ChatGPT

Architekturanalyse des AngouriMath Core-Moduls

Einleitung / Zielsetzung

AngouriMath ist eine plattformübergreifende .NET-Bibliothek für symbolische Algebra, die es ermöglicht, mathematische Ausdrücke zu analysieren und zu manipulieren. Mit AngouriMath können Benutzer Gleichungen (auch Gleichungssysteme) automatisch lösen, Ableitungen bilden, Ausdrücke parsen, Funktionen kompilieren, mit Matrizen arbeiten, Grenzen bestimmen, Ausdrücke in LaTeX umwandeln und vieles mehr [1]. Das **Core-Modul** bildet den symbolischen Kern dieser Bibliothek. In dieser Architekturanalyse wird die Struktur des Core-Moduls detailliert untersucht, um ein klares Bild der **Architektur** zu vermitteln – welche Komponenten vorhanden sind, wie sie zusammenwirken und wie der symbolische Kern aufgebaut ist.

Die Analyse umfasst:

- Die **Projektstruktur** und Verzeichnis-Hierarchie des Core-Moduls
- Die wichtigsten **Namespaces, Klassen und Hierarchien** – mit Fokus auf zentrale Typen wie `Entity`, `MathS`, Operationen, Werte etc.
- Die öffentlichen **API-Einstiegspunkte**, über die Anwender mit der Bibliothek interagieren
- Die internen **Abhängigkeiten** zwischen Komponenten innerhalb des Core-Moduls
- Eine erste **Einschätzung** zur Modularität, Erweiterbarkeit und möglichen Engpässen des Designs

Ziel ist es, einen verständlichen Überblick über den Aufbau des Core-Moduls von AngouriMath zu geben, insbesondere darüber, **welche Komponenten** es gibt, **wie** sie interagieren und **wie** der symbolische Kern strukturiert ist.

Projektstruktur

Das Core-Modul ist im Repository als eigenes Projekt unter `Sources/AngouriMath` organisiert. Die Quelltexte sind in thematischen Unterordnern gegliedert, hauptsächlich in **Core** und **Functions**:

- **Core/** – Enthält die grundlegenden Definitionen des symbolischen Kerns. Hier liegen u.a.:
- `Entity` und die dazugehörigen Unterklassen (für Knoten des Ausdrucksbaums) in `Core/Entity/...`
- Weitere zentrale Klassen, z.B. `EquationSystem` (für Gleichungssysteme) in `Core/EquationSystem.cs` [2], sowie Hilfstypen wie `Domain` (zur Definition von Zahlenbereichen) und `ApproachFrom` (für Grenzwertberechnung) im Core-Namespace.
- Schnittstellen und Exceptions des Kernmoduls (z.B. `ILatexiseable` für LaTeX-Ausgabe, Fehlerklassen für ungültige Operationen etc.).

- **Functions/** – Beinhaltet die Implementierung der symbolischen Algorithmen und Funktionen, getrennt vom reinen Datenmodell. Beispiele für Unterbereiche:

- `Functions/Compilation/...` – Compiler für Ausdrücke in ausführbare Formate (AngouriMath's „Fast Expression" VM) [3].
- `Functions/TreeAnalyzer.cs` und verwandte Klassen – Analysen und Transformationen des Ausdrucksbaums (z.B. Sortierung, Komplexitätsbewertung) [4] [5].
- `Functions/Patterns.cs` und `Functions/Simplificator.cs` – Mustererkennung und Vereinfachungsroutinen für algebraische Ausdrücke [5].
- `Functions/TrigonometricAngleExpansion.cs` & `TrigonometryTableValues.cs` – Behandlung spezieller trigonometrischer Identitäten und Werte [6].
- `Functions/ExpressionNumerical.cs` – Numerische Auswertung von Ausdrücken (Evaluation).
- `Functions/Fraction.cs` und `BaseConversion.cs` – Routinen zur Bruchrechnung und Basisumrechnung.

Daneben existieren im Repository weitere **Module außerhalb des Core**, die auf diesem aufbauen (wie `AngouriMath.FSharp` für F#-Integration, `AngouriMath.Interactive` für Jupyter/VSCode-Notebooks, `AngouriMath.Terminal` als eigenständige CLI [7]). Diese sind jedoch getrennte Projekte und nicht Teil des hier betrachteten Core-Moduls. Das Core-Modul selbst ist weitgehend eigenständig und bildet die Grundlage, auf der diese anderen Module aufsetzen.

Build & Distribution: Das Core-Modul wird als eigenständige .NET-Bibliothek kompiliert (NuGet-Paket „AngouriMath"). Im Repository wird es gemeinsam mit den erwähnten Zusatzprojekten gebaut und getestet (für C#, F#, Interactive, C++ etc., siehe Build-Status im README [8]). Der Core-Code ist in C# geschrieben und zielt auf .NET Standard (für breite Kompatibilität).

Die Projektstruktur spiegelt eine Trennung von **Datenstrukturen** (im Core-Verzeichnis) und **Algorithmen/Funktionen** (im Functions-Verzeichnis) wider. Dadurch bleibt der Ausdrucksbaum klar definiert, während komplexe Operationen ausgelagert und modular gehalten sind.

Zentrale Klassen & Hierarchien

Im Mittelpunkt des Core-Moduls stehen einige fundamentale Klassen und ihre Hierarchien:

- `Entity` – *Zentrale abstrakte Basisklasse* für alle symbolischen Ausdrücke [9]. Jede Zahl, Variable, jeder Operator oder Funktionsterm im Ausdrucksbaum ist ein `Entity`. Man kann `Entity` selbst nicht direkt instanziieren; stattdessen existieren *verschachtelte Klassen* (Nested Classes) innerhalb von `Entity` für die konkreten Untertypen [10]. Diese Hierarchie bildet den Knoten-Typenbaum des symbolischen Ausdrucks. Wichtigste Unterklassen bzw. -typen von `Entity` sind:
- `Entity.Number` – Abstrakte Basisklasse für Zahlenwerte. Sie umfasst u.a.:
  - `Entity.Number.Integer` – Ganze Zahlen (mit beliebiger Größe, intern als Big Integer gespeichert).
  - `Entity.Number.Rational` – Rationale Zahlen (Brüche, intern mit beliebiger Präzision, basierend auf `ERational` aus der PeterO.Numbers Library).
  - `Entity.Number.Real` – Reelle Zahlen (mit beliebiger Präzision als Dezimalzahl, basierend auf `EDecimal`).
  - `Entity.Number.Complex` – Komplexe Zahlen (mit Real- und Imaginärteil als `Real` bzw. `EDecimal`; alternativ wird teils `System.Numerics.Complex` genutzt) [11] [12]. Diese numerichen Entitäten kapseln also hochpräzise Zahlen-Arithmetik durch Verwendung externer Big Number-Bibliotheken. Die Klasse `MathS.Numbers` stellt Fabrikmethoden bereit, um solche Zahlentypen zu erzeugen (mit automatischem

Downcasting auf einfachere Typen, z.B. ergibt `Number.Create(1.0)` einen `Integer` statt `Real`, sofern aktiviert) [13].

