ChatGPT

Integration von PGA und CGA in AngouriMath Core

Einführung: Zielsetzung und Nutzen

Projektive und Konformale Geometrische Algebra (PGA bzw. CGA) ermöglichen es, **geometrische Objekte und Transformationen** einheitlich algebraisch darzustellen. Durch die Integration dieser Algebren in AngouriMath können Punkte, Geraden, Ebenen (in PGA) sowie zusätzlich Kreise, Kugeln etc. (in CGA) symbolisch behandelt werden. Insbesondere erweitert PGA den euklidischen Raum um *eine* Dimension (Homogene Koordinate) und umfasst damit Punkte, Geraden, Ebenen sowie starre Bewegungen (Rotationen, Translationen) in einem einzigen algebraischen System [1]. CGA erweitert den Raum um *zwei* Dimensionen und enthält darüber hinaus runde Objekte wie Kreise und Kugeln sowie konforme Transformationen (Skalierungen, Spiegelungen, Inversionen) [2]. Beide Algebren erlauben **Join- und Meet-Operationen**, d. h. das bequeme Bilden höherdimensionaler Hüllen (Join/$\wedge$) und Schnittobjekte (Meet) aus gegebenen Objekten [2] [3]. Die symbolische Verfügbarkeit dieser Konzepte in AngouriMath würde z. B. ermöglichen, **Schnittpunkte von Geraden und Ebenen** zu berechnen, geometrische **Invarianzen** algebraisch zu prüfen und komplexe Bewegungsabläufe (über Motoren/Rotors) analytisch zu behandeln – alles innerhalb des vorhandenen CAS-Rahmens.

Mathematische Modellierung: GA-Definition, Signaturen und Regeln

Geometric Algebra (GA) ist eine Clifford-Algebra, die auf einem Vektorraum mit einer bestimmten Signatur (metrischer Form) basiert. Basisvektoren e_i erfüllen $e_i^2 = \pm 1$ oder 0 je nach Metrik (positiv, negativ oder null). In PGA arbeitet man typischerweise im Raum $R^{3,0,1}$ (für 3D; analog $R^{2,0,1}$ für 2D), der drei euklidische Basisrichtungen und eine zusätzliche *Ideal*-Basis e_∞ (mit $e_\infty^2 = 0$) enthält. CGA für 3D verwendet einen Raum $R^{4,1}$ (oft beschrieben als $R^{3,0,2}$ mit zwei Nullbasen e_+ und e_-) [1]. Diese zusätzlichen Basisvektoren (e_+, e_- bzw. e_∞) erlauben es, Punkte im Unendlichen darzustellen und runde Objekte zu modellieren. So sind in CGA e_+ und e_- Nullvektoren mit $e_+^2 = e_-^2 = 0$ und typischerweise $e_+ \cdot e_- = 1$ (im passenden Maßstab).

Algebraische Produkte: GA definiert drei fundamentale Produkte: - Das **geometrische Produkt** (kein spezielles Symbol, oft einfach Juxtaposition) ist *nicht kommutativ* und vereint ein symmetrisches Skalarprodukt und ein antisymmetrisches äußeres Produkt: $a\,b = a\cdot b + a\wedge b$. - Das **äußere Produkt** $\wedge$ (Wedge) bildet das *Join* zweier Objekte: es ergibt den **kleinsten gemeinsamen Superspace** der Operanden [3]. Es ist vollständig antisymmetrisch ($e_i \wedge e_i = 0$; $e_i \wedge e_j = -\,e_j \wedge e_i$ für $i\neq j$) und entspricht der üblichen Orientierung (z. B. $e_1 \wedge e_2 \wedge e_3$ ist das orientierte Volumenelement). In der Tat ist in einem 3D-Euklidraum $I = e_1 \wedge e_2 \wedge e_3$ der Einheitspseudoskalar [4]. In PGA-3D käme noch e_∞ hinzu ($I_{PGA}=e_1\wedge e_2\wedge e_3\wedge e_\infty$), in CGA-3D analog $I_{CGA}=e_1\wedge e_2\wedge e_3\wedge e_+\wedge e_-$. Das äußere Produkt zweier GA-Entitäten ergibt immer die **höchstdimensionale Vereinigung** ihrer Inhalte – z. B. ist $P \wedge Q$ (zwei Punkte) in PGA die Gerade durch P und Q, in CGA entspricht $P \wedge Q$ einer Kreislinie bzw. Linie (durch P,Q, ∞) je nach Kontext. - Das **innere Produkt** (Skalarprodukt) $a \cdot b$ kontrahiert Vektoren in einen

niedrigerdimensionalen Anteil (Meet-Operation). Es ist definiert über die Metrik $\eta_{ij} = e_i\cdot e_j$; insbesondere gilt $e_i \cdot e_j = 0$ für $i\neq j$, und $e_i \cdot e_i = \begin{cases} \pm 1 &\text{falls nicht null,}\ 0 &\text{für Nullvektoren.}\end{cases}$ Das **Meet** zweier Objekte kann mittels Innerem Produkt und Dualität ermittelt werden [3] [5]. Beispielsweise entspricht der *antiprojektive* Produkt (Anti-Wedge) in GA dem Durchschnitt: $A \,\cap\, B = A \;\raisebox{0.1em}{$\lrcorner$}\; B =$ (Dual A) $\wedge$ (Dual B) dual. In unserer symbolischen Implementierung kann dies als eigener Operator oder via Dualisierung erreicht werden.

