgefasst, und nicht als unbegrenzte Gerade. Die findet Anwendung auf dem Gebiet der Computergrafik: Damit kann beispielsweise die Distanz zwischen dem Punkt eines Mausklicks und einem Streckenelement berechnet werden.

Enthaltensein eines Punktes:
`IStraightLine::containsPoint(..)`. Wird innerhalb von `calcParameterFromPoint()` verwendet.

Schnittpunkt zweier Geraden, lediglich in 2D:
`IStraightLine::calcIntersectionPoint2D(..)`. Es werden lediglich die Komponenten x und y berücksichtigt.

Schnittpunkt zweier Geraden, in 3D:
`IStraightLine::calcIntersectionPoint3D(..)`. Hinweis: Beliebige Geraden schneiden sich räumlich in der Regel nicht.

Kürzeste Verbindung zweier Geraden:
`IStraightLine::calcShortestConnection(..)`

5.2.1.7 Spezielle statische Members von TStraightLine

x-Achse: `TStraightLine::getAxisX()`

y-Achse: `TStraightLine::getAxisY()`

z-Achse: `TStraightLine::getAxisZ()`

5.2.2 Klasse TPlane & Interface IPlane

5.2.2.1 Allgemeine Angaben

Include: `<zeusmath/Geometry/Plane.h>`

Art der Klasse: Stack-Objekte mit Interface, nicht von `TZObject` abgeleitet

Verwendung: 3-dimensionale Ebene, aufbauend auf `TVector`

5.2.2.2 Konstruktoren

Defaultkonstruktor: Erzeugt eine Ebene, die mit der xy-Ebene zusammenfällt.

Konstruktor mit 3 Vektoren und einem booleschen Wert: Erzeugt eine Ebene entweder auf Grund der Punkt-Richtungsform im Falle `bConstructionByPointDirectionForm_Or_By3Points = true`, oder aber durch 3 Punkte im Falle das Flag ist `false`.

Konstruktor mit 4 Float-Werten: Definiert eine Ebene durch die Koordinatenform `ax+by+cz+d=0`. Übergeben werden die Parameter a, b, c und d.

Konstruktor mit 2 Vektoren: Definiert die Ebene durch einen Normalenvektor (steht auf der Ebene senkrecht) und einen gegebenen Punkt.

Kopierkonstruktor: Zur Konstruktion aus einem Objekt der Implementationsklasse

Konstruktor mit const IPlane&: Zur Konstruktion aus einem Objekt des Interfacetyps

5.2.2.3 Eigenschaften

Initialvektor / Anfangspunkt: Lesen mit `IPlane::getInitialVector()`, Setzen mit `IPlane::setInitialVector(..)`

Richtungsvektoren 1 und 2: Lesen mit IPlane::getDirectionVector1/2(), Setzen mit `IPlane::setDirectionVector1/2(..)`

Koeffizienten der Koordinatenform:
Lesen mit `IPlane::getCoordFormA/B/C/D()`, Setzen mit

`IPlane::setCoordFormA/B/C/D(..)`

Normalenvektor:
`IPlane::calcNormalVector(..), TPlane::getNormalVector()`

Gültigkeit: `IPlane::isValid()`

5.2.2.4 Initialisierungen

xy-Ebene: `IPlane::initAsPlaneXY()`, Test mit `isPlaneXY()`

xz-Ebene: `IPlane::initAsPlaneXZ()`, Test mit `isPlaneXZ()`

yz-Ebene: `IPlane::initAsPlaneYZ()`, Test mit `isPlaneYZ()`

5.2.2.5 Vergleichsmethoden

Gleichheit:
`Plane1 == Plane2`, `Plane1 != Plane2`, `IPlane::isEqual(..)`

Parallelität: `Plane1 || Plane2, IPlane::isParallel(..)`

5.2.2.6 Berechnungsmethoden

Punkt & Laufparameter:
`IPlane::calcPointFromParameters(..)`
und `IPlane::calcParametersFromPoint(..)`.

Hinweis: Ein beliebiger räumlicher Punkt liegt in der Regel nicht in einer gegebenen Ebene.

Liegen 2 gegebene Punkte auf derselben Seite bezüglich einer Ebene?
`IPlane:: arePointsOnSameSide(..)`

Distanz zu einem gegebenen Punkt: `IPlane::calcDistanceTo(..)`

Durchstosspunkt einer Geraden mit einer Ebene:

`IPlane::calcIntersectionPointWithStraightLine(..)`

Schnittpunkt von 3 Ebenen:
`IPlane::calcIntersectionPointWith2Planes(..)`

Schnittgerade von 2 Ebenen:
`IPlane:: calcIntersectionLine(..)`

Winkel zwischen 2 Ebenen:
`IPlane:: calcAngleTo(..)`

Enthaltensein eines Punktes:
`IPlane::containsPoint(..)`.
Wird innerhalb von `calcParametersFromPoint()` verwendet.

Enthaltensein einer Geraden:
`IPlane::containsStraightLine(..)`

5.2.2.7 Spezielle statische Members von TPlane

xy-Ebene: `TPlane::getPlaneXY()`

xz-Ebene: `TPlane::getPlaneXZ()`

yz-Ebene: `TPlane::getPlaneYZ()`

5.3 Transformationen

5.3.1 Klasse TCoordinatesTransformator & Interface ICoordinatesTransformator

5.3.1.1 Allgemeine Angaben

Include: `<zeusmath/Geometry/CoordinatesTransformator.h>`

Art der Klasse: Heap-Objekte mit Interface, abgeleitet von `TZObject`

Verwendung: Ein `TCoordinatesTransformator`-Objekt kapselt eine Koordinatentransformation zwischen 2 Koordinatensystemen S1 und S2, welche aus beliebigen Sequenzen von Rotationen und Translationen (Verschiebungen) besteht.