- `Entity.Variable` – Symbolische Variable. Repräsentiert eine Unbekannte oder symbolischen Parameter. Besitzt einen Namen/Identifier. In Ausdrücken fungieren Variablen als Platzhalter, die bei Bedarf ersetzt (`Substitute`) oder zur Differentiation/Integrierung verwendet werden.
- `Entity.Set` – Abstrakte Basisklasse für Mengen-Ausdrücke. Darunter fallen:
  - `Entity.Set.FiniteSet` – Endliche Menge explizit angegebener Elemente.
  - `Entity.Set.Interval` – Intervalle (z.B. $(a; b]$ für halboffen).
  - `Entity.Set.ConditionalSet` – Mengen definiert durch eine Bedingung (z.B. $\{x \mid Eigenschaft(x)\}$).
  - `Entity.Set.SpecialSet` – Vordefinierte spezielle Mengen wie $\mathbb{R}$ (Reelle Zahlen), $\mathbb{Z}$ etc., die als Konstanten verwendet werden.
    Ergebnisse von Lösungs- und Lösungsmengenberechnungen werden typischerweise als `Entity.Set`-Untertypen zurückgegeben (z.B. liefert eine Gleichungslösung eine `FiniteSet` von Lösungen, Ungleichungen ergeben Intervalle usw.) [14].
- **Operator- und Funktions-Klassen** – Für jede unterstützte mathematische Operation oder Funktion gibt es eine eigene abgeleitete `Entity`-Klasse. Diese sind im Code nach Kategorien gruppiert:
  - *Arithmetische Operatoren* (im **Continuous**-Bereich): z.B. `Sumf` (Addition, entspricht `+`), `Minusf` (Subtraktion `-`), `Mulf` (Multiplikation `*`), `Divf` (Division `/`), Potenz (`Pow` für `^`), etc. [15]. Diese Klassen sind in der Regel als *Versatzstücke (Nodes)* implementiert, die ihre Operanden als Kindknoten enthalten.
  - *Transzendente Funktionen* (Continuous): z.B. `Sinf` (Sinus), `Cosf` (Cosinus), `Tanf` (Tangens), `Ln` (natürlicher Logarithmus), `Exp` (Exponentialfunktion) usw. [16]. Auch Hyperbelfunktionen (Sinh, Cosh, ...), Wurzelfunktionen (`Sqrt`) und andere kontinuierliche Funktionen zählen hierzu. Intern sind sie meist als **versiegelte** (sealed) Klassen umgesetzt, die keine weiteren Unterklassen haben (jede steht für eine konkrete mathematische Funktion) [17].
  - *Logische Operatoren* (Discrete-Bereich): z.B. `Andf` (logisches UND), `Orf` (ODER), `Notf` (Negation), `Impliesf` (Implikation). Diese operieren auf booleschen Ausdrücken (die in AngouriMath ebenfalls als `Entity` behandelt werden).
  - *Vergleichsoperatoren*: z.B. `Lessf` / `Greaterf` (Kleiner-/Größer-als), `Equality` (Gleichheitsrelation „="). Ein **Gleichheitsausdruck** $f(x) = g(x)$ wird im Ausdrucksbaum als eigener Knoten (Equality-Entity) dargestellt, der bei der Lösung in eine Gleichung umgewandelt wird. (Für die Lösung von Gleichungen wird erwartet, dass man stattdessen $f(x) - g(x)$ bildet oder `SolveEquation` verwendet, da ein `Equality`-Knoten direkt nicht vom `EquationSystem` verarbeitet wird [18] [19].)
  - *Mengenoperationen*: z.B. `Unionf` (Vereinigung $\cup$), `Intersectionf` (Durchschnitt $\cap$), `In` (Element der Menge, Ausdruck wie `x in A` ergibt einen logischen Statement-Knoten) [20] [21].
  - *Spezialfunktionen*: Zusätzlich gibt es Klassen für Summenzeichen/Produkte (falls unterstützt), und andere seltener genutzte Konstrukte (z.B. `Piecewise` für Stückweise-definierte Funktionen, sofern implementiert).

Diese Operator-/Funktions-Knoten sind im Code hierarchisch organisiert. Oft existieren ein oder zwei Abstraktionsebenen: So können z.B. alle binären arithmetischen Operatoren eine gemeinsame Basisklasse haben. Im Habr-Beispiel (Entwurf des Autors) wurde etwa eine Einteilung in **OperatorEntity** vs. **FunctionEntity** diskutiert (Knoten mit Kindern vs. Blätter) [22]. Im finalen Code sind die Kategorien Continuous/Discrete eher auf Namensebene (Dateistruktur) sichtbar; die *Vererbungshierarchie* selbst ist flach gehalten: `Entity` ist Basisklasse, viele konkrete Subtypen erben direkt davon (teils mit Partial Classes). **Jeder konkrete Knotentyp ist typischerweise** `sealed`, während einige abstrakte

Gruppentypen existieren, um gemeinsame Eigenschaften zu definieren (z.B. alle trigonometrischen Funktionen könnten intern eine gemeinsame Basisklasse teilen, was aber nach außen nicht sichtbar ist) [17].