Signaturen und Objektmodelle: Durch die gewählte Signatur werden bestimmte GA-Objekte möglich: - In **PGA ($R^{3,0,1}$)** repräsentiert e_∞ den Punkt im Unendlichen (der "Idealpunkt"). Eine affine/euklidische Koordinate (x,y,z) erweitern wir homogen zu $(x,y,z,1)$ [6], was einem Punkt entspricht. Algebraisch kann man ihn als Schnitt von drei Ebenen sehen. Ebenen selbst sind durch eine lineare Gleichung darstellbar und erscheinen im GA-Kontext als 3-Blades (Trivectoren) in der 4D-Algebra [7] – z. B. entspricht eine Ebene $ax+by+cz+d=0$ einer GA-Entität $\Pi = a\,(e_2\wedge e_3\wedge e_\infty) + b\,(e_3\wedge e_1\wedge e_\infty)+c\,(e_1\wedge e_2\wedge e_\infty)+d\,(e_1\wedge e_2\wedge e_3)$ [7] (dies ist ein triviales Linearformat des Trivectors $x e_{234} + y e_{314} + z e_{124} + w e_{321}$ [8]). Eine **Gerade** in PGA-3D ist der Schnitt zweier Ebenen und erscheint als 2-Blade (Bivector). Ihre Darstellung kann z. B. in *Plücker-Koordinaten* erfolgen: eine Gerade besitzt Richtungsanteile und einen Moment (Aufpunkt-Information) – entsprechend wird sie als Summe von Basis-Bivectoren dargestellt [7] (z. B. $l_{vx} e_{41} + l_{vy} e_{42} + \dots + l_{mz} e_{12}$ für eine 3D-Gerade [7]). Ein **Punkt** schließlich (als Schnitt von drei Ebenen) ist im projectiven GA-Kontext ein Pseudoskalar-Multivector (4-Vector); praktisch können wir ihn aber einfacher als homogenen Vektor $(x,y,z,1)$ behandeln und intern in die entsprechende 3-Blade-Dualform überführen. *Flat Points* (Richtungen) sind 1-Vektoren der Form $v_x e_1 + v_y e_2 + v_z e_3 + 0\cdot e_\infty$ – sie repräsentieren Punkte "im Unendlichen" (Richtungsvektoren) [7].

- In **CGA ($R^{4,1}$)** gibt es neben den euklidischen $e_{1,2,3}$ zwei Nullbasen e_+ (oft als e_0) und e_- (e_∞). Hiermit lassen sich **runde Objekte** darstellen. Ein **Punkt** in CGA wird als *Round Point* durch einen 5-dimensionalen Vektor repräsentiert [9]. Gegeben einem euklidischen Punkt (x,y,z) bilden wir z. B. $P = x e_1 + y e_2 + z e_3 + 1\cdot e_+ + \frac{x^2+y^2+z^2}{2}\,e_-$ [10]. (Hier ist e_+ der "Anker" mit Koeffizient 1, e_- trägt den halben quadratischen Abstand – so wird P ein Nullvektor mit $P \cdot P = 0$ gemäß der Signatur.) Eine **Kugel** mit Mittelpunkt $C=(a,b,c)$ und Radius r hat analog die Darstellung $S = (a e_1+b e_2+c e_3 + \frac{a^2+b^2+c^2-r^2}{2}\,e_- + 1\cdot e_+)$, was einem 1-Vektor entspricht. Umgekehrt kann man solche Objekte auch als *Blades höherer Grade* auffassen: In CGA-3D ist eine Kugel ein 4-Vector (Quadrivektor) [11], ein **Kreis** ein 3-Vector (Trivektor) [12], eine **Gerade** oder **Paar von Punkten** (Dipol) ein 2-Vector (Bivektor) [13], und eine **Ebene** ein 3-Vector, nämlich der Spezialfall einer Kugel durch e_∞ (unendlicher Radius, was S flach macht). Diese Zuordnungen entsprechen dem mathematischen Modell der Terathon-Library: *"Die Klasse Sphere3D speichert eine Kugel als fünfdimensionalen Quadrivektor"* [11], *"Circle3D … als fünfdimensionaler Trivektor"* [12], *"Dipole3D … als fünfdimensionaler Bivektor"* [13], *"RoundPoint3D … als fünfdimensionaler Vektor"* [9] in CGA.

Zusammengefasst definieren wir also neue GA-Basisvektoren e_i je nach gewählter Signatur (für PGA z. B. $i=1..3$ plus e_∞, für CGA $i=1..3$ plus e_+, e_-). Deren algebraische Produkte folgen den obigen Regeln. Insbesondere werden symbolische **Vereinfachungsregeln** festgelegt wie $e_i \wedge e_i = 0$, $e_i \cdot e_i = \eta_{ii}$ (z. B. 0 für e_∞ bzw. $e_{\pm}$, $+1$ für $e_{1..3}$ in euklidischem Teilraum, ggf. -1 für zeitartige Komponenten in anderen GA), und $e_i e_j = -\,e_j e_i$ für $i\neq j$ (Antikommutativität der Basis im äußeren Anteil). Zudem erkennen wir spezielle Kombinationen: etwa wird $e_1\wedge e_2 \wedge e_3 \wedge e_\infty$ in PGA als Pseudoskalar I

behandelt, der benutzt werden kann, um Dualitäten zu berechnen. Diese Regeln ermöglichen es, GA-Entitäten symbolisch zu manipulieren und zu vereinfachen, als wären es “eingebaute” Objekte des CAS.

