Inhaltsverzeichnis

Einleitung

In dieser Arbeit ging es mir darum, die Anforderungen an ein Verfahren zur möglichst exakten Vorhersage der Konformation von organischen Moleküls genauer kennen zu lernen.
Eine solche Vorhersage ist beispielsweise in der Biochemie sehr wichtig, wo man durch die Geometrie der Proteine mögliche Reaktionsmöglichkeiten und Funktionen erahnen kann.
Bekanntermaßen gibt es in der Chemie einige sehr einfache Modelvorstellungen über die räumliche Struktur der Moleküle, besonders bekannt ist hier zum Beispiel das Tetraedermodell.
Kann man ein solches Modell für die Entwicklung eines Computerprogrammes benutzten, dass eine Konformation aus einer eingegebenen Lewisformel berechnet?
Und wenn das der Fall ist, ist ein solches Programm brauchbar, oder weicht die berechnete Struktur so gravierend von der Realität ab, dass sie komplett nutzlos ist?
Dies sind interessante Fragen, besonders wenn man dieselben für ein gegenteilige Verfahren stellt. So soll Dirac nach (Mic07) schon 1929 gesagt haben:

> „The fundamental laws necessary for the mathematical treatment of a large part of physics and the whole of chemistry are thus completely known, and the difficulty lies only in the fact that application of these laws leads to equations that are too complex to be solved.“

Also kann man die Struktur komplexer organischen Moleküle aus der Quantenphysik und Quantenchemie[1] berechnen, oder sind die dazu nötigen Gleichungen wirklich unlösbar?
Oder wenn sie doch lösbar sind, kann eine solche Lösung für große Moleküle auf heutigen Computern berechnet werden oder wäre dies zu Rechen- und Speicherintensiv?
Und selbst wenn eine Berechnung nach heutigem Kenntnissstand möglich wäre, sind die dazu nötigen Verfahren hinreichend gut bekannt und dokumentiert, dass man sie ohne tiefgreifende Kenntnisse über die Quantenchemie in nur

[1] Hierbei ist vielleicht anzumerken, dass neuerdings deutliche, makroskopische Indizien für eine andere fundamentale Theorie, die von einer Stringnetz-Flüssigkeit ausgeht, gefunden wurde, die vielleicht noch genauere Vorhersagen ermöglichen könnte, siehe New Scientist Magazin, 15 March 2007, Nummer 2595, Seite 8-9.

drei Monaten lernen und sinnvoll in einem Computerprogramm implementieren kann? Oder kann es sein, dass sie hierfür einfach zu komplex sind?

Mit der Beantwortung dieser Fragen erfährt man die Grenzen für solche Verfahren und weiß infern man Präzision mindestens für Geschwindigkeit, oder umgekehrt, opfern muss, um überhaupt ein sinnvolles Ergebnis zu erhalten.

In diesem Text stelle ich zuerst ein Verfahren, das schnell und mit geringer Genauigkeit eine Konformation ermitteln kann, dar und untersuche es auf seine Probleme hin.
Anschließend beschreibe ich ein übliches quantenchemisches Verfahren, dass verspricht höhere Genauigkeit zu liefern und analysiere auch hier seine Probleme.

Zusätzlich habe ich ein Computerprogramm entwickelt, dass beide Verfahren implementiert und somit durch die Vergleichsmöglichkeiten die unterschiedlichen Vor- und Nachteile nochmal illustriert.
Eine Beschreibung davon findet man im Anhang, zusammen mit dem Ausdruck des Quellcodes.

1 Direkte Konformationsbestimmung

1.1 Ein Tetraederverfahren

Ein für den Computer geeignetes Verfahren, dass die Konformation direkt berechnet, kann man aus dem aus der Schule bekannten Tetraedermodell herleiten.
Bei der Ermittlung einer Struktur geht man von einem Atom, dargestellt eben durch eine orientierten Tetraeder, aus und setzt die an dieses Atom gebundenen Atome so, dass sich jeweils zwei Ecken der beiden Tetraeder für jedes Elektronenpaar berühren, also dass die Tetraeder an Spitze (Einfachbindung), einer Kante (Doppelbindung) oder einer Fläche (Dreifachbindung) aufeinander liegen.
Geht man davon aus, dass sich der Atomkern genau im Zentrum des Tetraeder befindet, so liegt dann die Kern-Kern-Verbindungslinie genau auf der gemeinsamen Symmetrieachse der beiden Tetraeder. Man kann also unterschiedliche Bindungslängen dann dadurch realisieren, dass man die Tetraeder auf dieser Symmetrieachse (auch in einander) verschiebt.
Mathematisch lässt sich ein solcher Tetraeder am einfachsten durch die Koordinaten $\vec{r}_1, .., \vec{r}_4$ darstellen.
Als Koordinatensystem wählt man am einfachsten ein lokales am Atom, bei dem die Position des Atomkerns 0 ist. Dann muss für den Schwerpunkt, dem Zentrum des Tetraeders, $\vec{z} = \frac{1}{4}(\vec{r}_1 + ... + \vec{r}_4)$ gelten, dass $S = 0$ ist.
Addiert man nun jeweils 2 oder 3 Eckpunktkoordinaten, so ergibt sich auf Grund der Symmetrie des Tetraeders genau das Dreifache des Vektors zum Mittelpunkt der Kante oder Fläche.
Das heißt also, da im anderen Tetraeder dieselbe Verhältnisse gelten, dass wenn man die Koordinaten einer, zweier oder dreier Ecken addiert sich immer ein Vektor ergibt, der genau zum jeweils anderen Atomkern zeigt.
Diesen muss man nur noch entsprechend der Bindungslänge skalieren und erhält somit die Koordinaten des nächsten Atomkerns.
Nun hat man nur noch das Problem, um diesen Kern herum wieder einen Tetraeder zu konstruieren.
Dies geht allerdings relativ einfach, denn wenn man zwei zusammenhängende Tetraeder hat, sieht man leicht, dass der zweite Tetraeder genau der erste, gespiegelt an der Ebene senkrecht zur Kern-Kern-Verbindungslinie zwischen ihnen ist.
Noch einfacher ist es, eine spiegelnde Ebene parallel zu dieser zu benutzen, die

durch den Kern des ersten Atoms verläuft, denn das Verschieben einer Spiegelebene verschiebt nur das Spiegelbild und ändert nicht die Verhältnisse. Eine so verschobene Ebene erzeugt den Tetraeder nun genau so, dass das Zentrum des Tetraeders im Nullpunkt liegt, da ja die Ebene durch diesen verläuft und somit das alte Zentrum nicht geändert wird.
Nun kann man das nächste Atom in genau derselben Weise betrachten, man legt einfach wieder ein Koordinatensystem durch den Kern und setzt die Koordinaten der mit diesem Atom verbundenen Atome auf die Koordinaten der Tetraederecken, von dem durch die Spiegelung entstandenen Tetraeder.
Man muss noch beachten, dass mindestens eines der verbundenen Atome bereits positioniert wurde und mindestens eine Ecke bereits belegt ist. Dies lässt sich einfach dadurch berücksichtigen, dass man Ecken in einer bestimmten Reihenfolge besetzt und die gerade belegten Ecken ans Ende der Liste schiebt.
Um die Abstände der Tetraeder zu bestimmen braucht man noch eine Möglichkeit die Bindungslänge zu ermitteln. Ich benutze dafür die Formel von Schomaker und Stevenson die bei (Kro03) erwähnt wird und, die die Bindungslänge als Summe der kovalenten Radien und der skalierten Elektronennegativität ausdrückt. Dies ist natürlich ziemlich ungenau, nicht jedoch im Vergleich zu der Winkelungenauigkeit durch das Tetraedermodells. Diese ist so hoch, das prinzipiell wahrscheinlich sogar eine Einheitslänge funktionieren würde.

1.2 Genauigkeitsprobleme

Diese Methode ist schön schnell, was aber natürlich leider mit einer extrem hohen Ungenauigkeit bezahlt wird, da jede Atombindung einzeln betrachtet wird und die wichtigen Einflüsse der benachbarten Elektronen und Atome nicht berücksichtigt werden.
Die Effekte bei einem Atom sind vorallem die Veränderung der Bindungswinkel und -länge durch benachbarte Bindungen, wie z.B.: die Änderung des Bindungswinkels von Wasser auf 105°.
Ein bekanntes Modell, das dies (leider nur qualitativ) beschreibt ist das VSEPR-Modell (Zsc93, s.23).
Da es nur qualitativ ist, kann man es nicht direkt in einem Computerprogramm verwenden, man könnte allerdings versuchen die qualitativen Effekte mit einfachen Formel oder einer Erfassung aller Möglichkeiten zu quantifizieren.
Ein weiteres Problem ist die Erfassung der Rotationswinkel um Einfachbindungen und Diederwinkeln an Doppelbindungen, die sich sehr leicht ändern. Bei diesem Tetraederalgorithmus sind alle Winkel einheitlich festgelegt, was

natürlich chemisch nicht korrekt ist. Da diese Winkel aber von den Wechselwirkungen von allen Teilchen im Molekül abhängen, glaube ich nicht einem direkten Verfahren überhaupt möglich diese zu berechnen, man muss also ein Energieoptimierung benutzen.
Da außerdem die Atom der Reihe nach angeordnet werden, ist das Verfahren vollkommen ungeeignet für Moleküle die Ringen enthalten, wie beispielsweise bei Benzol, weil die Atome im Ring der Reihe nach betrachtet werden und die letzte Bindung vollkommen ignoriert wird, da das Atom dann ja schon positioniert worden ist.
Auch wenn lange Ketten mit Verzweigungen vorkommen, kann es vorkommen das sich zwei Ketten im Ergebnis kreuzen, sogar so stark, dass zwei Atomkerne übereinander positiniert werden. Dies ist natürlich physikalisch unmöglich.
Wenn dies passiert ist die berechnete Anordnung im Grund noch nicht mal als vorläufige Näherung zu gebrauchen, da die Atome dann sehr große Strecken bis zu einer korrekten Konformation zurücklegen müssten, was mit präzisen quantenchemischen Berechnung wahnsinnig lange dauern kann.
Noch schlimmer dabei ist es, dass bei der Quantentheorie ja nur von einer Anordnung von Elektronen und Kernen ausgegangen wird und nicht vom chemischen Modell der Bindung.
Es kann daher durchaus vorkommen, dass, wenn die Energie durch das Auflösen einer Bindung stark sinken würde, dies auch passiert. Damit erhält man zwar eine optimiertes Molekülgeometrie, nicht jedoch die von einem anderen Molekül.
Vor allem bei komplexen Biomolekülen, wie z.B.: Proteinen, kann es zu solchen Effekten kommen, da es dort ja viele freie Ketten gibt.
Eine mögliche Lösung hierfür wäre es, die Atome nicht gleich in der Tetraederform anzuordnen, sondern für jedes Atom die gewünschte Struktur und die Abweichung davon zu berechnen. Die Stärke der Abweichung kann man dann als Energie betrachten und die im nächsten Abschnitt genannten mathematischen Verfahren der Energieoptimierung darauf anwenden, wodurch man wieder schrittweise eine bessere Geometrie erhält.
Dadurch verliert man zwar gerade einen Großteil des Geschwindigkeitsvorteil, trotzdem ist dies noch sehr viel schneller als das quantenchemische Verfahren, da nur Wechselwirkungen zwischen zwei Atomen betrachtet werden.