- `MathS` – *Statische Hilfsklasse* (im Namespace `AngouriMath`). `MathS` bildet die primäre **API-Fassade** des Core-Moduls. Sie bündelt eine Vielzahl von **Factory-Methoden**, globalen Einstellungen und Hilfsfunktionen:
- **Funktions-Fabriken:** `MathS` stellt *statische Methoden* bereit, um neue Knoten/Entities zu erzeugen. Beispielsweise `MathS.Sin(Entity x)` erstellt einen Sinus-Knoten mit dem gegebenen Operand; `MathS.Var("x")` erzeugt eine neue Variable; `MathS.Sqrt(Entity e)` eine Wurzel etc. [23]. Viele solcher Aufrufe sind zur Bequemlichkeit auch als **Kurzformen** verfügbar, etwa als statische Importe. So kann der Nutzer wahlweise `MathS.Sin(x)` oder – nach `using static AngouriMath.MathS;` – einfach `Sin(x)` schreiben. Außerdem existieren Methoden für Konstanten: z.B. `MathS.pi` (π), `MathS.e` (Eulersche Zahl), `MathS.i` (imaginäre Einheit i), die als vordefinierte Entities bereitstehen.
- **Parsing:** Die Methode `MathS.FromString(string expr)` parst einen Ausdruck aus einem String und gibt den entsprechenden `Entity` zurück [24]. Diese Funktion nutzt einen internen Parser, der die Eingabe in den Ausdrucksbaum umwandelt. Alternativ existiert ein *impliziter Typumwandler* von `string` auf `Entity`, so dass man auch durch Zuweisung oder direkte String-Literale Ausdrücke erstellen kann (z.B. `Entity expr = "x + 2";` funktioniert durch implizites Parsen) [25]. Standardmäßig verwendet `FromString` ein Caching der Parse-Ergebnisse, um wiederholtes Parsen identischer Strings zu vermeiden; dies kann durch einen Parameter deaktiviert werden [26]. Wichtig: das Parsen (ob implizit oder via `FromString`) liefert den rohen Ausdruck, *ohne automatische Vereinfachung*.
- **Globale Einstellungen:** Über `MathS.Settings` können verschiedene Optionen beeinflusst werden [27] [28]. Beispielsweise:
  - `MathS.Settings.DowncastingEnabled` (bool): steuert, ob bei der Zahlenerstellung automatisch in „kleinere" Zahltypen konvertiert wird (z.B. 1.0 -> Integer) [29].
  - `MathS.Settings.ComplexityCriteria`: Funktion/Delegate zur Berechnung der Komplexität eines Ausdrucks, welche die Vereinfachungsalgorithmen beeinflusst [30].
  - `MathS.Settings.MaxExpansionTermCount` o.Ä.: könnte die Grenzen für Reihenentwicklungen, Summenausweitungen etc. setzen (falls vorhanden).
  - `MathS.Multithreading`-Hilfsfunktionen: Erlauben es, Berechnungen auf mehrere Threads zu verteilen bzw. steuern, ob bestimmte Algorithmen parallel ausgeführt werden [27]. Für industrielle Nutzung gibt es hier Mechanismen, um die Zuverlässigkeit zu erhöhen und Berechnungen parallel ablaufen zu lassen (z.B. für die Evaluierung vieler unabhängiger Ausdrücke).
- **Experimentelle Features:** Unter `MathS.ExperimentalFeatures` werden neue oder instabile Funktionen zugänglich gemacht, die noch nicht offiziell Teil der stabilen API sind [31]. Diese können vom Nutzer bewusst aktiviert/genutzt werden, um künftige Möglichkeiten auszuprobieren.

- **Diagnose-Werkzeuge:** `MathS.Diagnostic` enthält Methoden, die hauptsächlich zur Fehlerdiagnose oder internen Tests gedacht sind [32]. Diese sollten in Produktionscode nicht verwendet werden, bieten aber Einsicht in interne Abläufe (z.B. um Performance-Engpässe eines Ausdrucks zu analysieren).

- **Weitere wichtige Klassen im Core:**

- `EquationSystem` (`AngouriMath.Core.EquationSystem`): Repräsentiert ein **Gleichungssystem** aus mehreren Gleichungen. Diese Klasse ist *kein* `Entity`, sondern ein eigenständiger Hilfstyp, um mehrere Gleichungen gemeinsam zu behandeln [33]. Man kann ein `EquationSystem` aus einer Liste von `Entity`-Ausdrücken erstellen (wobei jeder Ausdruck einer Gleichung $f_i(x_1, ..., x_n) = 0$ entspricht – daher sollten sie keinen `Equality`-Knoten enthalten) [18] [34]. Die Methode `Solve` an `EquationSystem` löst das System für gegebene Unbekannte und gibt ein `Matrix`-Objekt zurück [35] [36]. Die Lösungsmatrix enthält in jeder Zeile einen Satz von Lösungen für die Variablen (Spalten) oder ist `null` bei unlösbarem System. `EquationSystem` implementiert auch `ToString` und `Latexise`, um das System lesbar darzustellen [37]. Intern nutzt `EquationSystem` die Einzelgleichungs-Solver des Core (siehe unten), indem es iterativ oder simultan löst.
- `Matrix` (`AngouriMath.Core.Matrix`): Stellt Matrizen dar, die sowohl als Eingabe (z.B. für LGS-Löser, oder für Operationen wie Inverse, Determinante) als auch als Ausgabe (Lösungsmengen, Wahrheitswerttabellen etc.) dienen. Das obige Beispiel `Solve` liefert etwa eine Matrix aller Lösungen [38]. Die Matrix-Klasse unterstützt Indexzugriff, Iteration und wahrscheinlich Basisoperationen (Addition, Multiplikation) – auch wenn AngouriMath kein Schwerpunkt auf numerischer Linearer Algebra hat, werden kleine Matrizen beispielsweise für Wahrheitstabellen (boolesche Auswertung) oder Lösungsdarstellungen verwendet [39].
- `FastExpression` (`AngouriMath.Core.FastExpression`): Repräsentiert eine *kompilierte Funktion*, d.h. einen vorkompilierten Ausdruck, der effizienter ausgewertet werden kann. Statt einen `Entity` jedes Mal neu zu traversieren, kann man mit `expr.Compile(var1, var2, ...)` ein `FastExpression`-Objekt erstellen und dieses wiederholt mit unterschiedlichen Werten auswerten [40] [41]. Intern übersetzt der *Compiler* den `Entity` in einen delegierten .NET-Expression-Tree oder eine eigene VM-Sprache. Die Klasse bietet Methoden wie `Call(params Complex[] values)` bzw. `Substitute(values)` zur Ausführung [42] [43]. Gemäß README ist die Auswertung auf diese Weise typischerweise **15x schneller** als über die rohe `EvalNumerical`-Methode [44]. Die Kompilierung erfolgt im Core mittels statischer Funktionen im Ordner `Functions/Compilation/IntoFE` (z.B. `Compiler`-Klasse) [3].
- `Domain` (`AngouriMath.Core.Domain`): Definiert mögliche Definitions- oder Zielbereiche (Kodomainen) für Ausdrücke, z.B. reelle Zahlen, komplexe Zahlen, etc. Über `WithCodomain(Domain)` kann man z.B. für eine Variable einschränken, dass nur reelle Lösungen gesucht werden sollen [45]. Dieser Mechanismus wird beim Lösen von Gleichungen/Ungleichungen und bei bestimmten Simplifizierungen relevant.
- **Hilfsklassen** in AngouriMath.Core: z.B. `ILatexiseable` (Interface für Objekte, die eine `Latexise()`-Methode haben – implementiert etwa von `Entity` und `EquationSystem`), sowie Exception-Klassen wie `WrongNumberOfArgumentsException`, welche geworfen wird, wenn bei Auswertung die Anzahl der bereitgestellten Werte nicht zu den freien Variablen passt [46]. Auch Fehler wie Division durch Null, ungültige Domänen (z.B. Logarithmus von negativem Wert in reellen Zahlen, falls Domain = RR) werden über eigene Exceptions oder Sonder-Entities (wie `Entity.NaN` o.ä.) behandelt.