Entwurf der symbolischen Datenstrukturen für GA-Objekte

Um PGA und CGA in AngouriMath abzubilden, werden **neue Entity-Typen** eingeführt, die Blades und Multivektoren repräsentieren. Ähnlich wie bereits Matrizen als `Entity.Matrix` implementiert sind [14], soll es eine Hierarchie für GA geben, z. B.:

- Eine abstrakte Basisklasse `Entity.GA` (oder spez. `GeometricAlgebraEntity`), die allgemeine Funktionalität bietet (Speichern der Signatur, gemeinsame Operationen etc.).
- Davon abgeleitet konkrete Klassen oder Strukturen für **Multivektoren** in bestimmten Räumen. Beispielsweise könnten wir `Entity.GA.PGA3D` und `Entity.GA.CGA3D` definieren, um die jeweiligen 3D-Algebren zu kapseln. Diese Klassen halten intern die Komponenten eines Multivektors (in einer geeigneten Darstellung, s. u.).
- Alternativ kann auch eine generischere Klasse `Entity.GA.Multivector` alle Fälle abdecken, mit einer internen Angabe der Signatur. Diese würde dann je nach initialer Konstruktion wissen, ob es sich z. B. um ein PGA- oder CGA-Objekt handelt.

Interne Repräsentation: Ein Multivektor lässt sich als Summe von Basis-Blades speichern. Im Code kann das durch z. B. eine Bitmaske oder Indizes für die beteiligten Basisvektoren pro Term geschehen. Ein einfaches Schema: jeder Basisvektor e_i bekommt eine feste Indexposition; ein Blade $e_{i} \wedge e_{j}\wedge\cdots$ kann durch eine sortierte Menge bzw. Bit-Kombination dieser Indizes identifiziert werden. Die Entity speichert dann etwa eine Map von dieser Blade-ID auf den Koeffizienten (wiederum ein `Entity`, meist eine Zahl oder symbolische Variable). Beispielsweise könnte der Multivektor $3 + 2e_1e_2$ intern als `{ mask 0: 3, mask (e1^e2): 2 }` gehalten werden. Für konkrete geometrische Objekte definieren wir Werks-Funktionen, die solche Multivektoren entsprechend initialisieren: - **Punkt in PGA:** z. B. `MathS.GA.PGA.Point(x, y, z)` erzeugt einen Multivektor der Form $P = x e_1 + y e_2 + z e_3 + 1\cdot e_\infty$. (Im Hintergrund würde dies als trivector $e_\infty \wedge (x e_1 + y e_2 + z e_3)$ oder als Dual eines entsprechenden Blades gespeichert, aber der Nutzer kann es wie ein Vektor mit homogener Koordinate behandeln.) - **Ebene in PGA:** `MathS.GA.PGA.Plane(a, b, c, d)` liefert das Ebenen-Objekt $a e_1 + b e_2 + c e_3 + d e_\infty$ (dies repräsentiert die Ebene $ax+by+cz+d=0$ in dualer Form als trivector, wie oben erläutert). Alternativ könnte man auch `Plane(P1, P2, P3)` anbieten, die durch drei Punkte eine Ebene aufspannt (intern $\Pi = P1 \wedge P2 \wedge P3$ berechnet). - **Gerade in PGA:** `MathS.GA.PGA.Line(P1, P2)` gibt die Verbindungsgerade durch zwei Punkte $P1,P2$ via $L = P1 \wedge P2$ (Ergebnis ist ein Bivector). Ebenso könnte `Line(dir, point)` mit Richtungsvektor und Aufpunkt genutzt werden (der Aufpunkt als wedge mit e_∞ ergibt den Moment-Part). Im Hintergrund würden die Plücker-Koordinaten berechnet und als entsprechende Blade-Kombination gespeichert [7]. - **Punkt in CGA:** `MathS.GA.CGA.Point(x, y, z)` analog wie oben, ergibt $x e_1 + y e_2 + z e_3 + \frac{x^2+y^2+z^2}{2} e_- + 1\cdot e_+$ [10]. - **Kugel in CGA:** `MathS.GA.CGA.Sphere(center, r)` könnte eine Kugel aus Mittelpunktpunkt und Radius erzeugen, etwa indem erst $C = MathS.GA.CGA.Point(cx,cy,cz)$ gebildet wird und dann $S = C - \frac{r^2}{2} e_-$ (so entspricht S dem oben genannten 5D-Vektor einer Kugel). - **Gerade in CGA:** `MathS.GA.CGA.Line(P1, P2)` oder `Line(direction, point)` analog PGA (nur dass hier $L = P1 \wedge P2$ ein Dipole-Bivector in CGA ergibt). - **Kreis in CGA:** `MathS.GA.CGA.Circle(P1, P2, P3)` als Schnitt einer Ebene mit einer Kugel durch $P1,P2,P3$ oder direkt als trivector $C = P1 \wedge P2 \wedge P3`.

`- **Ebene in CGA:**` MathS.GA.CGA.Plane(P1, P2, P3) `– Ebene durch drei Punkte, resultiert in einem 3-Blade (entspricht einer Sphere durch $\infty$).`

`Alternativ` Plane(normal, dist) `ähnlich PGA, wobei` dist` hier den Abstand vom Ursprung darstellen könnte (die Implementierung würde eine entsprechende CGA-Fläche erzeugen, z. B. als Kugel mit Radius ∞).

Intern sind all diese speziellen Entitäten nichts weiter als Multivektoren mit bestimmter **Grade**-Struktur (Blades festen Ranges): - Grade-0: Skalar (z. B. reelle Zahl). - Grade-1: Vektor (z. B. in PGA ein FlatPoint bzw. reiner Richtungsvektor; in CGA RoundPoint oder Richtung). - Grade-2: Bivector (in PGA: Linie; in CGA: Dipole = Linie/Punktpaar). - Grade-3: Trivector (in PGA: Ebene; in CGA: Kreis oder Ebene). - Grade-4: Quadrivector (in PGA: Punkt – dual gesehen; in CGA: Kugel oder Dual zur Linie). - Grade-5: Pseudoscalar (nur in CGA 5D vorhanden – z. B. Volumenelement).