Zusammenfassend kann man sagen, dass sich dieses Tetraederverfahren gut für alle Berechnungen von Molekülen eignet, die keine komplexen Strukturen wie Ringe oder lange Seitenketten enthalten, wenn man mit groben Näherun-

gen zufrieden ist und in erster Linie ein schnelles Ergebnis braucht.
Sinnvolle Anwendungsfälle sind also beispielsweise die Vorberechnung bei einfacheren Molekülen um sinnvolle Strukturen für einen komplexeren Algorithmus, für didaktische Zwecke, da das Tetraedermodell ja auch in der Schule vorkommt und letztendlich auch für eine Vorschau eines eingegebenen Moleküles.

2 Energieoptimierung

2.1 Mathematische Verfahren

Bei der Energieoptimierung, oder in diesem Fall bei der Geometrieoptimierung, wird versucht die Paramter einer Energiefunktion solange zu varieren bis diese ein Minimum erreicht.
Dies ist in erster Linie ein mathematisches Problem für das es viele Lösungswege gibt. Das einfachste ist das Verfahren des steilsten Abstiegs , dabei wird die Funktion nach allen ihren Parametern abgeleitet, wodurch man einen Vektor im Parameterraum erhält, der in die Richtung des größten Anwachsens der Energie zeigt.
Das übliche Modell davon ist es, die Energiefunktion als Potentialhyperfläche im n-dimensionalen Raum zu betrachten, also als ein n-dimensionales Gebirge. Die Ableitung liefern dann eine Tangente an der steilsten Fläche, an der man dann ein Stück hinab „rutscht“.
Dies wird dann solange wiederholt bis sich entweder die Energie oder die Ableitung kaum noch ändert.
Eine genaue mathematische Beschreibung hiervon findet sich in (BV04, s.463-478).

2.2 Probleme

Bei allen Optimierungsverfahren gibt es das Problem der Ausgangssituation. Man benötigt eine realistische Annäherung an das tatsächliche Optimum, um die Berechnung überhaupt durchführen zu können.
Hat man eine solche nicht, kann es leicht passieren, dass man bei der Optimierung in die Nähe eines anderen, lokalen Minimus gerät, dessen Energie zwar nicht minimal ist, jede kleine Bewegung jedoch eine höhere Energie gibt, bis ein Potentialberg überwunden ist.
Da man dies im Grund nicht oder nur sehr aufwändig verhindern kann, muss man eine gute Möglichkeit finden eine solche Anfangskonformation zu wählen. Hierbei ist beispielsweise die im ersten Abschnitt beschriebene Tetraedermethode geeignet.
Weiterhin taucht bei den Berechnungen im Rahmen dieses Optimierungsverfahrens das Problem auf, dass man die erste Ableitung der Energiefunktion kennen muss. Dies ist besonders dann problematisch, wenn diese Funktion sehr kompliziert ist, oder nicht geschlossen mathematisch dargestellt werden

kann, sondern stattdessen von einem Algorithmus berechnet wird, wie es bei der Energiefunktion des SCF-Verfahren der Fall ist. Dann kann man nämlich nur schwer eine analytische Ableitung bestimmen, die dann auch noch sehr schwer (also extrem rechenintensiv) auszuwerten ist.
Man kann dieses Problem dadurch umgehen lösen, dass man die Ableitung nach einer Koordinate numerisch berechnet, indem man die Energiefunktion an zwei Parameterpunkten berechnet und den Differenzenquotienten auswertet.
Dies hat natürlich auch Nachteile, die beiden größten sind, dass die Berechnung der Energiefunktion sehr aufwendig ist, vor allem, wenn man sie an $2n$-Punkten auswerten muss.
Außerdem gehen bei der numerischen Ableitung alle Informationen über die Punkte zwischen den beiden gewählten Punkten verloren. Deshalb muss man entweder große Sprünge(=Fehler) in kauf nehmen oder die Schrittweite sehr klein wählen, wodurch die Optimierung aber sehr viel länger dauert.
Ein weiteres Problem bei diesem Verfahren ist, dass es in der Nähe des Minimums sehr langsam wird und, wenn das Minimum übersprungen wird, anfangen kann um dieses Minimum herum zu oszillieren, ohne je zu einem Ende zu kommen.
Es gibt ein besseres Verfahren, das sogenannte Raphson-Newton-Verfahren, dass in (Rei94), (vU80, s. 138) und (BV04, s.484-542) näher beschrieben wird.
Die Grundidee ist, dass man über die Berechnung der zweiten Ableitung sehr viel schneller zum Minimum finden kann, indem man die Nullstelle der ersten Ableitung sucht.
Hierin sieht man auch schon den großen Nachteil dieser Funktion, man braucht die 2te Ableitung, die noch schwerer und langsamer als die erste zu berechnen ist.
Deshalb habe ich im Programm nur die langsamere Methode verwendet.

3 Energieberechnung

3.1 Das SCF-Verfahren

Diese Ausführung des Verfahrens ist eine Zusammenfassung von u.a. (Coo74, s. 1-54) und (Rei94, s. 260-300).
Bei der quantenchemischen Energieberechnung geht man von der Schrödingergleichung $\mathbf{H}\Psi = E\Psi$ aus und versucht eine gültigen Wellenfunktion Ψ mit einer minimalen Energie E zu ermitteln.
$\mathbf{H} = \mathbf{T}_K + \mathbf{T}_e + \mathbf{V}_{KK} + \mathbf{V}_{ee} + \mathbf{V}_{eK}$ ist der Hamiltonoperator.
$\mathbf{T}$ ist die kinetische Energie und $\mathbf{V_{ij}}$ die Energie der Wechselwirkung zwischen den Teilchen i und j.
Also $\mathbf{T}_K$ ist die kinetische Energie der Kerne, $\mathbf{T}_e$ die der Elektronen, $\mathbf{V}_{KK}$ die Wechselwirkungsenergie zwischen je zwei Kernen, $\mathbf{V}_{ee}$ die zwischen Elektronen und $\mathbf{V}_{eK}$ schließlich die zwischen Kernen und Elektronen.
Die einzelnen Operatoren werden aus der klassischen Mechanik und Elektrodynamik hergeleiten, indem man davon ausgeht, dass jeder Kern und jedes Elektron genau an einem Punkt lokalisiert ist.
Es ergibt sich in SI-Einheiten für jeweils ein Teilchen $\mathbf{T}_i = \frac{\bar{h}^2}{2m_i}\nabla_i^2$ und $\mathbf{V}_{ij} = \frac{Z_i Z_j e^2}{r_{ij}^2}$, wobei r_{ij} der Abstand zwischen Teilchen i und Teilchen j, m_i die Masse des Teilchens i, Z_i die Ladung, ∇_i^2 der Laplaceoperator (die Summe der zweiten Ableitung der Wellenfunktion bezüglich aller Koordinaten des Teilchens i), e die Elementarladung und $\bar{h}$ die Planckkonstante geteilt durch 2π ist.
Der gesamte Summand ergibt sich dann durch die Summe aller Einteilchenoperatoren, also z.B.: $\mathbf{T}_e = \sum_{i=1}^{N_e} T_i$, wenn N_e die Summe aller Elektronen ist.
Eine weitere Forderung an die Wellenfunktion Ψ ist, dass sie normiert ist, also $\Psi^*\Psi = 1$, wobei *, das komplex konjungierte meinte.
Daraus folgt das Umstellen der Schrödingergleichung $E = \mathbf{H}\Psi/\Psi = \Psi^*\mathbf{H}\Psi$ ergibt, wobei man allerdings über den gesamten Raum integrieren muss, um die Gesamtenergie zu erhalten, also $E = \int \Psi^*\mathbf{H}\Psi dV$. Dies ist auch Postulat 5 der Quantentheorie, (Pre95, vgl. s. 31).

Genaugenommen ist dieser Hamiltonoperator nur eine Näherung, da er die magnetischen Effekte der unterschiedlichen Spinzustände der Elektronen ebenso wenig wie die Erkenntnisse der Relativitätstheorie berücksichtig werden, da sie schwer zu berechnen wären und nur geringe Einflüsse auf das Ergebnis haben.
Trotzdem ist die Berechnung einer exakten Lösung schon sehr kompliziert, praktisch gilt sie sogar als unmöglich, wenn mehr als ein Atom vorkommt.

Aus diesem Grund führt man eine Reihe von weiteren Näherungen ein.
Die erste ist die sogenannte Born-Oppenheimer-Näherung, man setzt $T_K = 0$, da die Kerne sehr viel schwerer als die Elektronen sind, wodruch sie sich viel weniger bewegen und weniger Bewegungsenergie tragen (Eine genaue mathematische Begründung findet sich in (Rei94, s.142)).
Außerdem macht es auch ja keinen Sinn, eine Kernanordnung von sich bewegenden Kernen zu bestimmen, allerhöchsten könnte man dann, wie bei den Elektronen, Orbitale angeben.
Bei bewegungslosen Kernen ist weiterhin die Kernabstoßungsenergie V_{kk} und die Wellenfunktion konstant (siehe wieder (Rei94, s.142)).
Damit folgt für die Gesamtenergie $E_{ges} = E_e + E_k = E_e + V_{kk}$.
V_{kk} lässt sich direkt berechnen, E_e ist eine Lösung der Schrödinger-Gleichung für Elektronen $\mathbf{H}_e\Psi_e = E_e\Psi_e$ mit $\mathbf{H}_e = \mathbf{T}_e + \mathbf{V}_{ee} + \mathbf{V}_{ke}$ und Ψ_e als Wellenfunktion der Elektronen.
Ausgehend vom chemischen Modell der Orbitale, kann man die Wellenfunktion Ψ_e als Produkt von Molekülorbitalen ψ_i betrachtet, was sich auch mathematisch ergibt, sofern man die Elektronenwechselwirkung ignoriert (Rei94, s.262ff.).
Genaugenommen handelt es sich aber nicht um ein einzelnes Produkt, denn auf Grund dem Pauliprinzip muss die entstandene Wellenfunktion antisymmetrisch bezüglich der Elektronenkoordinaten sein. Also wenn man die Koordinaten zweier Elektronen vertauscht, was gleichbedeutend ist mit der Vertauschung der Orbitale in denen sie sich aufhalten, muss sich das Vorzeichen der Funktion ändern.
Mathematisch wird dies üblicherweise am besten durch eine Determinante ausdrücken, und zwar durch die sogenannte Slater-Determinante:

$$\Psi_e = \frac{1}{\sqrt{N!}} \begin{vmatrix} \psi_1(x_1) & \psi_1(x_2) & \psi_1(x_3) & \cdots & \psi_1(x_n) \\ \psi_2(x_1) & \psi_2(x_2) & \psi_2(x_3) & \cdots & \psi_2(x_n) \\ \psi_3(x_1) & \psi_3(x_2) & \psi_3(x_3) & \cdots & \psi_3(x_n) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \psi_n(x_1) & \psi_n(x_2) & \psi_n(x_3) & \cdots & \psi_n(x_n) \end{vmatrix}$$