Zusammengenommen definieren diese Klassen die **symbolische Infrastruktur** von AngouriMath. `Entity` und seine Subtypen bilden den **Baum der mathematischen Ausdrücke**. `MathS` bietet den **Zugriffspunkt** und Utility-Funktionen für den Nutzer. Weitere Core-Klassen wie `EquationSystem` und `Matrix` ergänzen die Funktionalität für spezielle Anwendungsfälle (Gleichungssysteme, tabellarische Ergebnisse). Die Hierarchie ist weitgehend *statisch festgelegt* – Erweiterung um neue Arten von Knoten erfordert Änderungen im Core selbst, da alle relevanten Operationen (Parsing, Differenzieren, Vereinfachen etc.) angepasst werden müssten.

Öffentliche API-Entry-Points

AngouriMaths Core-Modul stellt dem Anwender eine **einfache, hochgradig integrierte API** zur Verfügung. Die wichtigsten Einstiegspunkte sind:

- **Statische Klasse** `MathS`: Wie oben beschrieben, ist `MathS` die zentrale Schnittstelle. Typische Anwendungsfälle:
- **Ausdrücke erzeugen**: Über `MathS.Var` (Variablen), `MathS.Integer/Real/Complex` (Zahlen), oder direkte Methoden wie `MathS.Sin(...)`, `MathS.Cos(...)` etc., können Benutzer programmatisch Expressions bauen [23]. Beispiel: `var x = MathS.Var("x"); var expr = MathS.Sin(x) + 3;`. Alternativ nutzt man die C#-Operatoren und Typumwandlungen: Man kann einfach schreiben `Entity expr = x * x + 3 * x + 12;` dank überladener Operatoren auf `Entity` (die automatisch die entsprechenden Operator-Entities erzeugen) [47] [48]. Auch Zuweisungen wie `Entity a = 5;` (implizite int->Entity) oder `Entity b = "a + 1";` (implizites Parsen) sind gültig. Diese *eingebauten Konvertierungen und Operatorüberladungen* gestalten die API sehr natürlich – man kann Formeln fast im Klartext schreiben.
- **Parsing**: `MathS.FromString("expression")` parst Strings zu Entities. Beispiel: `Entity expr = MathS.FromString("x + 2");` ergibt denselben Ausdruck wie das Literal `"x + 2"` [24]. Das Parsing versteht gängige mathematische Notation (inkl. Funktionen, ^ für Potenzen, Wurzel, log, trigonometrische Funktionen, Klammern, Vergleichsoperatoren, boolesche Operatoren `and`, `or`, `->` etc.). Es unterstützt sogar Unicode-Symbole (α, β, π direkt im String) und ignoriert bestimmte Whitespaces (z.B. Zeilenumbrüche, was z.B. beim Einlesen von Matrizen nützlich ist – in neueren Versionen können Matrizen als mehrere Zeilen eingegeben werden [49]). Die geparsten Entities werden gecached, sofern nicht deaktiviert [26].
- **Globale Funktionen**: Einige Berechnungen können direkt über `MathS` durchgeführt werden. Z.B. `MathS.Simplify("expression")` als Shortcut für `FromString(...).Simplify()`, oder `MathS.Differentiate("expression", "x")` für Ableitungen ohne manuelles Parsen. Im Wiki werden *Extension-Methoden* erwähnt, die direkt auf `string` aufgesetzt sind, z.B. `"x^2 + 1".SolveEquation(var x)`, `"expr".Simplify()`, was die Nutzung noch weiter vereinfacht [50] [51]. Diese Extensions rufen intern natürlich die `MathS`- und `Entity`-Methoden auf, erlauben aber eine sehr knappe Notation.

- **Einstellungen setzen**: Durch das `MathS.Settings`-Objekt (via `using var _ = MathS.Settings.SomeSetting.Set(value)`) kann man temporär Einstellungen ändern. Z.B. `using var _ = MathS.Settings.DowncastingEnabled.Set(false); ...` schaltet innerhalb des using-Blocks das automatische Downcasting aus [52].

- `Entity`**-Instanzen und ihre Methoden**: Sobald man einen Ausdruck als `Entity` vorliegen hat (sei es durch Parsen oder Konstruktion), bietet die Klasse **viele Instanzmethoden** an, um damit zu arbeiten. Wichtige Methoden/Properties sind:

- `expr.Simplify()` – Führt eine umfassende algebraische Vereinfachung des Ausdrucks durch. Dies nutzt intern den *Simplificator* und *Patterns*-Mechanismus und versucht, einen äquivalenten Ausdruck minimaler Komplexität zu finden. `Simplify()` ist bewusst *keine automatische Hintergrund-Operation*, sondern muss explizit aufgerufen werden, da sie ressourcenintensiv sein kann [53]. Für leichte Bereinigungen existiert `expr.InnerSimplified` als Property: dieser liefert einen schnell vereinfachten Ausdruck (z.B. entfernt 0 in Summen, 1 in Produkten, fasst triviale Dinge zusammen) ohne komplexe Pattern-Matches [54]. Automatische Vereinfachung *während* der Erstellung (wie man es z.B. von CAS-Systemen kennt) implementiert AngouriMath

nicht – das erlaubt dem Benutzer, selbst zu entscheiden, wann er die kostenintensive Simplify-Prozedur anwendet [55].