Die **Klassenstruktur** kann entweder für jede Objektart eigene Klassen vorsehen (z. B. `PlaneEntity`, `LineEntity` als spezialisierte Unterklasse von GA-Entity) oder – pragmatischer – alle GA-Entitäten mit einer einheitlichen Multivektor-Klasse abbilden und lediglich via Methoden/Properties kenntlich machen. Wir tendieren zur zweiten Lösung: Ein einzelner Entity-Typ `Multivector` mit Eigenschaften wie `.Grade` (bei homogenen Blades) bzw. Methoden wie `.Grades()` (für multigrade MV) reicht aus. Spezielle Methoden können prüfen, ob ein Multivektor z. B. *rein grade-2* ist (dann handelt es sich um eine Gerade etc.). Die Fabrikmethoden `MathS.GA.*` nehmen dem Nutzer die manuelle Konstruktion ab und sorgen für korrekt formatierte Multivektoren.

Wichtig ist, dass GA-Entities *vollständig in den bestehenden Expression Tree integriert* werden. Das heißt, ein GA-Entity ist eine Unterklasse von `Entity` und kann überall verwendet werden, wo sonst Symbole, Zahlen, Matrizen etc. auftreten. Insbesondere unterstützen wir: - **Arithmetische Kombinationen:** GA-Entitäten lassen sich addieren, subtrahieren (ergibt Summen von Multivektoren), und skalieren (Multiplikation mit normalen Zahlen/Skalaren). Eine Summe verschiedener Grades bleibt ein allgemeiner Multivektor. - **Symbolische Gleichungen:** Man kann Gleichungen formulieren, in denen GA-Objekte vorkommen (z. B. $Line(P,Q) \cdot Plane(R,S,T) = 0$ könnte die Bedingung für einen Schnittpunkt sein). Die Gleichungslöser von AngouriMath könnten so erweitert werden, dass sie GA-Gleichungen behandeln, zumindest in speziellen Fällen (z. B. Schnittprobleme auf lineare Gleichungssysteme zurückführen). - **LaTeX-Ausgabe und Parsing:** GA-Entitäten sollten sich in lesbarer mathematischer Form darstellen. Wir planen, Basisvektoren als $e_1, e_2, \dots, e_\infty, e_+, e_-$ auszugeben (ggf. fett oder mit Indizes). Das äußere Produkt könnte als `\wedge` und das innere als `\cdot` in LaTeX erscheinen. Beispiel: `Latexise(e1 ^ e2 + 3)` -> "$3 + e_{1}\wedge e_{2}$". Das **Parsen** von GA-Ausdrücken erfordert mögliche Erweiterungen: Entweder reservieren wir bestimmte Namen (`e1`, `e2`, ..., `eInf`) als GA-Basis (kontextabhängig), oder wir bieten separate Parser oder Initialisierung an. Denkbar ist ein Modus `MathS.Settings.UseGA("PGA3D")`, der beim Einlesen eines Strings `'e1 + e2'` automatisch GA-Entities erzeugt statt allgemeiner Symbole. Alternativ bleiben wir zunächst bei der Program-API (MathS.GA.* Funktionen) und erweitern den Parser in Zukunft.

Operatoren und symbolische Auswertung in GA

Ein zentrales Stück der Integration sind die **Operatorüberladungen für GA**. In AngouriMath existieren bereits arithmetische Operatoren (+, -, , ^ *für Potenzen). Für GA kommen neu hinzu: - Wedge-Produkt (∧): Wir führen einen neuen binären Operatorknoten im Expression Tree ein, der das äußere Produkt repräsentiert. In der C#-API können wir den XOR-Operator* `^` für GA-Entities zweckentfremden*, analog zur Terathon-C++-Library* [15] *– das ist möglich, da in C# benutzerdefinierte Operatoren erlaubt sind. So könnte man im Code* `var L = P ^ Q;` *schreiben, was vom Overload als* `P.Wedge(Q)` *interpretiert wird. (Wir beachten, dass* `^` *normalerweise geringe Präzedenz hat, daher müssen in komplexen Ausdrücken Klammern gesetzt werden* [15] *. Im Parser für Input-Strings werden wir* `^` *nicht nutzen, um Konflikte mit Exponentiation zu vermeiden –*

dort könnte man ersatzweise ein Unicode-Symbol ∧ oder schlicht ein Funktionswort `wedge()` *einführen.) - Inneres Produkt (·): Da es in C# keinen direkten "Dot"-Operator gibt, definieren wir das Skalarprodukt als Methode, z. B.* `A.Dot(B)` *oder statische Funktion* `MathS.GA.Dot(A,B)`*. Alternativ könnte man den Bitwise-AND* `&` *überladen, um* `A & B` *als $A\cdot B$ zu verwenden, doch das wäre eher ungebräuchlich. Für die symbolische Engine entsteht jedenfalls ein eigener Knotentyp* `InnerProductNode` *im Ausdrucksbaum. Dieser Operator berechnet den unteren Gradanteil: z. B. $e_i \cdot e_j = \eta_{ij}$ und $e_i \cdot (e_j \wedge e_k) = $ Term, der $e_j \cdot e_k$ enthält usw. (nach bekannten Rechenregeln der GA). - Geometrisches Produkt: Das* geometric product *vereinigt beide obigen. Wir können entscheiden, ob wir das vorhandene* `*` *dafür verwenden – d.h. wenn zwei GA-Entities mit* `*` *multipliziert werden, führt die Engine einen GA-Produktknoten aus. Dabei müssen wir die Kommutativitätsannahme aufheben, da $A * B \neq B * A$ im Allgemeinen. In AngouriMath ist* `*` *normalerweise kommutativ; wir werden daher im GA-Kontext die Multiplikationsknoten als nicht-kommutativ markieren. Dies kann z. B. durch einen speziellen Node-Typ oder ein Flag geschehen. Eine Möglichkeit: Wir implementieren das geometrische Produkt ebenfalls als eigenen Operator (z. B. Unicode-Symbol ⟑, das Eric Lengyel vorgeschlagen hat* [16]*, oder schlicht als spezielle Funktion), um Verwechslungen zu vermeiden. Wahrscheinlicher ist jedoch, dass wir in der API** `A * B` zulassen und intern abfangen: sofern A oder B GA-Entitäten sind, behandelt der Simplifier es als geometric product (GPNode), andernfalls wie gewohnt als kommutative Multiplikation. Für fortgeschrittene Nutzer könnten wir zudem $\unicode{x27D1}$ (⟑) als kennzeichnendes Symbol anbieten.