Die Orbitale $\psi_i(x_j)$ sind dabei Spinorbitale, die sich also nicht nur in ihrer räumlichen Position unterscheiden, sondern auch in dem Spin den die Elektronen in ihnen haben.
Die Elektronen versuchen natürlich sich in den energetisch günstigsten Orbitalen aufzuhalten, daher befinden sich bei normalen, ungeladenen Molekülen mit geschlossenen Schalen jeweils zwei (nach dem Pauli-Prinzip die maximal

mögliche Zahl) Elektronen in einem Ortsorbital, also sind nur $n/2$ Ortsorbitale besetzt.
Bezeichnet man die Ortsorbitale mit ϕ_i, so sind die vollständigen Orbitale durch $\psi_i = \phi_i\alpha$ oder $\psi_i = \phi_i\beta$ gegeben, je nach Spinzustand (α oder β) des Orbitals ψ_i.
Um nun eine vollständig geschlossene Form der Molekülorbitale zu erhalten, muss man nur noch die Gestalt der Molekülorbitale ermitteln.
Geht man von dem chemischen Konzept des Molekülbaus durch Atome aus, so ergibt sich leicht die Idee, dass die Orbitale eines einzelnen Atoms sich in den Bindungen überlagern, also dass die Molekülortsorbital Überlagerungen der Atomortsorbitale sind. Dieses Modell ergibt natürlich keine exakte Lösungen, aber für die meisten Fälle eine hinreichend gute.
Nun sind Wellenüberlagerungen durch Addition der einzelnen Wellen gegeben, man kann also schreiben:
$\phi_i = \sum_{j=1}^{M} c_{ij}\chi_j,$
wobei M die Anzahl aller Atomorbitale (von allen Atome) ist, c_{ij} die Parameter darstellen und χ_j das Atomorbital Numero j ist. Ein schöne Übersicht über diese Näherungen findet man in (Coo74, s.84).

Setzt man nun die durch die Determinante gegebene Wellenfunktion in das obrige Postulat 5 ein, so ergibt sich nach (HW94, s.405-409) und (Rei94, s.273-276): $E = \sum_{i=1}^{N_e} H_i + \frac{1}{2}\sum_{i=1}^{N_e}\sum_{j=1}^{N_e}(V_{ij,ij} - V_{ij,ji})$
Dabei bedeutet H_i die unabhängige Energie eines Elektrons in einem Orbitals, also das Ergebnis der Anwendung des Operators $(\mathbf{T}_i + \sum_{k=1}^{N_k}\mathbf{V}_{ik})$ auf die Wellenfunktion des Orbital i, und $V_{ij,kl}$ die Elektronenwechselwirkung zwischen zwei Elektronen, das erste gleichzeitig in den Orbitalen i und j, das zweite gleichzeitig in den Orbitalen k und l, also die Anwendung des Operators $\mathbf{V}_{ef}$ auf diese beiden Orbitale.
Zur einfacheren Schreibweise definiert man einen Einelektronenoperator $\mathbf{h}_i = \mathbf{T}_i + \sum_{k=1}^{N_k}\mathbf{V}_{ik}$. Dann ergibt sich präzise:
$H_i = \int \psi_i(e)^*\mathbf{h}_e\psi_i(e)dV_e d\sigma_e$
$V_{ij,kl} = \int\int \psi_i(e)^*\psi_j(f)^*\mathbf{V}_{ef}\psi_k(e)\psi_l(f)dV_e d\sigma_e dV_f d\sigma_f$
Der Parameter von ψ gibt das Elektron an, dass sich im Orbital befindet, dV_e ist das Volumenelement des Elektrons e und $d\sigma_e$ dasselbe für den Spin.
Haben die Orbitale nicht alle denselben Spinwert, ergibt die Integration über die Spinkoordinaten 0 und das ganze Integral verschwindet, haben sie denselben, kann man den Spin ignorieren, da er auf 1 normiert ist.

In einem normalen, nicht ionisieren Molekül sind nur die $N_e/2$ Molekülorbitale mit der geringsten Energie besetzt, allerdings doppelt von zwei Elektronen mit unterschiedlichem Spin.
Man kann dann wie in (Rei94, s.285) schreiben:
$E = 2\sum_{i=1}^{N_e/2} H_i + 2\sum_{i=1}^{N_e/2}\sum_{j=1}^{N_e/2}(V_{ij,ij} - \frac{1}{2}V_{ij,ji})$
Jede der Summe ergibt nur die Hälfte, was den Faktor 2 begründet. (Bei der Doppelsumme ergibt sich eigentlich 4, was sich aber gegen die 0.5 der vorherigen Formel kürzt). Da bei $V_{ij,ji}$ allerdings zwei Orbitale kombiniert werden, bleibt dieser Faktor dort allerdings erhalten.
Man kann dann in den Integralen statt mit den Spinorbitalen ψ mit den Ortsorbitalen ϕ arbeiten.

Nun berücksichtigt man die Darstellung der Molekülorbital als Atomorbitalen mit $\phi_i = \sum_{j=1}^{M} c_{ij}\chi_j$:
Das Integral einer Summe von Funktionen ist bekanntlich die Summe der Integrale der einzelnen Funktionen, setzt man also die Definition von ϕ_i in die Integrale ein, so ergibt sich:
$H_i = \sum_{\mu=1}^{M}\sum_{\nu=1}^{M} c_{i\mu}c_{i\nu}\int \chi_j(e)^*\mathbf{h}_e\chi_k(e)dV_e$
$V_{ij,kl} = \sum_{\mu=1}^{M}\sum_{\nu=1}^{M}\sum_{\rho=1}^{M}\sum_{\sigma=1}^{M} c_{i\mu}c_{j\nu}c_{k\rho}c_{l\sigma}\int\int \chi_\mu(e)^*\chi_\nu(f)^*\mathbf{V}_{ef}\chi_\rho(e)\chi_\sigma(f)dV_edV_f.$
(Die linken Parameter bestimmen die Orbitale, also $1 \leq i,j,k,l \leq N_e$). Schreibt man das große Integral als $(\mu\nu,\rho\sigma)$ und das kleine als $h_{\mu\nu}$ so ergibt das Einsetzen der Integrale in die Energiefunktion nach (Rei94, s.294) und (Low78, s. 314):
$E = \sum_{\mu=1}^{M}\sum_{\nu=1}^{M} P_{\mu\nu}\left[h_{\mu\nu} + \frac{1}{2}\sum_{\rho=1}^{M}\sum_{\sigma=1}^{M} P_{\rho\sigma}\left((\mu\nu,\rho\sigma) - \frac{1}{2}(\mu\sigma,\rho\nu)\right)\right].$
mit $P_{\mu\nu} = 2\sum_{i=1}^{N/2} c_{i\mu}^* c_{i\nu}$.
Der Inhalt der eckigen Klammer wird auch mit $F_{\mu\nu}$ als Element der Fockmatrix F bezeichnet.
Multipliziert man den zu Anfang gegebenen Ausdruck für die Energie $\int \Phi^*\mathbf{H}\Phi dV$ aus, ergibt sich eine ähnliche Form. Durch Analogieschluss könnte man nun schließen, dass es einen Fockoperator geben muss, der dem Hamiltonoperator ähnelt, aber nur auf jeweils ein Orbital wirkt, da die Energie als Summe aller Orbitale ausgedruckt ist.
Dies ist tatsächlich der Fall, und man kann, wie es in (Rei94, s. 277-281) getan wird, ihn mathematisch herleiten.
Wendet man ihn dann auf die Gleichungen an, ergeben sich nach (Rei94, s.

294) und (FIZ07, Kurs. 61407) die Roothaan-Hall-Gleichungen:
$\sum_{j=1}^{M} c_{lj}(F_{ij} - \epsilon_l S_{ij}) = 0.$
Hierbei bezeichnet l mit $1 \leq l \leq N_e/2$ ein beliebiges Molekülorbital, i mit $1 \leq i \leq M$ ein beliebiges Atomorbital, ϵ_l einen mit dem Molekülorbital verbunden Energiewert und S_{ij} das Überlappungsintegral zwischen Atomorbital i und j: $S_{ij} = \int \phi_i^* \phi_j dV$. Übersichtlicher kann man es in Form einer Matrix schreiben: $(F - \epsilon S)C = 0$.
Dies ist ein kompliziertes Gleichungssystem, das man nicht direkt lösen kann, da die Fockmatrix F erst durch die gesuchten Koeffizienten C bestimmt wird. Man geht daher iterativ vor, berechnet, ausgehend von einer festgelegten Matrix C, die Matrix F. Damit löst man das Gleichungssystem und erhält eine neue C Matrix.
Irgendwann ist diese neue Matrix mit der alten fast identisch und man kann annehmen, dass man eine ausreichend gute Näherung gefunden hat.
Es gibt einen Fall, wo das Lösen bei bekannter Fockmatrix relativ einfach ist, nämlich wenn $S = 1$ gilt. (1 = Einheitsmatrix im nxn-Matrixraum). Dann ergibt sich nämlich $FC = \epsilon C$.
Dies ist genau ein Eigenwertproblem einer symmetrischen Matrix, wobei ϵ der Vektor der Eigenwerte und C die Matrix der Eigenvektoren ist. Hierfür gibt es einigermaßen schnelle Algoritmen, der älteste ist der Jakobialgoritmus. (Coo74, s. 95-101).
Dort ist ebenfalls bewiesen, dass eine beidseitige Multiplikation mit $S^{-\frac{1}{2}}$ immer diesen Spezialfall liefert: (Coo74, s. 137-146).
Wie man nun genau die Atomorbitale und Integrale berechnet ist im Anhang erklärt.
Der Name dieses Verfahren ist „self consistent field“-Verfahren, kurz SCF, wobei unklar ist, ob diese Abkürzung nun für die Matrizen steht oder die Anfangsbuchstaben des Namens.

3.2 Geschwindigkeitsprobleme

Das größte Problem stellen die Integrale $(\mu\nu, \rho\sigma)$ dar. Jede dieser vier Parameter läuft von 1 bis M, es gíbt also M^4 von diesen Integralen. Zwar gibt es eine gewisse Symmetrie, wie in (vU80, s.60) angegeben, so dass 88% dieser Integrale gleich sind, trotzdem muss man noch $\frac{1}{8}M^4$ berechnen. Hat man durchschnittlich 3 Orbitale pro Atom, ergeben sich ungefähr $10N_k^4$ Integrale.
Zwar bezeichnet man in der theoretischen Informatik solche Laufzeiten als effizient, trotzdem ergeben sich schon für 10 Atome 100 000 Integrale und eine Laufzeit von etwa 1,5 Minuten (falls der Computer 1000 Integrale pro Sekunde berechnen kann).
Für 100 Atome ist die Zahl der Integrale schon zehntausendmal so hoch, also 1 Milliarde. Damit ist schon eine Laufzeit von 280 Stunden verbunden und ein Speicherverbrauch von ca. 3.8 GB (bei 4 Byte pro Integral).
Für 200 Atome ist diese Zahl nochmals 16 mal so hoch, und die Rechenzeit beträgt etwa ein *halbes Jahr* und der Speicherverbrauch 60.8 GB.
Dieses Verfahren ist also für große Moleküle nicht zu gebrauchen. Dies erkennt man auch an meinem Beispielsprogramm, wenn man versucht größere Moleküle zu berechnen.
Es gibt aber alternative Verfahren, beispielsweise ZDO, CNDO, NDDO, MNDO, INDO und MINDO, die sogenannten „neglect of differential overlap“-Verfahren (Rei94, s. 307f.).
Man ignoriert dabei die meisten Elektronenwechselwirkungsintegrale, wobei je nach Methode andere ignoriert werden.