- `expr.Differentiate(Variable x)` / `expr.Differentiate()` – Bildet symbolisch die Ableitung nach der angegebenen Variable. Rückgabewert ist wieder ein `Entity`, der die analytische Ableitung darstellt. Zusätzlich gibt es `expr.Integrate(Variable x)` für bestimmte Integrale (soweit implementiert) [56]. Die interne Implementierung setzt auf bekannte Ableitungsregeln für alle Elementarknoten (Produktregel, Kettenregel etc.). Nicht alle Integrale kann AngouriMath lösen; das Vorhandensein der `Integrate`-Methode deutet aber auf wenigstens partielle Integrationsfähigkeiten hin.
- `expr.SolveEquation(Variable x)` – Löst eine Gleichung $expr = 0$ nach der gegebenen Variablen auf [57]. Hierbei wird der Ausdruck als linke Seite einer Gleichung $expr = 0$ interpretiert. Die Lösung(en) werden als `Set`-Entity zurückgegeben (meist ein `FiniteSet` der gefundenen Lösungen oder ggf. ein `SpecialSet` wie Ø für keine Lösung). Beispiel: `"2 sin(a x) - b".SolveEquation("x")` liefert $\{(arcsin(b/2) + 2\pi n_1)/a,\ (\pi - arcsin(b/2) + 2\pi n_1)/a\}$ [58]. Für lineare und viele nichtlineare Gleichungen kennt AngouriMath analytische Lösungsverfahren (Polynomgleichungen, trigonometrische Gleichungen mit Umformung auf isolierte trig. Funktion, etc.).
- `expr.Solve(Variable x)` – Löst allgemein einen logischen **Statement**-Ausdruck nach x [59]. Ein Statement kann ein Mix aus Gleichungen und Ungleichungen sein, ggf. mit logischen Verknüpfungen (`and`, `or`). `Solve` beherrscht z.B. Systeme wie $x^4 = 16 \land x \in \mathbb{R} \land x > 0$ in einem Rutsch [60]. Das Ergebnis ist wiederum ein `Set` (z.B. eine einzelne Lösung $\{2\}$ in diesem Fall [61]). `Solve` ist mächtiger als `SolveEquation`, da es auch Ungleichungen und Kombinationen lösen kann – intern werden diese jedoch auf einzelne Lösungsschritte zurückgeführt.
- `expr.EvalNumerical()` / `expr.EvalBoolean()` – Wertet den Ausdruck numerisch bzw. boolesch aus. `EvalNumerical()` versucht, das `Entity` so weit wie möglich auf einen konkreten numerischen Wert zu reduzieren (für gegebene Variablenersetzungen). Dies ist nützlich, um z.B. nach `Substitute` den Zahlenwert zu erhalten. Dabei wird automatisch eine ausreichende Präzision gewählt (z.B. Nutzung von BigFloat via `EDecimal` bei Bedarf). `EvalBoolean()` macht ähnliches für logische Ausdrücke (gibt `Entity.Boolean` zurück bzw. kann implizit zu bool konvertieren). Beispiel: `"1+5".EvalNumerical()` ergibt ein `Entity.Number.Integer` 6, das dann z.B. via `(int)expr` zu 6 castbar ist [62].
- **Substitution und Parameter**: `expr.Substitute(Variable, Entity value)` ersetzt eine Variable durch einen gegebenen Wert/Ausdruck. Dies funktioniert auch mit mehreren gleichzeitig (Methodenüberladungen erlauben etwa `expr.Substitute(("x", 2), ("y", 3))`). Für reine numerische Auswertungen kann man direkt `expr.Substitute("x", 2.5).EvalNumerical()` nutzen. Bei einem kompilierten Ausdruck (`FastExpression`) erfolgt die Substitution über dessen `Call(...)` / `Substitute(...)` Methode, welche eine Reihe von Werten direkt einsetzt und auswertet [43] [63].
- **Weitere Methoden**: `expr.Expand()` multipliziert Klammern aus (Algebraische Expansion), `expr.Factorize()` faktoriziert Ausdrücke soweit möglich. `expr.Alternate(n)` generiert alternative umgeformte Darstellungen des Ausdrucks (bis zu *n* Umformungsiterationen) [64]. Über diese Alternativformen wird dann in `Simplify()` die beste (nach `SimplifiedRate`) ausgewählt [65]. `expr.Latexise()` gibt einen LaTeX-Code für den Ausdruck zurück (praktisch für die Ausgabe). `expr.ToString()` versucht, einen mathematisch lesbaren String zurückzugeben – meist so, dass dieser wieder geparst werden könnte, aber absolute Garantie dafür gibt es nicht in allen Fällen [66].
- **Matrix-Methoden**: Falls `expr` eine Matrix ist, erlaubt AngouriMath Operationen wie `Transpose()`, `Determinant()`, `Inverse()` etc., sofern implementiert. Auch kann man einzelne Elemente mit `matrix[row, col]` ansprechen (Index überladen). Das Lösen von

linearen Gleichungssystemen erfolgt über `EquationSystem` (siehe oben) oder direkt `MathS.Equations(..).Solve(...)` wie im README gezeigt [67] [68].

- **Logik- und Mengenmethoden**: `expr.SolveBooleanTable(var)` erzeugt eine Wahrheitstabelle (in Matrixform) für boolesche Ausdrücke [39]. Mengen bieten z.B. `set.Contains(Entity)` etc., und man kann Mengenausdrücke vereinigen `A.Unite(B)` oder schneiden `A.Intersect(B)` über entsprechende Methoden oder Operatoren (manche sind als `Entity`-Methoden mit sprechenden Namen implementiert, z.B. `expr.In(Entity set)` ergibt einen Statement ob ein Element in einer Menge liegt [69]).

Zusammengefasst interagieren Entwickler mit dem Core-Modul entweder **direkt über `Entity`-Objekte** (die via Operatoren und Methoden manipuliert werden) oder **indirekt über die Hilfsklasse `MathS`** (die das Erstellen und einige häufige Aktionen erleichtert). Diese Dualität ermöglicht es, sowohl *deklarativ* (Strings & high-level MathS-Funktionen) als auch *imperativ* (Objekte & Methoden) zu arbeiten, je nach Bedarf.

Beispiel eines kompletten Workflows in C#:

```
using AngouriMath;
using static AngouriMath.MathS;

// Ausdruck definieren
Entity expr = "x^2 + 2x + 1";

// Vereinfachen (hier: faktorisiere x^2+2x+1 zu (x+1)^2)
Console.WriteLine(expr.Simplify());
// Ausgabe: (x + 1) ^ 2

// Ableiten
Entity deriv = expr.Differentiate("x");
Console.WriteLine(deriv);
// Ausgabe: 2 * x + 2

// Nullstellen lösen
var solutions = expr.SolveEquation("x");
Console.WriteLine(solutions);
// Ausgabe: { -1 }  (eine endliche Lösungsmenge mit Element -1)

// Numerische Auswertung der Ableitung bei x=3
var subs = deriv.Substitute("x", 3).EvalNumerical();
Console.WriteLine(subs);
// Ausgabe: 8
```

Die **F#-API** (im separaten Modul `AngouriMath.FSharp`) ermöglicht idiomatischen F#-Zugriff, ist aber im Kern nur ein Wrapper um die C#-Funktionen. Für Jupyter-Notebooks gibt es das `AngouriMath.Interactive`-Paket, das z.B. LaTeX-Ausgabe automatisch rendert. Diese bauen alle auf den hier beschriebenen Core-Fähigkeiten auf.

Interne Abhängigkeitsanalyse

Innerhalb des Core-Moduls existiert eine durchdachte Trennung von Verantwortlichkeiten. Die **Datenstrukturen** (Ausdrucksknoten, Werte, Variablen etc.) sind weitgehend *losgelöst von den Algorithmen*, welche auf ihnen operieren. Dies zeigt sich in der Projektstruktur:

- Die `Entity`-Klasse und ihre Unterklassen kennen zwar grundlegende Operationen (wie `Differentiate`, `Simplify` etc.), **die eigentliche Logik dieser Operationen ist jedoch ausgelagert** in Hilfsklassen unter `AngouriMath.Functions`. Zum Beispiel ruft `Entity.Simplify()` intern vermutlich den `Simplificator` und `Patterns`-Matcher auf, die im `Functions`-Namespace definiert sind [5]. Ebenso delegiert `Differentiate` an einen zentralen Differenziationsalgorithmus, der für jeden Knotentyp die richtige Ableitung bereithält (ggf. über eine Dispatch-Tabelle oder mittels Pattern Matching auf die Entity-Typen). `SolveEquation` und `Solve` rufen interne Solver-Routinen auf, welche in der Bibliothek implementiert sind – etwa ein Polynomsolver, ein Algorithmus zur Auflösung von trigonometrischen Gleichungen, oder ein Gauss-Eliminationsverfahren für lineare Gleichungssysteme. Diese befinden sich möglicherweise in einem Unterordner `Functions/Algebra` oder direkt im Core (die Gleichungslöser könnten auch in Form von *Methods an Entity* implementiert sein, intern aber Hilfsklassen nutzen).