Bei der Initialisierung wird das System S2 ausgehend vom System S1 definiert. Damit können Koordinaten bezüglich des Systems S1 in das System S2 transformiert werden (normale Transformation) sowie von S2 nach S1 (umgekehrte Transformation).

5.3.1.2 Konstruktoren

Defaultkonstruktor: Erzeugt einen Identitäts-Transformator, d.h. die Systeme S1 und S2 sind gleich. Die normale und die umgekehrte Transformation werden als Ausgabevektor den Eingabevektor zurückgeben.

5.3.1.3 Initialisierungen

Identitäts-Transformation:
`ICoordinatesTransformator: initAsIdentity()`,
Test mit `isIdentity()`

Hinzufügen einer Rotation um die x-Achse:
`ICoordinatesTransformator::addRotationX(..)`

Hinzufügen einer Rotation um die y-Achse:

`ICoordinatesTransformator::addRotationY(..)`

Hinzufügen einer Rotation um die z-Achse:
`ICoordinatesTransformator::addRotationZ(..)`

Hinzufügen einer Translation/ Verschiebung:
`ICoordinatesTransformator::addTranslation(..)`

Setzen des Streckfaktors:
`ICoordinatesTransformator::setStretchFactor(..)`

5.3.1.4 Transformationsmethoden

Transformation eines Punktes des Systems S1 in das System S2:
`ICoordinatesTransformator::transform(..)`

Umkehrtransformation eines Punktes des Systems S2 in das System S1:
`ICoordinatesTransformator::transformReverse(..)`

5.3.1.5 Eigenschaften

Abfrage der hinterlegten, zusammengesetzten Rotationsmatrix:
`ICoordinatesTransformator:: getRotationMatrix()`

Abfrage der hinterlegten, zusammengesetzten inversen Rotationsmatrix:
`ICoordinatesTransformator:: getReverseRotationMatrix()`

Abfrage des hinterlegten, zusammengesetzten Translationsvektors:
`ICoordinatesTransformator:: getTranslationVector()`

Abfrage des Streckfaktors:
`ICoordinatesTransformator:: getStretchFactor()`

6 Analysis / Calculus

Dieses Paket befasst sich mit Berechnungen und Methoden aus dem Gebiet der Analysis.

6.1 Ableiten und Integration

Schliesslich werden im Rahmen numerischer Approximationen Operationen der Infinitesimalrechnung abgedeckt, das heisst Ableitungen n-ten Grades sowie numerische, bestimmte Integration.

6.1.1 Klasse TCalculusHelper

6.1.1.1 Allgemeine Angaben

Include: `<zeusmath/Calculus/CalculusHelper.h>`

Art der Klasse: Statische Klasse, Objektinstanzierung ist nicht möglich, kein Interface

Verwendung: Numerische Approximation von Ableitungen beliebigen Grades sowie numerische, bestimmte Integration reellwertiger Funktionen. Die numerische Ableitung 1. Grades wird in der Methode `TEquationsSolver::findRoot(..)` verwendet, damit Nullstellen beliebiger reellwertiger Funktionen gefunden werden können.

6.1.1.2 Methoden

Ableitungen n-ten Grades:
`TCalculusHelper::calcDerivative(..)`. Es muss ein Funktionszeiger auf eine reellwertige Funktion angegeben werden, sowie der gewünschte x-Wert und der gewünschte Grad n≥0 der Ableitung.

Numerische, bestimmte Integration:
`TCalculusHelper::integrate(..)`. Es muss ein Funktionszeiger auf eine reellwertige Funktion angegeben werden, das Integrationsintervall x1..x2, und optional,

ob absolut integriert werden soll (Default: false), optional die Anzahl Intervalle.

Auswertung einer reellwertigen Funktion:
`TCalculusHelper::evalFunction(..)`. Es muss ein Funktionszeiger auf eine reellwertige Funktion angegeben werden, sowie der x-Wert. Obwohl es sich um eine intern verwendete Hilfsmethode handelt, ist diese Methode öffentlich, da sie dem Aufrufer die Syntax von Aufrufen mit Funktionszeigern abnimmt.

7 Logik

7.1 Fuzzy Logik

Das Zeus-Framework bietet einige Klassen für Fuzzy Logik an:

API	
`TFuzzySet`	Das Fuzzy Set beinhaltet eine Menge zum Rechnen mit Fuzzy Logik. `zeusbase/Math/FuzzySet.h`
`TFuzzyLogic`	Fuzzy Logik Klasse. `zeusbase/Math/FuzzyLogic.h`

A Versionsgeschichte

Änderungen am Dokument sind hier festgehalten

Version	*Beschreibung*	*Datum*
0.1	Erstellen der ZeusMath Bibliothek	12.03.07
0.2	Anpassen an Zeus-Framework 0.5.0. Fuzzy Logik aus ZeusBase in ZeusMath verschoben.	29.06.07
0.6	Matrix und Vektor Klassen erweitert Komplexe Zahlen implementiert. Graphen, Neuronale Netze und genetische Algorithmen aus ZeusBase Bibliothek	03.01.08
0.6.3	Restrukturierung und Erweiterung durch Klassen aus der Stochastik	

B Literaturverzeichnis

[1]: Benjamin Hadorn, Zeus Basic Library, http://www.xatlantis.ch/doc/ZeusBase.pdf

C Index

V

W