Es gibt allerdings auch noch ein paar Genauigkeitsprobleme, die durch die getroffenen Näherungen begründet sind.
So steigt z.B.: auch die kinetische Energie der Kerne mit steigendem Abstand, da sie mehr „Bewegungsfreiraum“ haben. Das heißt die Energie ist bei größeren Abständen in Wirklichkeit höher als von dem Verfahren erkannt wird. Dies kann auch erklären, warum das Ergebnis meines Programmes bei Wasserstoff 0.085nm, beträgt und nicht wie in der Literatur 0.074 (Rei94, s.200). Bei schweren und damit langsameren Sauerstoffmolekül dagegen beträgt der berechnete Wert 123nm, was sehr nahme am experimentellen nach (Win07) von 0.120nm liegt.
Eine weitere schlechte Näherung ist es die Molekülorbitale aus als Atomorbitalen zusammengesetz zu betrachten. Die Elektronen können nämlich auch auf andere Energieebenen wechseln, als gemeinhin angenommen.
Ein Beispiel hierfür ist, dass in (vU80, s. 137) gesagt wird, für eine erfolgreiche

Winkelberechnung von Atomen mit p-Orbitalen, müsse man auch d-Orbital benutzen, da sonst planar und pyramidale Bindungswinkel falsch berechnet werden.
Ähnliches kann man auch im Programm sehen, wenn der Bindungswinkel von Wasser mit unter 98° angegeben wird, obwohl er in Wirklichkeit ca. 105° ist. (Die Tetraedermethode macht genau den umgekehrten Fehler, mit 109° ist der Winkel dort zu groß). Die ständige Benutzung von d-Orbitalen könnte das Programm also verbessen, allerdings ist die Berechnung von d-Orbitalen, wie man an den verwendeten Gleichungen sieht, sehr langsam und ihre Einbeziehung würde das sowieso schon sehr langsame Programm noch weiter verlangsamen, vielleicht sogar bis zur völligen Unbenutzbarkeit.

4 Weitere grundlegende Probleme

Es gibt weitere grundlegende Probleme bei solchen Berechnungen.
So wurde die gesamte Zeit davon ausgegangen, dass nur die Wechselwirkungen zwischen den Atomen des Moleküls zur Konformation beitragen. Wie man aber der Literatur, z.B.: (LNC94, s. 205) und (GY82, s. 151-175) entnehmen kann, tragen auch benachbarte Moleküle zur Struktur bei.
So können die Moleküle der Lösung, vor allem Wasser, in die Zwischenräume in einem großen Molekül eindringen und Wasserstoffbrückenbindungen oder auch nur van-der-Waals-Kräfte hervor rufen, die die Struktur ändern.
Bei Wasser sind vor allem die Wasserstoffbrückenbildungen wichtig, die größe Moleküle in fast beliebiger Zahl mit ihrer Umgebung - also dem Wasser - knüpfen können. Dies bewirkt eine zusätzlich Enthalpieänderung, die die Struktur maßgeben beieinflussen kann.
Insbesondere bei Proteinen sind auch sogenannte Chaperone bekannt, Proteine, die die Struktur anderer Proteine mitbestimmen.
Es ist auch eine gängige Modellvorstellung, dass sich auf diese Weise Prionenerkrankungen, wie z.B.: BSE, verbreiten, also, dass die „infizierten“ Prionen durch Kontakt mit anderen Prionen, diese wie Chaperone verformen und dadurch „anstecken“.
Das die Struktur sich nicht wiederherstellt, könnte man durch das oben erwähnte Multi-Minima-Problem erklären. Wie der Computer muss auch das Universum eine energetisch güsntige Konformation finden und wenn es dazu eine Art Geometrieoptimierung benutzt, was es eigentlich muss, da sich gewöhnlich Atome nur über sehr kurze Strecken beamen, kann es dabei durch aus auch zu einer solchen Situation kommen.

Ein weiteres Problem bei der Berechnung ist, die Vorstellung überhaupt Kernkoordinaten berechnen zu können.
Auf Grund des Heisenbergschen Unschärfeprinzips bewegen sich nämlich, wie alle quantentheoretischen Teilchen, die Atomkerne eines Moleküles und befinden sich nie an einer genau definierten Position.
Es gibt zwar Punkte mit höherer Aufenthaltswahrscheinlichkeit, versucht man aber diese zu beschreiben, so müssen man wie bei Elektoren Orbitale, oder ähnliches angeben, was aber eben keine Konformationen liefert.

Fazit

Wie man sehen kann, hat das Tetraederverfahren die in es gesteckten Erwartungen erfüllt.
Zwar sind die Ergebnisse unpräzise und zeigen sogar stellenweise drastische Abweichung, was ja aber auch erwartet wurde. Die Geschwindigkeit dagegen ist hervorragen und es ist somit auch kein Problem, große Moleküle mit zehntausenden von Atomen in wenigen Sekunden - wenn auch falsch - zu berechnen.
Das quantenchemische Verfahren dagegen enttäuscht. Die extrem langsame Geschwindigkeit habe ich zwar erwartet, doch bin ich davon ausgegangen, dass es dafür einen entsprechenden Genauigkeitsvorteil geben würde. Dieser war jedoch im eingesetzten Verfahren nicht zu erkennen, da die Abweichungen von den experimentellen Daten in manchen Molekülen sogar noch größer waren, als bei dem Tetraederalgorithmus.
Leider ist es hierbei unklar, ob dies von Programmierfehlern im entwickelten Computerprogramm herührt, oder von allgemeinen Schwäche der Verfahren. Vergleiche mit der Literatur zeigten jedoch, dass starke Abweichungen durchaus zu erwarten sind.

Das Ziel Grenzen in den Möglichkeiten der Berechnung aufzuzeigen, wurde jedoch zumindest in einer Richtung erfüllt.
So hat man deutlich gesehen, dass ein direktes Verfahren in komplexen Molekülen nicht zu zufriedenstellenden Ergebnissen führt und man auch komplexere Verfahren benutzen muss.
In der anderen Richtung hat man aber auch gesehen, dass ein solches komplexeres Verfahren, nicht unmittelbar aus der fundamentalen Quantenchemie gewonnen werden kann, da selbst bei der Benutzung von Nährungen ein solches Verfahren zu langsam ist.
Also ist hohe Präzision ebenso wie hohe Geschwindigkeit nicht zu erwarten.

Man kann auch abschließend sagen, dass man für die erfolgreiche Berechnung einer Konformation einen Algorithmus benötigt und entwickeln muss, der einerseits von chemischen Modellvorstellungen ausgehen sollte, die von tatsächlichen physikalischen Gegebenheiten abstrahieren, und anderenseits auch die Geschenisse im kleinsten Maßstab berücksichtigen sollte.
Dies zusammen ergibt, dass die Entwicklung sinnvoller Verfahren eine sehr komplizierte Gratwanderung darstellt.

A Atomorbitale und ihre Integrale

Bei der Berechnung von Orbitalen muss man einen mathematischen Ausdruck finden, der die Wellenfunktionen an einem Atom möglichst genau beschreibt. Es gibt nun gewisse Slater Typ Orbitale $sto = fr^{n-1}e^{-\varsigma * r}R_l^m(\vartheta, \theta)$, die der berechneten Form gut entsprechen. (Rei94, s.297).
f ist dabei nur ein Skalierungsfaktor, r der Abstand zum Kern, n, m, l die Quantenzahlen, e die Eulersche Zahl und $R(\vartheta, \theta)$ ein Winkelanteil.
ς ist ein Parameter der für jedes Atom verschieden ist. Eine Liste dieser Parameter mit Herleitung kann man in (ED63) finden.
Diese Funktion ist allerdings nur sehr schwer zu integrieren, weshalb man sie als Linearkombination von Gauss Typ Funktionen schreibt. Eine solche GTF sieht so aus:
$gtf = fx^u y^v z^w e^{-\alpha r^2}$.
f ist wieder ein Vorfaktor, x, y, z zeigen die Abhängigkeit von den Koordinaten an, r ist wieder der Kernabstand, $u + v + w$ ergibt die Hauptquantenzahl und α ist ein Parameter.
Integrale über dieser Funktion sind leicht zu berechnen, da r^2 in $x^2 + y^2 + z^2$ zerfällt und das Integral in drei kleine für jede Koordinate aufgeteilt werden kann.
Kompliziert ist die Bestimmung des Parameters α. Zum Glück wurde die Berechnung bereits durchgeführt und es gibt entsprechende Tabellen, beispielsweise in (Coo74, s.88f).
Man muss bei dieser Annäherung darauf achten nicht die Linearkombinationen zu verwechseln, zwar ist jedes Molekülorbital eine Linearkombination von Atomorbitalen, dass die Atomorbitale wiederum Linearkombination von Gaussfunktionen ist, hat damit aber nichts zu tun. Approximiert man eine STO durch n-GTFs, so sagt man, man benutzt eine STO-nG-Basis. Mein Programm arbeitet mit einer standardmäßigen STO-3G-Basis.