- **Abhängigkeitsrichtung:** Die Abhängigkeiten verlaufen im Allgemeinen **von den Datenstrukturen hin zu den Funktionsmodulen**, nicht umgekehrt. Das heißt, das `Functions`-Modul greift intensiv auf `Entity` und dessen Untertypen zu (um den Baum zu traversieren, Eigenschaften auszulesen und neue Entities zu bauen). Umgekehrt ist die `Entity`-Klasse relativ unabhängig: ihre Methodenaufrufe könnten theoretisch eigene Implementierungen enthalten, doch aus Wartbarkeitsgründen sind viele Operationen ausgelagert. Dadurch bleibt der Kern der `Entity`-Klasse übersichtlich (Definition der Typen, Operanden, Basis-Funktionen), während die komplizierten Algorithmen modular in eigenen Klassen gepflegt werden können. So kann man z.B. Verbesserungen am Vereinfachungsalgorithmus durchführen, ohne die Entity-Klassen selbst ändern zu müssen – man modifiziert die `Simplificator`-Logik in `Functions.Patterns` oder fügt neue Patternregeln hinzu.

- **Internal vs Public API:** Viele Hilfsklassen im `Functions`-Bereich sind `internal` oder im Namespace `AngouriMath.Functions` verborgen. Sie tauchen in der öffentlichen API nicht auf, tragen aber wesentliche Funktionalität. Zum Beispiel:

- `AngouriMath.Functions.TreeAnalyzer` enthält Methoden zur Bewertung der "SimplifiedRate" eines Ausdrucks und zur Normalisierung der Darstellungsreihenfolge (z.B. Terme sortieren, um Konsistenz zu gewährleisten) [70] [5]. `Entity.Alternate(int)` wird vermutlich TreeAnalyzer nutzen, um permutierte Formen eines Ausdrucks zu generieren und zu beurteilen [71] [72].
- `AngouriMath.Functions.Simplificator` (und `Patterns`) implementieren die eigentlichen Algebra-Regeln: z.B. Identitäten wie $\sin^2 x + \cos^2 x = 1$ oder Faktorisierungs-/Ausmultiplizierungsregeln. Diese Klasse durchsucht den Baum nach Mustern und ersetzt Subbäume entsprechend. Sie arbeitet eng mit `TreeAnalyzer` (Komplexitätsmaß) und `Fraction` (zur Vereinfachung von Brüchen) zusammen [5]. Die Aufrufe von `expr.Simplify()` münden in diesem Modul.
- `AngouriMath.Functions.ExpressionNumerical` implementiert die Auswertung eines Baumes zu einer Zahl. Dies könnte über rekursive Traversierung geschehen oder via

Übersetzung in einen LINQ-Expression-Tree und Kompilierung. Bei reinen numerischen Blättern (Integer, Rational, Real) greift es auf die eingebetteten `EDecimal/EInteger`-Operationen zurück. Für trigonometrische oder special functions nutzt es .NET-eigene Funktionen (z.B. `System.Math.Sin`) oder eigene Implementierungen für erhöhte Präzision.

- `AngouriMath.Functions.TrigonometricAngleExpansion` und `TrigonometryTableValues` enthalten vermutlich spezielle Vereinfachungslogiken für trigonometrische Ausdrücke. Z.B. Umwandlung von $\tan x$ in $\sin x / \cos x$ oder Nutzung bekannter Werte $\sin(\pi/6) = 1/2$ etc., um Ausdrücke zu vereinfachen oder numerisch auszuwerten. Diese könnten sowohl von Simplify als auch von Eval genutzt werden.

- `AngouriMath.Functions.BaseConversion` könnte für die Umwandlung zwischen verschiedenen Darstellungen sorgen (z.B. rational -> gemischte Zahl, oder interne Basis-Representation für Zahlen).

- **Kommunikation zwischen Komponenten:** Der symbolische *Ausdrucksbaum* (`Entity`-Objekte) dient als zentrale Datenstruktur, die zwischen Algorithmen ausgetauscht wird. Beispielsweise generiert der Parser aus einem String einen `Entity`-Baum; dieser wird dann unverändert an den Simplificator gereicht, der einen neuen `Entity` (vereinfachte Form) zurückgibt; dieser kann dann an den Solver gehen usw. Die Komponenten interagieren hauptsächlich **über die Entities**. Es gibt wenige globale Zustände – Einstellungen werden über `MathS.Settings` explizit gesetzt und gelesen, Caching findet kontrolliert (z.B. in `FromString`) statt, aber ansonsten arbeiten die Funktionen rein auf den übergebenen Strukturen.

- **Leistungsmerkmale & Threading:** Der Core ist darauf ausgelegt, auch mit komplexeren Ausdrücken umzugehen, jedoch können bestimmte Operationen exponentielle Komplexität aufweisen (ein bekanntes Problem symbolischer Algebra). Daher wurden Mechanismen eingebaut, um Performance zu steuern:

- Die `SimplifiedRate` hilft, in `Simplify()` unnötig komplexe Alternativformen zu vermeiden [65].
- Der *Cache* für Parsen verhindert redundante Baumkonstruktionen bei mehrfach gleicher Eingabe [73].
- Für aufwändige numerische *Berechnungen* oder umfangreiche Aufrufe bietet `MathS.Multithreading` Unterstützung, um z.B. eine Batch-Berechnung auf mehrere Kerne zu verteilen [27]. Intern sind viele Algorithmen aber sequenziell. Der Nutzer kann jedoch z.B. in einem Data-Parallel-Szenario mehrere Ausdrücke parallel vereinfachen, da AngouriMath-Core keine globale Mutable-States enthält, die dem im Wege stehen (solange man die Settings nicht global verändert).

- Einige Operationen haben eingebaute **Abbruchbedingungen** oder heuristische Limits, um nicht ewig zu laufen (z.B. könnte der Solver nach einer bestimmten Anzahl von Alternativlösungen abbrechen, oder der Simplificator nicht alle Permutationen ausprobieren, sondern heuristisch vorgehen).