Simplifikation und Rechenregeln: Sobald diese Operatoren im Baum vorhanden sind, werden **Auswertungs- und Vereinfachungsregeln** hinterlegt: - Das äußere Produkt wird antisymmetrisch ausgewertet: Terme, in denen ein Basisvektor doppelt auftritt, verschwinden sofort ($e_i \wedge e_i = 0$). Die Reihenfolge wird normalisiert (z. B. $e_2 \wedge e_1$ wird zu $-\,e_1 \wedge e_2$ umsortiert). Bei Summen wird das Distributivgesetz angewendet ($A \wedge (B+C) = A\wedge B + A\wedge C$). Diese Regeln implementieren wir direkt im Operator-Node, sodass bereits beim Konstruktor oder Simplify-Aufruf die Normalform erreicht wird. - Das innere Produkt reduziert Grade: $e_i \cdot e_j = \eta_{ij}$ wird zu einem Skalar (1, -1 oder 0) vereinfacht. $e_i \cdot (e_j \wedge e_k)$ wird z. B. via Identität $a \cdot (b\wedge c) = (a\cdot b)\,c - (a\cdot c)\,b$ umgeformt. Allgemein werden bekannte algebraische Regeln ausgenutzt, um Ergebnisse zu vereinfachen (AngouriMaths Pattern-Matching-System kann hierzu genutzt werden). Insbesondere sollte das Ergebnis eines reinen Treffens zweier kompatibler Objekte wieder ein geometrisches Objekt ergeben: z. B. $Plane \cdot Line$ in PGA ergibt ein Skalar (der Abstand, sofern normalisiert), $Sphere \cdot Plane$ in CGA könnte die Schnittkreis-Entität liefern, etc. Solche höherwertigen Interpretationen könnten in Form von Abkürzungen oder speziellen Methoden (z. B. `Meet(A,B)`) gekapselt werden. - Das geometrische Produkt wird i.A. in $\wedge$- und $\cdot$-Teil zerlegt berechnet. Allerdings können wir auch direkt Regeln implementieren: z. B. $e_i * e_i = \eta_{ii} + 0$ (weil sowohl $\wedge$- als auch $\cdot$-Teil beitragen) und $e_i * e_j = e_i \wedge e_j + e_i\cdot e_j = e_i \wedge e_j$ für $i\neq j$ (da $e_i\cdot e_j=0$). Für zusammengesetzte Multivektoren nutzen wir die Bilinearität und definieren das Produkt entsprechend. Da dies komplex sein kann, konzentrieren wir uns anfangs auf *nicht allzu tiefe* Vereinfachung – Nutzer können das Resultat als Summe von Blades akzeptieren. Trotzdem werden triviale Vereinfachungen wie Faktor 0/1, Ausmultiplizieren usw. durchgeführt. - **Kompatibilität mit normalen Entitäten:** Ein Skalar (normale Zahl oder symbolischer Ausdruck) wird als GA-Entity Grade-0 behandelt. Die GA-Operatoren sind so definiert, dass ein Skalar immer voll mitmultipliziert (skalieren) kann: $s \wedge X = sX$ (falls s ein gewöhnliches $Entity.Number$ ist, wandeln wir es intern ggf. in eine GA-Entity vom Typ Scalar um, oder wir erlauben gemischte Operationen). Ebenso soll $s * X$ das gleiche wie $X * s$ ergeben (Skalare kommutieren mit allem und beeinflussen nur den Koeffizient). Diese Handhabung erleichtert die Mischung von GA und klassischen Ausdrücken. - Wir integrieren die GA-Entitäten in die generelle **Simplify-Logik** von AngouriMath. Das bedeutet, wir ergänzen den Simplifier um Fälle für GA-Nodes. Z. B. kann eine Summe von GA-Multivektoren auf Gleichartigkeit geprüft und zusammengefasst werden (ähnlich wie Polynomterme gesammelt werden). Auch sollte eine Kombination wie $P - P$ (zwei

identische Punkt-Entitäten) vereinfacht zu $0`werden. Hierfür überschreiben wir in`GeometricAlgebraEntity`Methoden wie`IsEqual`,`Simplify` etc., um GA-spezifische Normierungen durchzuführen.

Dualität und Zusatzelemente: Oft ist es hilfreich, den *Dual* eines Blades zu bilden (z. B. um Meets zu berechnen oder orthogonale Komponenten zu finden). Wir planen, für jede GA-Signatur einen konstanten **Pseudoskalar I** bereitzustellen (z. B.`MathS.GA.PGA.Pseudoscalar` könnte $e_1 e_2 e_3 e_\infty$ liefern). Eine Methode`Dual()` oder ein Operator (vielleicht`~` Tilde) könnte dann implementiert werden, die $A^* = A I^{-1}$ berechnet. So könnte man $Meet(A,B) = Dual(Dual(A) \wedge Dual(B))$ intern umsetzen [3] [5]. Die API kann dem Nutzer wahlweise direkte`Meet(a,b)`/`Join(a,b)` Funktionen bieten, oder er nutzt $\wedge$ und`Dual()` selbst. Für Verständlichkeit werden wir aber wohl`GA.Join(A,B)` und`GA.Meet(A,B)` bereitstellen, welche intern die entsprechenden Operationen durchführen.