Die genaue Berechnung der Integrale ist auch in (Coo74, s.112-119) beschrieben und auch die Endergebnisse finden sich dort auf den Seiten 235 bis 239.
Ich wiederhole sie hier deshalb nicht nochmal, sondern mache nur ein paar Anmerkungen.
Es gibt dort nämlich eine ganze Reihe von Tippfehlern, die ich durch Vergleich mit (vU80, s. 58f.) herausfiltern musste. Im präziseren (vU80) sind allerdings leider nur die s-Orbitale beschrieben.
In (Coo74) ist auf Seite 235 der Normierungsfaktor angegeben, der das Über-

lappungsintegral des Orbitals mit sich selbst auf 1 bringen soll. Ein unbekannter Leser hat im Büchereiexemplar dort ein zusätzlich pi^{frac32} eingezeichnet. Eine einfache Überprüfung mit dem Mathematikprogramm Derive hat ergeben, dass dieser Faktor in der Tat mit eingebunden werden muss.
In den Büchern wird nun eine Hilfsfunktion $f_k(l_1, l_2, \overline{PA}_x, \overline{PB}_x)$ benutzt (siehe (Coo74, s. 236) und (vU80, s. 58)), allerdings ist die Definition in diesen Büchern unterschiedlich.
In (Coo74) findet man diese (Variablennamen sind an (vU80) angepasst):
$f_k(l_1, l_2, \overline{PA}_x, \overline{PB}_x) = \sum_{i=\max(0,i-l_2)}^{\min(k,l_1)} \binom{l_1}{i}\binom{l_2}{i-1}\overline{PA}_x^{l_1-i}\overline{PB}_x^{l_2+i-k}$
und in (vU80) findet man
$f_k(l_1, l_2, \overline{PA}_x, \overline{PB}_x) = \sum_{i=0}^{l_1}\sum_{j=0}^{l_2} {}^{(\text{und} i+j=k)}\overline{PA}_x^{l_1-i}\binom{l_1}{i}\overline{PB}_x^{l_2-j}\binom{l_2}{j}$

Die Funktion in (vU80) besteht also aus zwei Summen, durch die zusätzliche Bedingung $i + j = k$, lässen sich aber trotzdem als einzelne Summe darstellen, wenn man j durch $k - i$ ersetzt. Aus der zweiten Summe kann man die Bedingungen $0 \leq j \leq l_2$ gewinnen, also $0 \leq k - i \leq l_2 \Rightarrow -k \leq -i \leq l_2 - k \Rightarrow k \geq i \geq k - l_2$.
Kombiniert man dies mit den Bedingungen $l_1 \geq i \geq 0$, folgt $\min(k, l_1) \geq i \geq \max(0, k - l_2)$.
Dies sind gerade die Grenzen der Funktion in (Coo74). Allerdings unterscheidet sich diese Funktion auch dann noch von der in (vU80), wenn man wie eben beschrieben j durch $k - i$ ersetzt und die Grenzen anpasst. Der zweite Binomialkoeffizient müsste dann nämlich $\binom{m}{j-i}$ heißen und nicht $\binom{m}{j-1}$.
Dieser Unterschied lässt sich allerdings durch einen Tippfehler erklären, was sich dadurch bekräftigen lässt, das in der Ausgabe der Bücherei der unbekannte Leser auch diese Stelle als falsch markiert hat.
Diese Hilfsfunktion gibt übrigens gerade je nach Parameter die unterschiedliche Faktoren bei der ausmultiplizierten Form von $(x + \overline{PA})^l(x + \overline{PB})^m$ an.
Auf Seite 238 findet sich nun im Elektronwechselwirkungsintegral der Ausdruck $\frac{pi}{\gamma_1+\gamma_2}$, mit einem per Hand eingezeichnetem Wurzelzeichen. Diese Wurzel findet sich auch (vU80), weshalb ich davon ausgegangen bin, dass sie eingefügt werden muss.
Ein weiterer markierter Fehler ist die Definition von Tau auf Seite 239 von (Coo74). Hier habe ich mich auch für die Definition in (vU80) entschieden.

Davon, dass diese Fehlerkorrekturen angemessen waren, hat mich ein Beispiel in (Coo74) und die Überprüfung mit Derive überzeugt.
Letztere hat gezeigt, dass sowohl das Überlappungsintegral, wie auch das Inte-

gral der kinetischen Energie stimmen. Die weiteren Integrale konnte ich leider nicht damit testen, da Derive diese nicht mehr lösen konnte.
Im erwähnten Beispiel in (Coo74), verstreut auf den Seiten 85f. und 156 wird das hypothetische lineare BeH_2 mit einer STO-2G-Basis berechnet. Mein Programm kann dies nicht direkt nach berechnen (da es weder Beryllium noch eine STO-2G-Basis benutzt), aber ich habe mit einer stellenweisen modifizierten Version das Molekül durchgerechnet.
Alle der ca. 100 Integrale waren korrekt, allerdings habe ich dabei noch einen Fehler im Buch gefunden. Der α-Parameter einer GTF war um exakt den Faktor 10 zu groß, was sowohl bei einem Nachrechnen der Integrale mit Derive, wie auch mit Nachrechnen der Linearkombinationserstellung herauskam. Wahrscheinlich handelt es sich also dieser Stelle, um ein Tippfehler und nicht bei allen 100 Integralen.

Ich weiche allerdings in meinem Programm noch etwas weiter von der Berechnung in (Coo74) ab. Ich habe die Integrale in einen positionsunabhängigen und einen positionsabhängigen Teil zerlegt, wie man im Programmcode erkennen kann.
Alle positionsunabhängigen-Teile werden einmal beim Programmstart berechnet und bei der eigentlichen Integralberechnung nur gelesen. Hierdurch wurde das Programm ein wenig schneller, der Effekt war aber nur minimal.
Außerdem zerlege ich die STOs nicht explizit GTFs, sondern nur implizit bei der Berechnung der Integrale, wodurch der Quellcode einwenig übersichtlicher ist.

B Mathematische Symbole

Hier sind die im Hauptteil verwendeten mathematischen Symbole aufgelistet:

C	Koeffizientenmatrix
c_{ij}	Koeffizienten zur Linearkombination von Atomorbitalen
dV	Volumenelement
$d\sigma$	Spinelement
ϵ	Molekülorbitalenergieeigenwertvektor
e	Elementarladung oder Index eines Elektrons oder Orbitals
E	Energie
E_{ges}	Energie des gesamten Moleküls
E_e	Energie der Elektronen
E_k	Energie der Kerne
F	Fockmatrix
F_ij	Element der Fockmatrix
$\bar{h}$	Planckenergie geteilt durch 2π
i	Index
j	Index
k	Index oder Kern
l	Index
$\mathbf{H}$	Hamiltonoperator
$\mathbf{H}_e$	Hamiltonoperator, beschränkt auf Elektronen
H_i	Ein-Elektronen Energie des i-ten Molekülorbitals
$\mathbf{T}$	allgemein kinetische Energieoperator
T_k	kinetische Energie der Kerne
$\mathbf{T}_k$	kinetische Kernenergieoperator
$\mathbf{T}_e$	kinetische Elektronenenergieoperator
$\mathbf{V}_{ij}$	allgemein Wechselwirkungsenergieoperator zwischen Teilchen i und j
$\mathbf{V}_{ij,kl}$	Wechselwirkungsoperator zwischen Elektronen in Orbitalen i, j und $, k, l$
$\mathbf{V}_{kk}$	Kern-Kern-Abstoßungsenergieoperator
V_{kk}	Kern-Kern-Abstoßungsenergie
$\mathbf{V}_{ee}$	Elektron-Elektron-Abstoßungsenergieoperator
$\mathbf{V}_{ek}$	Elektron-Kern-Anziehungsenergieoperator
m_i	Masse eines beliebigen Teilchens
N_e	Anzahl aller Elektronen
M	Anzahl aller Atomorbitale

r_{ij}	Abstand zwischen den Teilchen i und j
S	Überlappungsmatrix
S_{ij}	Element der Überlappungsmatrix
$\vec{r}_1$ bis $\vec{r}_4$	Eckpunkte eines Tetraeders
x_i	Koordinaten eines Teilchens
Z_i	Ladungszahl eines Teilchens (-1 bei Elektronen)
$\vec{z}$	Schwerpunkt eines Tetraeders
α, β	Spineigenschaft
Ψ	Wellenfunktion
Ψ_e	Wellenfunktion aller Elektronen
ψ_e	Wellenfunktion/Spinorbital eines Elektrons.
ϕ_e	Ortsorbital eines Elektrons.
∇^2	Laplaceoperator
χ_i	Atomorbital i (eindeutig im gesamten Molekül).
$(\mu\nu, \rho\sigma)$	Wie $V_{\mu\nu,\rho\sigma}$

C Bilder

D Programmübersicht

Das Programm ist in der Programmiersprache Pascal programmiert und zwar mit der kostenlosen Open-Source-Entwicklungsumgebung Lazarus.
Bei der Benutzung muss man, das Molekül, das man berechnen lassen möchte in der oberen Bildschirmhälfte als Lewisformel eingeben. Atome erzeugen und löschen kann man dabei per Rechtsklick. Das Programm unterstützt nur H,C,N und O, die häufigsten Atome in organischen Molekülen.
Man kann jeder Zeit in der mittleren Fensterfläche die 3D-Ansicht aktivieren, dann sieht man die aktuell berechnete Konformation und kann sie durch Anklicken bewegen.
Programmiertechnisch wird dieses allgemeine Verhalten von der Datei gui.pas gesteuert. Die Lewis-Formel wird in der Datei lewisRenderer.pas verwaltet, die 3D-Ausgabe über OpenGL in oglDrawing.pas und atomicDrawing.pas.
Die Tetraedergeometrieberechnung wird automatisch bei Veränderung der Lewisformel durchgeführt, die SCF-Optimierung wird dabei durch den Button „Geometrieoptimierung“ aktiviert. Diese ist allerdings sehr langsam und funktioniert bekanntlich nur für sehr kleine Moleküle.
Diese Funktionen sind im Quellcode in der Datei molecules.pas, die die generellen Molekülberechnungen enthält.
Dort stehen auch die Routinen für das Laden und Abspeichern von Molekülen.
Die Berechnung von Orbitalen und Orbitalintegralen findet man in den Dateien gtf.pas und stosim.pas. In stosim.pas werden auch die vorberechneten Integrale gespeichert.

Es gibt zwei ausführbare Dateien folgind_normal und folding_schnell. Letzteres sollte ein wenig schneller sein, weil alle Fehlertest wie die Überprüfung auf einen Zugriff auf nicht vorhandene Elemente fehlt.