- **Externe Abhängigkeiten:** Das Core-Modul hängt von wenigen externen Bibliotheken ab. Hauptsächlich die **PeterO.Numbers** (für EInteger, EDecimal, ERational) zur Behandlung großer Ganzzahlen, Dezimalzahlen und Rationalzahlen. Außerdem wird `System.Numerics.Complex` für komplexe Doppelzahlen eingesetzt, z.B. als Rückgabewert von `FastExpression.Call` (dort als Array von Complex für Parameter) [43]. Die Verwendung dieser Libraries ist gekapselt in den `Entity.Number`-Subklassen. Sonst nutzt AngouriMath nur das .NET-Framework

(Collections, LINQ Expressions ggf., RegEx für Parser etc.). Dadurch bleibt das Core-Modul unabhängig und portabel (NuGet-Paket läuft auf .NET Framework, .NET Core, Mono etc.).

- **Interaktion mit Erweiterungsmodulen:** Module wie `AngouriMath.FSharp` oder `AngouriMath.Interactive` referenzieren das Core-Modul und nutzen dessen öffentliche API. Umgekehrt kennt das Core-Modul diese nicht – es liefert lediglich Hooks (z.B. Latexise-Funktion für hübsche Ausgabe, oder `Entity.ToString()` in einfacher ASCII-Form), die von Interactive genutzt werden können. Das Terminal-Projekt ist im Prinzip ein Wrapper, der Eingaben parst (via Core) und Ergebnisse ausgibt. Diese klare Trennung gewährleistet, dass Änderungen im Core (z.B. neue Funktionstypen) unmittelbar allen Frontends zur Verfügung stehen, ohne dass der Core an diese Frontends angepasst werden muss.

Zusammengefasst ist die Architektur des Core-Moduls zweischichtig:

- **Schicht 1: Symbolische Objekte (Domain Model)** – Klassen im Namespace `AngouriMath` (und teils `AngouriMath.Core` für Hilfstypen) definieren die statische Struktur (Ausdruckstypen, Werte, Variablen, Gleichungssysteme, Matrix). Diese Schicht ist weitgehend zustandslos und rein strukturell.

- **Schicht 2: Algorithmische Funktionen (Services)** – Klassen im Namespace `AngouriMath.Functions` implementieren Operationen auf den symbolischen Objekten (Differentiation, Simplification, Solving, Evaluierung etc.). Diese Schicht enthält die mathematische Logik und kann als eine Sammlung von Services gesehen werden, die das Domain Model verarbeiten.

Die *Entities* kennen ihre grundlegenden Operationen, rufen aber intern diese Service-Schicht auf, was die Wartbarkeit erhöht. Die Entkopplung ist nicht vollständig (z.B. ruft `Entity.Simplify` direkt Simplificator auf – also eine Abhängigkeit, aber eben eine bewusste interne Verknüpfung). Dennoch könnten Teile der Funktions-Schicht ausgetauscht oder verbessert werden, ohne das öffentliche Objektmodell zu verändern.

Erste Einschätzung: Modularität, Erweiterbarkeit, potenzielle Engpässe

Die Architektur des AngouriMath-Core wirkt durchdacht und relativ modular:

- **Modularität:** Die Trennung von Ausdrucksstruktur und konkreten Algorithmen sorgt für eine gute **Modularisierung**. Neue Features können oft in der Function-Schicht hinzugefügt werden, ohne die Entity-Klassen zu ändern. Beispielsweise könnte man neue Vereinfachungsregeln hinzufügen, indem man `Simplificator` / `Patterns` erweitert, oder einen neuen Solver für Differentialgleichungen als separate Klasse implementieren, die Entities verwendet. Die Kern-Entity-Hierarchie ändert sich seltener – sie umfasst die mathematischen Basisbausteine, die sich kaum ändern. Auch die Aufteilung in separate Projekte (Core, FSharp, Interactive, Terminal) zeigt Modularität auf höherer Ebene: Das Core-Modul kann eigenständig genutzt oder mit verschiedenen Frontends kombiniert werden.

- **Erweiterbarkeit:** Für Anwender bedeutet Erweiterbarkeit hier vor allem, welche mathematischen Konzepte ohne Änderung der Bibliothek ausgedrückt werden können. Die vorhandene Hierarchie deckt bereits viele Standardfunktionen ab (Arithmetik, trigonometrische,

logische, Mengen). Sollten Nutzer *neue* Arten von symbolischen Objekten benötigen (z.B. einen neuen Spezialfunktionsknoten für Gamma-Funktion), ist das aktuell nur durch Mitwirken am AngouriMath-Core erreichbar – es gibt kein Plug-in-System, um eigene `Entity`-Subklassen einzuschleusen. Die Bibliothek ist also **geschlossen** für externe Erweiterung auf dieser Ebene (nach dem OCP-Prinzip: closed for modification). Allerdings kann man das vorhandene Gerüst flexibel kombinieren und auch erweitern, indem man z.B. eigene Methoden um die Core-Funktionalität baut. Die `MathS.ExperimentalFeatures`-Klasse deutet an, dass die Entwickler zukunftige Erweiterungen vorsahen, aber kontrolliert einführen. Insgesamt ist die Codebasis (C# mit Partial Classes) aber so gestaltet, dass Contributors relativ leicht neue Patterns oder Funktionen hinzufügen können, da die Struktur klar ist.

- **Wartbarkeit:** Durch die klare Aufspaltung und sprechende Klassennamen (z.B. `TrigonometricAngleExpansion`, `TreeAnalyzer`, `EquationSystem`) ist der Code für Entwickler gut navigierbar. Der Einsatz von Partial Classes (z.B. `Entity` verteilt auf mehrere Dateien nach Kategorien, wie `Entity.Continuous.Operators.Classes.cs` [74]) hilft, die Riesen-Klasse `Entity` in handhabbare Teile aufzuteilen. Dies erhöht die Wartbarkeit, da Änderungen in einem Bereich (z.B. neue trigonometrische Funktion) lokal erfolgen können. Allerdings erfordert jede Änderung an den Entities, dass man alle relevanten Algorithmen anpasst (z.B. fügt man eine neue Funktion hinzu, muss man Parser, Differentiator, Simplifier etc. auch erweitern). Dank der Kapselung in Funktionklassen ist klar, wo diese Anpassungen durchzuführen sind.

- **Performance & potenzielle Engpässe:** Symbolisches Rechnen ist naturgemäß rechenintensiv. Die Architektur versucht dem entgegenzuwirken durch:

- **Optimierungen** wie Caching (Parse-Cache) und Kompilierung (FastExpression) für wiederholte Auswertung [44].
- **Heuristische Steuerung** der Algorithmen (Complexity Score, begrenzte Iterationen in Simplify) [65] [53].
- **Arbiträre Präzision**: Die Verwendung von `EDecimal` & Co. erlaubt exakte/rationale Berechnungen, verhindert aber auch unabsichtliche Überläufe. Dies ist gut für Korrektheit, jedoch können Big Integers und High-Precision-Decimals sehr groß werden und Performance fressen, wenn man unbedacht z.B. $50!$ berechnet.
- **Speicher**: Jeder unterschiedene Ausdruck ist ein eigenes Objekt im Baum. Bei sehr großen Ausdrücken (viele Tausend Knoten) kann dies zum Speicherproblem werden. Die interne Representierung ist rein baumbasiert, keine spezielle Komprimierung außer dem parse-Cache für Wiederholungen. Dies ist typisch und zunächst kein Engpass, aber zu beachten.