Beispielhafte API-Nutzung

Um die vorgeschlagenen Erweiterungen greifbarer zu machen, hier einige **Beispiele** (in C#-ähnlicher Syntax), wie man mit PGA/CGA in AngouriMath arbeiten könnte:

- **Punkt, Gerade, Ebene in PGA:**

```
var P  = MathS.GA.PGA.Point(1, 2, 3);            // Punkt mit
Koordinaten (1,2,3)
var Q  = MathS.GA.PGA.Point("a", "b", "c");      // symbolischer Punkt
(Parameter a,b,c)
var L  = MathS.GA.PGA.Line(P, Q);               // Gerade durch P und Q
(P ∧ Q)
var n  = MathS.Vector(0, 0, 1);                 // normaler 3D-Vektor
(AngouriMath.Vector)
var Pl = MathS.GA.PGA.Plane(n.x, n.y, n.z, 5);   // Ebene:
0*x+0*y+1*z+5=0, also z+5=0
var X  = MathS.GA.PGA.Meet(L, Pl);              // Schnittpunkt von
Gerade L und Ebene Pl
Console.WriteLine(X.Simplify());               // gibt z.B.
Point(?, ?, ?) oder konkreten Punkt aus
```

Hier sieht man:`Line(P,Q)` verwendet intern $\wedge$,`Meet(L,Pl)` intern das Antiwedge (Dual-Kombination). Man kann anschließend das Ergebnis vereinfachen oder in Koordinaten umwandeln (ggf. via Projektion auf eine Basis).

- **Kreis und Kugel in CGA:**

```
var A = MathS.GA.CGA.Point(0, 0, 0);
var B = MathS.GA.CGA.Point(1, 0, 0);
var C = MathS.GA.CGA.Point(0, 1, 0);
var circ = MathS.GA.CGA.Circle(A, B, C);        // Kreis durch A, B, C
(Trivector)
var S = MathS.GA.CGA.Sphere(A, 2);             // Kugel mit Mittelpunkt
```

```
A und Radius 2
var L = MathS.GA.CGA.Line(B, C);                // Gerade durch B und C
var pts = MathS.GA.CGA.Meet(S, L);              // Meet: Schnittpunkte
von Kugel und Gerade
```

Hier würde `pts` ein Multivektor (Summe von zwei RoundPoints) ergeben, entsprechend den beiden Schnittpunkten. Durch weitere Operation könnte man diese zwei Punkte extrahieren oder weiterverwenden.

- **Algebraische Manipulation:**

```
var X = MathS.GA.PGA.Point("x", "y", "z");
var Y = MathS.GA.PGA.Point("x0", "y0", "z0");
var line = X ^ Y;                               // Nutzer kann ^ als ^
benutzen
var plane = MathS.GA.PGA.Plane(0, 0, 1, -4);    // z-4=0
var dist =
(plane.Dot(X)).Simplify();          // sollte 0 ergeben, falls X auf der
Ebene liegt
```

In diesem Beispiel würde `plane.Dot(X)` das Skalarprodukt zwischen Ebenen-Trivector und Punkt-Pseudoskalar berechnen – in PGA entspricht das (bis auf Normierung) dem Einsetzen von (x,y,z) in die Ebenengleichung. Das Ergebnis `dist` wäre also $z-4$. Setzt man `dist=0`, hätte man die Bedingung, dass X auf der Ebene liegt. Solche symbolischen Überprüfungen sind mit GA-Integration direkt möglich.

- **Differenzieren** (theoretisch): Sollte eine GA-Entität von einem Parameter abhängen (etwa ein Punkt $P(t) = (t, t^2, 0)$), könnte man $\frac{d}{dt}P(t)$ definieren, was komponentweise erfolgt. Für GA-Objekte höherer Grade wäre die Ableitung die Summe der Ableitungen ihrer Koeffizienten. AngouriMath kann das leisten, indem `Entity.GA` die Ableitung überschreibt oder an die Unter-Entities delegiert.

Diese Beispiele zeigen, dass die GA-Integration nahtlos ins bestehende Schema passt. Der Nutzer kann high-level Methoden (`Point`, `Line`, `Sphere`, `Meet` etc.) verwenden, um geometrische Objekte zu erstellen und zu kombinieren. Im Hintergrund werden die GA-Regeln angewendet, sodass man sich auf die Mathematik statt auf die Koordinatentricks konzentrieren kann.

Integration in die AngouriMath-Architektur

Damit PGA und CGA *wirklich Teil des Core-Moduls* werden, sind einige Implementierungsdetails zu beachten:

- **Verortung im Code:** Die neuen Klassen würden im Namensraum `AngouriMath.Core` oder einem Unterordner `AngouriMath.Core.GA` definiert. `Entity` als Basisklasse erhält ggf. neue abgeleitete Typen. Wichtig ist, dass alle bestehenden Mechanismen (Parsing, LaTeX, Simplify, etc.) die GA-Typen kennen oder generisch behandeln. Beispielsweise muss die `Entity`-hierarchy um die Cases für `GeometricAlgebraEntity` erweitert werden, analog zu `Entity.Matrix` und Co.