E Quellcode

E.1 gui.pas

```
{Dieser Programmteil steuert die Interaktion mit dem Benutzer.
 Hierzu werden seine Einnahmen über eine LCL-Form entgegen genommen,
 an die anderen Bestandteile des Programmes weitergereicht und
 die Ergebnisse ausgegeben.
}

unit gui;

{$mode objfpc}{$H+}

interface

uses
  windows,Classes, SysUtils, LResources, Forms, Controls, Graphics, Dialogs,
  w32initopengl, ExtCtrls,oglDrawing,atomicdrawing, Buttons,atoms,molecules,
  StdCtrls,lewisrenderer,extmath;

type

  { TForm1 }

  TForm1 = class(TForm)
    //Verwendete Designelemente
    colorbox: TComboBox;
    energyCalculateBtn: TButton;
    Button3: TButton;
    Label7: TLabel;
    molSaveBtn: TButton;
    geooptbtn: TButton;
    molLoadBtn: TButton;
    modelbox: TComboBox;
    atomNr: TEdit;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Label4: TLabel;
    Label6: TLabel;
    OpenDialog1: TOpenDialog;
    optimizeStepLabel: TLabel;
    loadOGL1: TButton;
    PaintBox1: TPaintBox;
```

```
    SaveDialog1: TSaveDialog;
    Splitter2: TSplitter;
    viewBarRotX: TScrollBar;
    viewBarRotY: TScrollBar;
    viewBarDistance: TScrollBar;
    viewDistance: TLabel;
    viewRotX: TLabel;
    viewRotY: TLabel;
    Label5: TLabel;
    label9: TLabel;
    loadOGLBtn: TButton;
    status: TMemo;
    Panel1: TPanel;
    Panel2: TPanel;
    Panel3: TPanel;
    optimizeStepBar: TScrollBar;
    Splitter1: TSplitter;
    Timer1: TTimer;
    //Diese Funktionen werden automatisch bei bestimmten Ereignissen
    //aufgerufen
    procedure energyCalculateBtnClick(Sender: TObject);
    procedure Button3Click(Sender: TObject);
    procedure loadOGLBtnClick(Sender: TObject);
    procedure molSaveBtnClick(Sender: TObject);
    procedure geooptbtnClick(Sender: TObject);
    procedure molLoadBtnClick(Sender: TObject);
    procedure modelboxChange(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure optimizeStepBarScroll(Sender: TObject; ScrollCode: TScrollCode;
      var ScrollPos: Integer);
    procedure PaintBox1MouseDown(Sender: TOBject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
    procedure PaintBox1MouseMove(Sender: TObject; Shift: TShiftState; X,
      Y: Integer);
    procedure PaintBox1Paint(Sender: TObject);
    procedure Panel1MouseDown(Sender: TOBject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
    procedure Panel1MouseMove(Sender: TObject; Shift: TShiftState; X, Y: Integer
      );
    procedure Panel1Resize(Sender: TObject);
    procedure viewBarDistanceScroll(Sender: TObject; ScrollCode: TScrollCode;
      var ScrollPos: Integer);
    procedure Timer1Timer(Sender: TObject);
  private
```

```
    { private declarations }
    cancelGeoOptimizing:boolean;
    procedure optimizingProgress(sender: tobject;energy:number;steps: longint;
                                 var cancel: boolean);
  public
    { public declarations }
    //Eigentliche Berechnungsklassen, definiert in anderen Dateien
    oglW32: TOGLW32Base;    //oglDrawing.pas
    oglRenderer: TOGLRenderer; //oglDrawing.pas
    atomRenderer: TAtomRenderer; //atomicDrawing.pas
    lewisRenderer: TLewisRenderer;//lewisRenderer.pas

    currentMolecule:TMolecule; //molecules.pas

    //Position im 3D-Raum
    z,ry,rx: single; //abstand,rotation um x,y
    cx,cy:longint;//klickpos
    cz,cry,crx: single; //abstand,rotation um x,y beim klicken
    //Ausgaben an den Benutzer umwandeln
    procedure viewChanged;
    procedure publicLog(s:string);
    function lenToUser(n: number): string;
    function energyToUser(n: number): string;
    function angleToUser(n: number): string;

  end;

var
  Form1: TForm1;


implementation

uses gl,gtf,math;
{ TForm1 }


//Start
procedure TForm1.FormCreate(Sender: TObject);
begin
  z:=5;
  oglW32:=nil;
  currentMolecule:=TMolecule.create;
  lewisRenderer:=TLewisRenderer.create;
  lewisRenderer.useMolecule(currentMolecule);
```

```
  lewisRenderer.paintBox:=PaintBox1;
end;

//Ende
procedure TForm1.FormDestroy(Sender: TObject);
begin
  if oglW32<>nil then begin
    atomRenderer.free;
    oglRenderer.free;
    oglW32.free;
  end;
  if lewisRenderer<>nil then
    lewisRenderer.free;
end;

//Rückmeldung an den Benutzer nach Änderung
procedure TForm1.modelboxChange(Sender: TObject);
begin
  if (atomRenderer<>nil) then begin
    atomRenderer.useModel(modelbox.ItemIndex,colorBox.itemindex);
    if modelbox.ItemIndex<>atomRenderer.currentModel then begin
      case modelbox.ItemIndex of
        0: z-=5;
        1: z+=5;
      end;
    end;
    viewChanged;
  end;
end;

procedure TForm1.optimizeStepBarScroll(Sender: TObject;
  ScrollCode: TScrollCode; var ScrollPos: Integer);
begin
  optimizeStepLabel.Caption:=inttostr(optimizeStepBar.Position);
end;

procedure TForm1.optimizingProgress(sender: tobject; energy: number; steps: longint;
  var cancel: boolean);
begin
  Application.ProcessMessages;
  publicLog('Noch maximal '+inttostr(steps)+' Schritte, aktuelle Energie: '+
            energyToUser(energy));
  cancel:=cancelGeoOptimizing;
end;
```

```
procedure TForm1.viewChanged;
begin
  viewBarDistance.Position:=trunc(−z);
  viewDistance.Caption:=inttostr(−trunc(z));
  if rx<−180 then rx:=rx+360;
  if rx>180 then rx:=rx−360;
  viewRotX.Caption:=inttostr(trunc(rx))+'°';
  viewBarRotX.Position:=trunc(rx);
  if ry<−180 then ry:=ry+360;
  if ry>180 then ry:=ry−360;
  viewRotY.Caption:=inttostr(trunc(ry))+'°';
  viewBarRotY.Position:=trunc(ry);
end;

//Änderungen der Lewisformel erkennen
procedure TForm1.PaintBox1MouseDown(Sender: TOBject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if lewisRenderer=nil then exit;
  lewisRenderer.userMouseDown(x,y,button);
end;

procedure TForm1.PaintBox1MouseMove(Sender: TObject; Shift: TShiftState; X,
  Y: Integer);
begin
  if lewisRenderer=nil then exit;
  lewisRenderer.userMouseMove(x,y,shift);
end;

procedure TForm1.PaintBox1Paint(Sender: TObject);
begin
  if lewisRenderer<>nil then
    lewisRenderer.render(PaintBox1.Canvas,PaintBox1.ClientRect);
end;

//Mausbewegung im 3D-Raum erkennen
//Rotation um x und y-Achse je nach Mausbewegung
procedure TForm1.Panel1MouseDown(Sender: TOBject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  cx:=x;
  cy:=y;
  cz:=z;
  crx:=rx;
  cry:=ry;
```

```
end;

procedure TForm1.Panel1MouseMove(Sender: TObject; Shift: TShiftState; X,
  Y: Integer);
begin
  if ssRight in shift then begin
    z:=cz+(cy-y)/50;
    viewChanged;
  end else if ssleft in shift then begin                                    230
    rx:=crx+y-cy;
    ry:=cry+x-cx;
    viewChanged;
  end;
end;

procedure TForm1.Panel1Resize(Sender: TObject);
begin
  if oglRenderer<>nil then
    oglRenderer.resize(panel1.width,panel1.height);                         240
end;

procedure TForm1.viewBarDistanceScroll(Sender: TObject; ScrollCode: TScrollCode;
  var ScrollPos: Integer);
begin
  z:=-viewBarDistance.Position;
  rx:=viewBarRotX.Position;
  ry:=viewBarRotY.Position;
  viewChanged;
end;                                                                        250

//Periodisch zeichnen
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  if (oglW32<>nil)and (atomRenderer<>nil) and (currentMolecule<>nil) then begin
    glMatrixMode(GL_modelview);
    glLoadIdentity;
    glTranslatef(0,0,z);
    glRotatef(ry,0,1,0);
    glRotatef(rx,1,0,0);                                                    260
    glColor4f(1,1,1,0.5);
    atomRenderer.draw(currentMolecule);
    oglW32.endScene;
  end;
end;
```

```
//Informationsausgaben
procedure TForm1.publicLog(s: string);
begin
  status.lines.add(FormatDateTime('hh:mm:ss.zzzz',time)+': '+s);
end;

function TForm1.lenToUser(n: number): string;
begin
  if abs(n)<1e-5 then exit('0 nm');
  result:=format('%2.3f',[n*0.05291772])+' nm';
end;

function TForm1.energyToUser(n: number): string;
begin
  if abs(n)<1e-5 then exit('0 kJ/mol');
  result:=format('%2.5f',[n*(4.3594e-18)*(6.02214e23)/1000])+' kJ/mol';
end;

function TForm1.angleToUser(n: number): string;
begin
  if abs(n)<1e-5 then exit('0°');
  result:=format('%2.1f',[180*n/pi])+'°';
end;


//Energieberechnung aufrufen
procedure TForm1.energyCalculateBtnClick(Sender: TObject);
var t:cardinal;
begin
  currentMolecule.initMOSCFEnergyCalculation;
  publicLog('Energieberechnung gestartet');
  publicLog('Energieberechnung beendet: '+
            energyToUser(currentMolecule.calculateEnergy()));
end;

//Informationen über ausgewähltes Atom anzeigen
procedure TForm1.Button3Click(Sender: TObject);
var i,j:longint;
    a: tatom;
    s:string;
    b:array[1..4] of TVector3n;
begin
  a:=tatom(currentMolecule.atoms[strtoint(atomNr.text)]);
  publicLog('Atom '+atomNr.text+' '+ATOM_NAMES[a.typ]+' hat '+
```

```
            inttostr(a.james)+' unterschiedliche Bindungen: ');
  //Bindungslängen
  for i:=1 to a.james do begin
    b[i]:=vecsub(a.p,a.bond[i].a.p);
    publicLog(inttostr(i)+': '+ATOM_NAMES[a.bond[i].a.typ]+' Länge: '+
              lenToUser(sqrt(veclensqr(b[i])))
              );
  end;
  //Bindungswinkel (nach Formelsammlung)                                    320
  for i:=1 to a.james do begin
    s:='Winkel '+inttostr(i)+': ';
    for j:=1 to a.james do
      s:=s+angleToUser(arccos(((b[i]['x']*b[j]['x']+b[i]['y']*b[j]['y']+
                               b[i]['z']*b[j]['z']))/
                               sqrt(veclensqr(b[i])*veclensqr(b[j])))+#9;
    publicLog(s);
  end;
end;
                                                                            330
//OpenGL laden
procedure TForm1.loadOGLBtnClick(Sender: TObject);
//Licht position
const
  mat_specular : Array[0..3] of GlFloat = (1.0, 1.0, 1.0, 1.0);
  mat_shininess : Array[0..0] of GlFloat = (50.0);
  mat_ambient : Array[0..3] of GlFloat = (1, 1, 1, 1.0);
  mat_diffuse  : Array[0..3] of GlFloat = (1, 1, 1, 1.0);

  light_position : Array[0..3] of GlFloat = (5.0, 5.0, 10.0, 1.0);          340
  light_ambient : Array[0..3] of GlFloat = (0.1, 0.1, 0.1, 1.0);
  light_diffuse  : Array[0..3] of GlFloat = (0.8, 0.8, 0.8, 1.0);
const
   att: GLfloat=0.5;
begin
  //Objekte erzeugen
  oglW32:=TOGLW32Base.create(panel1.handle);
  oglRenderer:=TOGLRenderer.Create;
  oglRenderer.init();
  oglRenderer.resize(panel1.width,panel1.Height);                           350
  atomRenderer:=TAtomRenderer.create(oglRenderer);
  atomRenderer.useModel(modelbox.ItemIndex);
  z:=-5;
  //Standardopenglbeleuchtung
  glEnable(GL_LIGHTING);
  glEnable(GL_LIGHT0);
```

```
  glEnable(GL_COLOR_MATERIAL);
  glLoadIdentity;

  glMaterialfv(GL_FRONT, GL_SPECULAR, @mat_specular[0]);
  glMaterialfv(GL_FRONT, GL_SHININESS, @mat_shininess[0]);
  glMaterialfv(GL_FRONT, GL_AMBIENT, @mat_ambient[0]);
  glMaterialfv(GL_FRONT, GL_DIFFUSE, @mat_diffuse[0]);

  glLightfv(GL_LIGHT0, GL_AMBIENT, @light_ambient[0]);
  glLightfv(GL_LIGHT0, GL_DIFFUSE, @light_diffuse[0]);
  glLightfv(GL_LIGHT0, GL_POSITION, @light_position[0]);

  if currentMolecule<>nil then
    panel1.caption:='Zeichne Molekül...'
   else
    panel1.caption:='Kein Molekül geladen';
  loadOGLBtn.Visible:=false;

  publicLog('3D Renderer geladen');
end;

//Geometrieoptimierung aufrufen oder abbrechen
procedure TForm1.geooptbtnClick(Sender: TObject);
var c:char;
    i:longint;
    s:string;
begin
  if geooptbtn.tag=0 then begin
    //Möglichkeit zum Abbrechen bieten
    geooptbtn.tag:=1;
    geooptbtn.Caption:='Optimierung abbrechen';
    molSaveBtn.Enabled:=false;
    molLoadBtn.Enabled:=false;
    energyCalculateBtn.Enabled:=false;
    loadOGLBtn.Enabled:=false;

    //Aufrufen
    cancelGeoOptimizing:=false;
    currentMolecule.initMOSCFEnergyCalculation;
    publicLog('Geometrieoptimierung gestartet');
    currentMolecule.GeoOptimize(optimizeStepBar.position,@optimizingProgress);
    //Fertig, Ergebnisse ausgeben
    if cancelGeoOptimizing then
      publicLog('Geometrieoptimierung vom Benutzer abgebrochen')
    else publicLog('Geometrieoptimierung beendet: ');
```

```
    for i:=0 to currentMolecule.atoms.count-1 do begin
      s:='Atom '+inttostr(i)+': '+
         ATOM_NAMES[tatom(currentMolecule.atoms[i]).typ]+': ';
      for c:='x' to 'z' do
        s:=s+c+': '+lenToUser(tatom(currentMolecule.atoms[i]).p[c])+' ';
      publicLog(s);
    end;
    //Abbrechen verhindern
    molSaveBtn.Enabled:=true;
    molLoadBtn.Enabled:=true;
    energyCalculateBtn.Enabled:=true;
    loadOGLBtn.Enabled:=true;
    geooptbtn.Tag:=0;
    geooptbtn.caption:='Geometrie optimieren';
  end else cancelGeoOptimizing:=true; //Abbrechen
end;

//Dateibehandlung
procedure TForm1.molSaveBtnClick(Sender: TObject);
begin
  if SaveDialog1.Execute then currentMolecule.saveToFile(saveDialog1.filename);
end;

procedure TForm1.molLoadBtnClick(Sender: TObject);
begin
  if openDialog1.Execute then begin
    currentMolecule.loadfromFile(openDialog1.filename);
    publicLog('Molekül geladen');
    PaintBox1.Refresh;
  end;
end;



initialization
  {$I gui.lrs}

end.
```