Ein möglicher Engpass der aktuellen Architektur ist die **fehlende Integration externer Simplifier oder Solvers**: Alles ist in-house implementiert. Sollte ein Nutzer z.B. komplexere Integrale lösen wollen, stößt AngouriMath evtl. an Grenzen (die Modularität würde es erlauben, an dieser Stelle externe CAS-Software einzubinden, ist aber nicht vorgesehen). Dafür ist AngouriMath spezialisiert auf Kerngebiete und garantiert plattformübergreifend verfügbar.

- **Thread-Safety:** AngouriMath-Core ist im großen Ganzen threadsicher, solange man Settings pro Thread setzt. Entities selbst sind unveränderlich (immutable) – Operationen wie `Simplify()` geben einen neuen `Entity` zurück statt den bestehenden zu ändern. Das bedeutet, man kann den gleichen Ausdruck in mehreren Threads gleichzeitig verschieden bearbeiten, ohne Konflikte. Einzig globale Einstellungen oder caches könnten Race-Conditions erzeugen, aber Settings sind über den `using var _ = ... Set()`-Mechanismus vermutlich *thread-local* (ggf. mittels `AsyncLocal` oder ähnlichem implementiert). Der Parser-Cache ist global; in Szenarien mit

vielen verschiedenen Threads, die dieselben Strings parsen, könnte es Synchronisationsbedarf geben – Details dazu sind nicht dokumentiert, aber vermutlich geringes Risiko (und zur Not abschaltbar). Insgesamt scheint aber die Architektur auf **Nebenläufigkeit** vorbereitet (siehe Multithreading-Klasse) und vermeidet globale Zustände so weit wie möglich.

- **Qualität und Dokumentation:** Das Vorhandensein einer ausführlichen Wiki/Dokumentation, automatisch generierter API-Doku und Community-Beiträge (Blogs auf Habr, Reddit) zeigt, dass das Design auch nach außen hin kommuniziert wurde. Das erleichtert die Nutzung und erhöht die Langfristigkeit. Allerdings weist das GitHub-Repo darauf hin, dass AngouriMath inzwischen als *deprecated* markiert ist (Weiterentwicklung ungewiss) [75]. Die Architektur selbst ist aber solide; die Deprecation dürfte eher aufgrund von Ressourcenmangel oder einer Nachfolgeplanung (vielleicht AngouriMath 2.0?) erfolgt sein.

Fazit: Das Core-Modul von AngouriMath ist architektonisch klar strukturiert. Es bietet eine **saubere Trennung** zwischen dem symbolischen **Datenmodell** (Ausdrucksbaum) und den **Operationen** darauf. Komponenten wie Vereinfachung, Lösung von Gleichungen und Differentiation sind modular aufgebaut, was die Bibliothek relativ leicht wart- und erweiterbar macht (zumindest für Mitentwickler). Die **API** für den Endnutzer ist gut durchdacht und vereinfacht komplexe Vorgänge in wenige Methodenaufrufe – dies weist auf eine gelungene Kapselung der komplexen internen Abläufe hin.

Die **Modularität** zeigt sich auch darin, dass zusätzliche Frontends (F#, Terminal, Interactive) ohne Änderungen am Core umgesetzt wurden. Mögliche **Engpässe** liegen im Bereich Performance bei hochkomplexen symbolischen Aufgaben – ein allgemeines Problem, das AngouriMath durch caching und compile-Funktionen abmildert, aber nicht vollständig löst. Für typische Anwendungsfälle (mittlere Ausdrucksgrößen, Standardprobleme) ist die Architektur jedoch **gut geeignet** und erlaubt dem Nutzer, sich auf die Mathematik zu konzentrieren, während das Core-Modul die internen Details kapselt und handhabt.

---

[1] [7] [8] [39] [40] [41] [44] [67] [68] [75] GitHub - asc-community/AngouriMath: New open-source cross-platform symbolic algebra library for C# and F#. Can be used for both production and research purposes.
https://github.com/asc-community/AngouriMath

[2] [18] [19] [33] [34] [35] [36] [37] [38] AngouriMath: AngouriMath.Core.EquationSystem Class Reference
https://codedocs.xyz/asc-community/AngouriMath/classAngouriMath_1_1Core_1_1EquationSystem.html

[3] [42] [43] [46] [63] AngouriMath: AngouriMath.Core.FastExpression Class Reference
https://codedocs.xyz/asc-community/AngouriMath/classAngouriMath_1_1Core_1_1FastExpression.html

[4] [9] [10] [20] [21] [45] [56] [69] [70] AngouriMath | docs/AngouriMath/Entity
https://am.angouri.org/docs/AngouriMath/Entity.html

[5] [6] AngouriMath | docs/AngouriMath.Functions
https://am.angouri.org/docs/AngouriMath.Functions.html

[11] [12] [13] [27] [28] [29] [31] [32] [52] [62] AngouriMath | docs/AngouriMath.MathS
https://am.angouri.org/docs/AngouriMath.MathS.html

[14] [50] [57] [58] [59] [60] [61] Solvers · asc-community/AngouriMath Wiki · GitHub
https://github.com/asc-community/AngouriMath/wiki/Solvers

[15] [16] [17] Compilation of math functions into Linq.Expression / Habr
https://habr.com/en/articles/546926/

22 47 48 Developing a symbolic-expression library with C#. Differentiation, simplification, equation solving and many more / Habr
https://habr.com/en/articles/486496/

23 24 25 26 66 73 Basic operations · asc-community/AngouriMath Wiki · GitHub
https://github.com/asc-community/AngouriMath/wiki/Basic-operations

30 51 53 54 55 64 65 71 72 AngouriMath | wiki/Simplification
https://am.angouri.org/wiki/Simplification.html

49 AngouriMath 1.3 update - Habr
https://habr.com/en/articles/565996/

74 AngouriMath/Sources/AngouriMath/Core/Entity/Continuous/Entity.Continuous.Operators.Classes.cs at ffc1d6efcaae3b2429d3d95d028e0027e1d00d38 · asc-community/AngouriMath · GitHub
https://github.com/asc-community/AngouriMath/blob/ffc1d6efcaae3b2429d3d95d028e0027e1d00d38/Sources/AngouriMath/Core/Entity/Continuous/Entity.Continuous.Operators.Classes.cs