- **Parsing-Erweiterung:** Falls wir entscheiden, das Parsen von Ausdrücken mit GA zu erlauben, müssten wir im Parser-Workflow Hooks einbauen. Etwa könnte man erkennen: ein unbekannter Identifier "e1" – wenn ein GA-Modus aktiv ist, erzeugen wir kein Symbol `e1` sondern die entsprechende GA-Basis Entity. Das erfordert eine Art *Context* (z. B. `MathS.GA.ActiveAlgebra = Algebra.PGA3D`), damit der Parser weiß, welche Signatur gilt (sonst könnte "e1" in verschiedenen Algebren definiert sein). Alternativ werden wir vorerst auf String-Parsing verzichten und nur die API nutzen, was einfacher ist.
- **Symbolische Gleichheit und Hashing:** Für das Vergleichen von Expressions (z. B. beim Vereinfachen oder in Dictionaries) müssen GA-Entities eindeutig vergleichbar sein. Zwei GA-Multivektoren sind gleich, wenn ihre *kanonischen* Blade-Koeffizienten übereinstimmen. Wir überschreiben also `.GetHashCode()` und `.Equals()` passend, oder implementieren die Vergleichsmethoden von AngouriMath neu für GA. Dabei ist auch die Signatur zu berücksichtigen: Ein Blade mit gleicher Kombination von e_i aber aus unterschiedlichen Algebren darf nicht vermischt werden. Im Grunde sollte eine GA-Entity einen Verweis auf ihren Algebra-Typ tragen und der Vergleich stellt sicher, dass nur innerhalb derselben Algebra geprüft wird. Versucht man zwei Entities verschiedener GA zu kombinieren (z. B. PGA-Punkt ∧ CGA-Punkt), kann ein Fehler oder Ausnahme geworfen werden – solche Mischoperationen sind mathematisch nicht sinnvoll.
- **Interne Simplification-Pipeline:** AngouriMath hat wahrscheinlich eine generische Simplify-Funktion, die rekursiv Nodes vereinfacht. Wir integrieren GA-Regeln entweder *lokal* (in den Operator-Knoten selbst) oder *global* (im Pattern-Matching-System). Z. B. könnten wir ein Muster hinzufügen: `$E \wedge $E -> 0` für identische Sub-Entities E (Antikommutativitätsregel) oder `$a * $b -> ...` für GA-Multiplikation. Da wir aber viel spezielle Logik haben, ist es sauberer, innerhalb der GA-Klassen zu vereinfachen. Beispielsweise implementiert `GA.Multivector.Simplify()` das Zusammenfassen von Termen und ruft für Unterentitäten (Koeffizienten) ebenfalls Simplify auf. So wird das hierarchisch aufgelöst.
- **LaTeX und Ausgabe:** Wir erweitern die Ausgabe-Funktionen. In AngouriMath gibt es typischerweise eine `.ToString()` und `.Latexise()` Methode pro Entity. Für GA formatieren wir z. B. Summenteile durch Sortieren der Basis-Kombinations-Indizes und schreiben $e_{i}\wedge e_{j}$ oder $e_{i}e_{j}$ je nach Kontext. Nullvektoren können speziell behandelt werden: evtl. e_∞ statt e_{4} ausgeben, um Benutzerfreundlichkeit zu erhöhen. Diese String-Repräsentationen sollten auch in der Debug/`ToString` lesbar sein (z. B. "1 + 2*e1^e2" für $1 + 2 e_{12}$).
- **Performance-Überlegungen:** GA-Operationen können schnell zu vielen Termen führen (durch Ausmultiplizieren). Wir sollten daher so weit möglich *symbolisch faktorisiert* bleiben. Beispielsweise statt $x e_1 + x e_2$ könnten wir $x*(e_1+e_2)$ speichern, solange x ein gemeinsamer Faktor ist. Das erfordert aber erweiterte Erkennung im Simplifier. Zumindest für lineare Kombinationen sollten wir dies berücksichtigen. Ebenso könnte bei wiederholten Produkten eine Kürzung mit Inversen* möglich sein (z. B. $A \wedge (A^{-1}\wedge B) = B$, falls A^{-1} das passende inverse Element ist). In GA gibt es Versoren, die inverse via $A^{-1} = \tilde{A} / (A \tilde{A})$ etc. Wir können solche speziellen Vereinfachungen zunächst außen vor lassen und in Zukunft ergänzen.
- **Differenzierung und Integration:** Falls unterstützt, implementieren wir im Differentiator eine Fallunterscheidung: die Ableitung einer GA-Entity wird per Produktregel auf ihre Koeffizienten angewendet. Da GA-Basisvektoren konstante Entitäten (nicht abhängig von x etc.) sind, verschwinden deren Ableitungen; nur die Koeffizienten (scalare Funktionen) tragen bei. Somit reduziert sich die Ableitung eines Multivektors $M(x) = \sum_k f_k(x) B_k$ (mit konstanten Blades B_k) zu $\sum_k f'_k(x) B_k$. Dies können wir handhaben, indem `GeometricAlgebraEntity.Differentiate(var)` entsprechend umgesetzt wird.
- **Einbindung in Gleichungslöser:** Theoretisch könnten wir dem Gleichungslöser beibringen, z. B. $Line(P,Q) \wedge X = 0$ nach X aufzulösen (was dem Schnittpunkt entspricht). Solche

Spezialfälle sind aber eher komplex. Im ersten Wurf fokussieren wir uns auf **Darstellung und Simplifikation**; die Integration in Löser geschieht ggf. später durch Reduktion auf skalare Gleichungen (z. B. extrahiere aus X die Koordinaten und löse). Hier ist vor allem wichtig, dass GA-Entitäten in Gleichungen überhaupt auftreten können ohne das System zu überfordern – sprich, Gleichungen mit GA sollen nicht als "nicht unterstützter Typ" gelten.