E.2 oglDrawing.pas

```
{Dieser Programmteil dient zum Anzeigen von 3D-Objekten über
 die Grafikschnittstelle OpenGL
}
unit oglDrawing;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils,molecules,graphics,gl,glu;

type

{ TOGLObject }

//Verwaltet ein einfaches 3D-Objekt
TOGLObject=class
private
  _color: T4fArray;
  list: GLuint;
public
  constructor create();
  procedure setColor(c: T4fArray);overload;
  procedure setColor(c: TColor);overload;
  property color:T4fArray read _color write setcolor;
end;

{ TOGLGLUObject }

//Bieten einen übersichtlichen Zugriff auf die von OpenGLU automatisch erzeugten
//primitiven Objekte (Kugel und Zylinder)
TOGLGLUObject=class(TOGLObject)
private
  quadric: PGLUquadric;
public
  constructor create();
  destructor destroy;override;

  procedure setToSphere(radius:GLdouble;slices,stacks:GLint);
  procedure setToCylinder(baseRadius,topRadius,height:GLdouble;slices,stacks:GLint);
  procedure setToCylinder(radius,height:GLdouble;slices,stacks:GLint);
end;
```

```
{ TOGLRenderer }

//Verwaltet den Grafikprozess
TOGLRenderer=class
  procedure init();
  procedure resize(w,h:longint);
  procedure draw(obj: TOGLObject);
  procedure drawAt(obj: TOGLObject;x, y, z: GLfloat);
end;
var next_display_list: longint=1;
implementation

{ TOGLRenderer }
//Initialisiert Standardeinstellungen (siehe www.delphigl.com oder ein
//OpenGL Handbuch)
procedure TOGLRenderer.init();
begin
  glShadeModel(GL_SMOOTH);
  glClearColor(0.0, 0.0, 0.0, 0.5);
  glClearDepth(1.0);
  glEnable(GL_DEPTH_TEST);
  glDepthFunc(GL_LEQUAL);
  glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);
end;

//Passt die Perspektive an die Ausgabefläche an
procedure TOGLRenderer.resize(w, h: longint);
begin
  glViewport(0, 0, w, h);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(45,w/h,0.1,100);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
end;

//Zeichnet ein Objekt
procedure TOGLRenderer.draw(obj: TOGLObject);
begin
  glCallList(obj.list);
end;

//Zeichnet ein Objekt an einer bestimmten Position
procedure TOGLRenderer.drawAt(obj: TOGLObject; x, y, z: GLfloat);
```

```
begin
  glTranslatef(x,y,z);
  draw(obj);
  glTranslatef(−x,−y,−z);
end;


{ TOGLGLUObject }

constructor TOGLGLUObject.create();
begin
  inherited;
  quadric:=gluNewQuadric();
  list:=next_display_list;
  inc(next_display_list);
end;

destructor TOGLGLUObject.destroy;
begin
  gluDeleteQuadric(quadric);
  inherited;
end;

//Lässt eine Kugel berechnen
procedure TOGLGLUObject.setToSphere(radius: GLdouble; slices, stacks: GLint);
begin
  glNewList(list,GL_COMPILE);
  glColor4fv(@_color);
  gluSphere(quadric,radius,slices,stacks);
  glEndList;
end;

//Lässt einen Zylinder berechnen
procedure TOGLGLUObject.setToCylinder(baseRadius, topRadius, height: GLdouble;
  slices, stacks: GLint);
begin
  glNewList(list,GL_COMPILE);
  glColor4fv(@_color);
  gluCylinder(quadric,baseRadius,topRadius,height,slices,stacks);
  glEndList;
end;

procedure TOGLGLUObject.setToCylinder(radius, height: GLdouble; slices,
  stacks: GLint);
begin
```

```
  setToCylinder(radius,radius,height,slices,stacks);
end;

{ TOGLObject }

constructor TOGLObject.create();
begin                                                    140
  color[0]:=1;
  color[1]:=1;
  color[2]:=1;
  color[3]:=1;
end;

procedure TOGLObject.setColor(c: T4fArray);
begin
  _color:=c;
end;                                                     150

//Delphifarbe zu OGL-Farbe
procedure TOGLObject.setColor(c: TColor);
const R_MASK=$0000FF;
      G_MASK=$00FF00;
      B_MASK=$FF0000;
begin
  _color[0]:=(c and R_MASK)/R_MASK;
  _color[1]:=(c and G_MASK)/G_MASK;
  _color[2]:=(c and B_MASK)/B_MASK;                      160
end;

end.
```

E.3 atomicDrawing.pas

```
{Dieser Programmteil benutzt oglDrawing um ein bestimmtes Molekül über OpenGl
 3-dimensional auszugeben.}
unit atomicDrawing;

{$mode objfpc}{$H+}
interface

uses
  Classes, SysUtils,ogldrawing,atoms,molecules,Graphics,extmath;

type
{ TAtomRenderer }

//Zeichenklasse
TAtomRenderer=class
  private
    ogl: TOGLRenderer;
    atoms: array[1..8] of TOGLGLUObject; //Speichert für jedes Atom eine Kugel
    link: TOGLGLUObject; //Speichert einen Zylinder als Bindungsstab
    usesModel: longint; //Verwendetes Modell (0: Kalotten, 1: Kugel/Stab)
  public
    colorScheme: longint;
    constructor create(oglRenderer: TOGLRenderer);
    procedure draw(mol: TMolecule);
    destructor destroy;override;
    //Legt das Modell (0: Kalotten, 1: Kugel/Stab) und
    //das Farbschema (0: normal, 1: Drucken) fest
    procedure useModel(const m: longint;const c:longint=-1);
    property currentModel: longint read usesModel write useModel;
end;
      //Farben für Atome
const atomColors: array[0..1,-1..8] of TColor=
      ((clBlack,clSilver,clSilver,0,0,0,0,clGray,clBlue,clRed),
      (clWhite,clGray,clSilver,0,0,0,0,clGray,clBlue,clRed));
      //van der Waals-Radius (Radius der Kalottenmodellkugeln) (für Kalottenmodell)
      vanderWaalsRadius: array[1..8] of number=
                  (2.268,0,  0,0,0,3.21,2.93,2.89); //in a0}
      //Radius einer Bindung (für KS.modell)
      bondWidth: number=0.2;
      //Radius eines Kerns (für KS.modell)
      kernRadius: number=0.5;
implementation
```

```
uses gl,glu,math;
{ TAtomRenderer }

constructor TAtomRenderer.create(oglRenderer: TOGLRenderer);
var i:integer;
begin
  ogl:=oglRenderer;                                                    50
  //Speicherplatz für Atome reservieren
  FillChar(atoms,sizeof(atoms),0);
  atoms[AT_H]:=TOGLGLUObject.Create;
  atoms[AT_C]:=TOGLGLUObject.Create;
  atoms[AT_N]:=TOGLGLUObject.Create;
  atoms[AT_O]:=TOGLGLUObject.Create;
  link:=TOGLGLUObject.create();
  //Modell aktivieren
  useModel(0);
end;                                                                   60

procedure TAtomRenderer.draw(mol: TMolecule);
const R_MASK=$0000FF;
      G_MASK=$00FF00;
      B_MASK=$FF0000;
var i,j,k:integer;
    typ:longint;
    atom:TAtom;
    len:float;
    p: TVector3f;                                                      70
begin
  //Hintergrund leeren
  glClearColor((atomColors[colorScheme][-1] and R_MASK)/R_MASK,
               (atomColors[colorScheme][-1] and G_MASK)/G_MASK,
               (atomColors[colorScheme][-1] and B_MASK)/B_MASK,0);
  glClear(GL_COLOR_BUFFER_BIT or GL_DEPTH_BUFFER_BIT);
  //Molekülmittelpunkt auswählen
  glTranslatef(-mol.center['x'],-mol.center['y'],-mol.center['z']);
  //Atome der Reihe nach zeichnen
  for i:=0 to mol.atoms.count-1 do begin                               80
    atom:=TAtom(mol.atoms[i]);
    typ:=atom.typ;
    p:=vec2f(atom.p);
    case usesModel of
    0: //Einfache Kugel beim Kalottenmodell
       ogl.drawAt(atoms[typ],p['x'],p['y'],p['z']);
    1: begin //Lage der Bindungsstäbe muss berechnet werden
      glTranslatef(p['x'],p['y'],p['z']);
```

```
for j:=1 to atom.james do
  if dword(atom.bond[j].a)>dword(atom) then //Bindungssortierung
  begin //(Trick, damit jede Bindung nur einmal gezeichnet wird)
    glPushMatrix; //Position sichern
    len:=sqrt(veclensqr(vecsub(vec2f(atom.bond[j].a.p),p))); //Bindungslänge
    {       y     z              OpenGL Koordinatensystem
            y   z   *
            0 z
    xxxxxxx000xxxxxxx
          z 0
        z   y
      z     y
    }
    //Andere Atomkernkoordinaten von globalen XYZ-KO in ein um diesen Kern
    //zentriertes Kugel-KO-System transformieren und den Stab entsprechend
    //den Kugelkoordinaten rotieren
    if not iszero(atom.bond[j].a.p['x']-p['x']) then
      glRotatef(arctan2(atom.bond[j].a.p['y']-atom.p['y'],
                        atom.bond[j].a.p['x']-p['x'])*180/pi,0,0,1)
     else
      glRotatef(90*sign(atom.bond[j].a.p['y']-atom.p['y']),0,0,1);
    glRotatef(arccos(max(-1,min(1,(atom.bond[j].a.p['z']-atom.p['z'])/len)))*
              180/pi,0,1,0);
    glScalef(1.1-atom.bond[j].count*0.1,1.1-atom.bond[j].count*0.1,len);
    //Ein Zylinder pro Bindung
    case atom.bond[j].count of
      1: ogl.draw(link);
      2: begin //Nebeneinander, leicht verschoben
           glTranslatef(bondWidth,0,0);
           ogl.draw(link);
           glTranslatef(-2*bondWidth,0,0);
           ogl.draw(link);
         end;
      3: begin //Verschoben in Dreiecksstruktur
           glTranslatef(bondWidth,bondWidth*0.7,0);
           ogl.draw(link);
           glTranslatef(-2*bondWidth,0,0);
           ogl.draw(link);
           glTranslatef(bondWidth,-bondWidth*1.7,0);
           ogl.draw(link);
         end;
    end;
    glPopMatrix;
  end;
//Kugel zeichnen
```

```
      ogl.draw(atoms[typ]);
      glTranslatef(-p['x'],-p['y'],-p['z']);
      end;
    end;
  end;
  glTranslatef(mol.center['x'],mol.center['y'],mol.center['z']);
end;

destructor TAtomRenderer.destroy;
var i:integer;
begin
  for i:=low(atoms) to high(atoms) do
    if atoms[i]<>nil then
      atoms[i].free;
  inherited destroy;
end;

//Model laden
procedure TAtomRenderer.useModel(const m,c: longint);
var i:integer;
begin
  if c<>-1 then colorScheme:=c; //Farbschema wechseln
  usesModel:=m;
  case m of
    0: for i:=low(atoms) to high(atoms) do //Kalottenmodell
         if atoms[i]<>nil then begin
           atoms[i].SetColor(atomColors[colorScheme,i]);  //Farbe bestimmen
           atoms[i].setToSphere(vanderWaalsRadius[i],50,50); //Kugel mit Radius
         end;
     1: begin //Kugel-Stab-modell
       //Zylinder erzeugen
       link.setColor(atomColors[colorScheme][0]);
       link.setToCylinder(bondWidth,1,10,10);
       //Kugeln erzeugen
       for i:=low(atoms) to high(atoms) do
         if atoms[i]<>nil then begin
           atoms[i].SetColor(atomColors[colorScheme,i]);
           atoms[i].setToSphere(kernRadius,15,15);
         end;
     end;
  end;
end;
end.
```