Modularität und Erweiterbarkeit

Ein explizites Ziel ist, die GA-Integration **modular und erweiterbar** zu gestalten. PGA (RGA) und CGA sind nur zwei mögliche Algebren; zukünftig könnten weitere Räume interessant sein – z. B. 2D-Versionen, Minkowski-Spacetime Algebra (STA) für relativistische Rechnungen, u.a. Um dies zu ermöglichen, schlagen wir folgende Maßnahmen vor:

- Die Implementation trennt klar zwischen **algebraspezifischen Definitionen** (Basisvektoren, Signatur) und dem **generischen Rechenkern** (Multivektor-Operationen). Man kann z. B. eine Struktur `AlgebraDescriptor` einführen, die Anzahl der Dimensionen, Signatur (Liste der metrischen Werte für Basisvektoren) und ggf. Namen/Symbole der Basis enthält. PGA3D wäre dann ein Descriptor mit Dim=4, Signatur `[+1,+1,+1,0]` (oder entsprechendes Format), CGA3D Dim=5, Signatur `[+1,+1,+1,0,0]` mit Angabe welche Indizes Nullvektoren sind und evtl. $e_+\cdot e_- = 1$ falls implementiert als off-diagonales Metrikelement.
- Der `GeometricAlgebraEntity` könnte einen solchen Descriptor referenzieren. So könnten wir zukünftig einfach einen neuen Descriptor hinzufügen (z. B. `"STA4D": [+1, -1, -1, -1]` für Raumzeit 1+3) und damit neue Basisvektoren definieren. Die Rechenlogik (wie $\wedge$, $\cdot$ funktionieren) bleibt dieselbe – nur die Metrik η_{ij} ändert sich entsprechend. Dadurch ist der Kerncode wiederverwendbar.
- In der API könnten wir neue Algebren als weitere Unter-Namespace anbieten, z. B. `MathS.GA.STA` für spacetime algebra, `MathS.GA.PGA2D` für ebene GA, etc. Intern würden sie auf obige Descriptoren verweisen. Diese Modularisierung stellt sicher, dass **nur die benötigten Algebren geladen/genutzt** werden. Der Anwender importiert bzw. nutzt z. B. nur `AngouriMath.Core.GA.PGA` – dann werden auch nur die dafür definierten Basis instanziiert. Technisch können wir Basisvektoren als *Singleton Entities* definieren, die beim ersten Gebrauch erstellt werden (Lazy Initialization). So entsteht kaum Overhead, wenn jemand GA gar nicht nutzt.
- Falls gewünscht, könnte man GA-Unterstützung auch komplett optional halten – etwa in einer getrennten Assemblie `AngouriMath.GA` –, die das Core-Modul erweitert. In diesem Konzept gehen wir aber davon aus, dass die Integration eng genug ist, um ins Kernprojekt aufgenommen zu werden (was wünschenswert ist, um symbolische und geometrische Rechnung nahtlos zu vereinen).

Durch diese Erweiterbarkeit wird AngouriMath zu einer Plattform, auf der sukzessive neue Geometrische Algebren eingebunden werden können, ohne das grundlegende Architekturkonzept zu ändern. Wir legen mit PGA und CGA (nach Eric Lengyels Modell) den Grundstein. Dieser erlaubt bereits jetzt, euklidische Geometrie, starre Körperbewegungen und konforme Abbildungen symbolisch zu behandeln – ein Novum in .NET-basierten CAS. Zukünftige Module könnten darauf aufbauen, z. B. um spezielle Lösungsverfahren für GA-Gleichungen bereitzustellen oder Visualisierungen geometrischer Lösungen zu unterstützen.

Fazit: Das vorgestellte Konzept integriert Projective und Conformal Geometric Algebra in AngouriMaths Core, indem neue Entity-Typen für Multivektoren geschaffen und alle nötigen Operatoren ($\wedge$, $\cdot$, geometrisch) symbolisch implementiert werden. Es stützt sich auf das erprobte mathematische Modell der Terathon-Bibliothek [1] [17], welches die Repräsentation von Punkten, Geraden, Ebenen, Kreisen,

Kugeln usw. in entsprechenden Clifford-Algebren definiert. Durch eine saubere architektonische Trennung können wir diese Algebren modular anbieten und bei Bedarf erweitern. Die Nutzer profitieren von einer **einheitlichen, hochgradig ausdrucksstarken Sprache** für Geometrie innerhalb von AngouriMath: komplexe geometrische Konstruktionen lassen sich knapp formulieren, und AngouriMath übernimmt die algebraische Verarbeitung – von der Vereinfachung bis zur Lösung von Problemen – **symbolisch genau**, ohne auf numeräre Approximation angewiesen zu sein. [3] [9]

---

[1] [2] [16] [17] Projective Geometric Algebra
https://projectivegeometricalgebra.org/

[3] [5] Join and meet - Conformal Geometric Algebra
https://conformalgeometricalgebra.org/wiki/index.php?title=Join_and_meet

[4] [PDF] Geometric Algebra: An Introduction with Applications in Euclidean ...
https://scholarworks.sjsu.edu/cgi/viewcontent.cgi?article=7943&context=etd_theses

[6] [7] [8] [15] GitHub - EricLengyel/Terathon-Math-Library: C++ math library for 2D/3D/4D vector, matrix, quaternion, and geometric algebra.
https://github.com/EricLengyel/Terathon-Math-Library

[9] [10] [11] [12] [13] Terathon-Math-Library/TSConformal3D.h at main · EricLengyel/Terathon-Math-Library · GitHub
https://github.com/EricLengyel/Terathon-Math-Library/blob/main/TSConformal3D.h

[14] AngouriMath 1.3 update / Habr
https://habr.com/en/articles/565996/