E.4 lewisRenderer.pas

```
{ Dieser Programmteil zeichnet eine Lewisformel, die der Benutzer
  beliebig ändern kann}
unit lewisRenderer;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils,Controls,menus,graphics,atoms,molecules,Extctrls;

type
  { TLewisRenderer }

  TLewisRenderer=class
    status: (sNormal,sAddBond); //Reaktion auf Benutzer (sAddBond verbindet Atome)
    molecule: TMolecule;        //Angezeigtes Molekül
    symHeight,symWidth: longint; //Größe eines Symbols

    selected: tatom;            //Ausgewähltes Atom
    mx,my,ox,oy:longint;        //Mausposition
    atomPopup: TPopupMenu; //Menü zur Benutzerinteraktion
    //Ereignisse des Menüs
    procedure itemAddClick(Sender: TObject);
    procedure itemBondAddClick(Sender: TObject);
    procedure itemBondDeleteClick(Sender: TObject);
    procedure itemDeleteClick(Sender: TObject);
  public
    defX, defY:longint;  //Nullpunkt
    paintBox: tpaintBox; //Ausgabefläche
    constructor create;
    destructor destroy;override;
    procedure useMolecule(mol: TMolecule);
    procedure rearrange();//Ordnet die Atome intelligen (unbenutzt)
    procedure render(canvas: TCanvas;rect:TRect); //Zeichnen
    //Benutzerinteraktion
    procedure userMouseDown(x,y:longint;Button: TMouseButton);
    procedure userMouseMove(x,y:longint;Shift: TShiftState);

    property usedMolecule: TMolecule read molecule write useMolecule;
  end;

implementation
```

```
uses forms,math,Dialogs;
{ TLewisRenderer }

//Ausgewähltes Atom löschen und Vorschaugeometrie aktualisieren
procedure TLewisRenderer.itemDeleteClick(Sender: TObject);
begin
  if selected<>nil then molecule.deleteAtom(selected);
  molecule.GeoOptimizeTrivial;
  paintBox.Refresh;
end;

//Alle Bindungen am Ausgewählten Atom löschen und Vorschaugeometrie aktualisieren
procedure TLewisRenderer.itemBondDeleteClick(Sender: TObject);
begin
  if selected=nil then exit;
  molecule.deleteBonds(selected);
  molecule.GeoOptimizeTrivial;
  paintBox.Refresh;
end;

//Moduswechseln (s.u.)
procedure TLewisRenderer.itemBondAddClick(Sender: TObject);
begin
  if selected=nil then exit;
  status:=sAddBond;
  paintBox.Refresh;
end;


//Nach Typ des neuen Atoms fragen und Vorschaugeometrie aktualisieren
procedure TLewisRenderer.itemAddClick(Sender: TObject);
var nt:string;
    i:longint;
begin
  nt:='H';
  if InputQuery('YAMS','Bitte geben Sie ein, welches Atom eingefügt werden soll. '+
                '(H,C,N oder O)',nt) then begin
    for i:=1 to high(ATOM_ABBREV) do
      if uppercase(ATOM_ABBREV[i])=uppercase(nt) then
        with molecule.addAtom(i) do begin
          flatX:=mx;
          flatY:=my;
          molecule.GeoOptimizeTrivial;
          paintBox.Refresh;
          exit;
```

```
        end;
    ShowMessage('Atom unbekannt');
  end;

end;

//Erzeugung (Menüerzeugung)
constructor TLewisRenderer.create;
var item:TMenuItem;
begin
  defX:=50;
  defY:=50;
  symHeight:=15;
  symWidth:=10;
  atomPopup:=TPopupMenu.Create(Application.MainForm);
  item:=TMenuItem.Create(atomPopup);
  item.Caption:='Atom Löschen';
  item.OnClick:=@itemDeleteClick;
  atomPopup.Items.Add(item);
  item:=TMenuItem.Create(atomPopup);
  item.Caption:='Atombindungen löschen';
  item.OnClick:=@itemBondDeleteClick;
  atomPopup.Items.Add(item);
  item:=TMenuItem.Create(atomPopup);
  item.Caption:='Atombindung hinzufügen';
  item.OnClick:=@itemBondAddClick;
  atomPopup.Items.Add(item);
  item:=TMenuItem.Create(atomPopup);
  item.Caption:='Atom hinzufügen';
  item.OnClick:=@itemAddClick;
  atomPopup.Items.Add(item);
end;

destructor TLewisRenderer.destroy;
begin
  atomPopup.free;
  inherited destroy;
end;

procedure TLewisRenderer.useMolecule(mol: TMolecule);
begin
  molecule:=mol;
  rearrange;
end;
```

```
//Flache Atomanordnung
procedure TLewisRenderer.rearrange();
const flatBondLength=20;
var arranged: array of boolean;
  //Ein Atom und Nachbarn anordnen
  procedure rar(a: tatom; dir: longint);
    //Bindungspriorität                                                    140
    function comp(const a,b:TBond): longint;
    const AP:array[1..8] of longint=(0,0,0,0,0,3,1,2);
    begin
      if arranged[a.a.index] then result:=1
      else if arranged[b.a.index] then result:=-1
      else if AP[a.a.typ]<AP[b.a.typ] then result:=1
      else if AP[a.a.typ]>AP[b.a.typ] then result:=-1
      else if a.count>b.count then result:=-1
      else if a.count<b.count then result:=1
      else result:=0;                                                      150
    end;
  //Mögliche Richtungen
  const DirRel:array[1..3] of array[0..3]of longint=
                ((2,3,0,1),(3,0,1,2),(1,2,3,0));
        RealDir: array[0..3] of array['x'..'y'] of longint=
                  ((-1,0),(0,-1),(1,0),(0,1));
  var i,j:longint;
      realBondCount: longint;
      fromBond: longint;
      temp:^tbond;                                                         160
      b: array[1..4] of ^TBond;
  begin
    //Bindungen nach Priorität sortieren
    dir:=DirRel[1,dir];
    for i:=1 to a.james do
      b[i]:=@a.bond[i];
    for i:=2 to a.james do
      for j:=i-1 downto 1 do
        if comp(b[j]^,b[j+1]^) > 0 then begin
          temp:=b[j];                                                      170
          b[j]:=b[j+1];
          b[j+1]:=temp;
        end;
    //Nachbaratome anordnen und von dort aus fortfahren
    for i:=1 to a.james do
      if not arranged[b[i]^.a.index] then begin
        b[i]^.a.flatX:=a.flatX+RealDir[DirRel[i,dir]]['x']*flatBondLength;
        b[i]^.a.flatY:=a.flatY+RealDir[DirRel[i,dir]]['y']*flatBondLength;
```

```
        arranged[b[i]^.a.index]:=true;
        rar(b[i]^.a,DirRel[i,dir]);
      end;
  end;
begin
  setlength(arranged,molecule.atoms.count);
  if length(arranged)=0 then exit;
  FillChar(arranged,length(arranged)*sizeof(arranged[0]),0);
  arranged[0]:=true;
  //Erstes Atom an Nullpunkt
  with tatom(molecule.atoms[0]) do begin
    flatX:=defX;
    flatY:=defY;
  end;
  rar(tatom(molecule.atoms[0]),0);
end;

//Molekül zeichnen
procedure TLewisRenderer.render(canvas: TCanvas;rect:TRect);
  //Zeichnet mehrere Linien nebeneinander
  procedure drawLines(x1,y1,x2,y2,count:longint);
  const LINE_SPACE:longint=3;
  var px,py: single;
      lx,ly: single;
      sx,sy,ex,ey: single;
      sizesub: single;
      len:single;
      i:longint;
  begin
    //Linienvektor
    lx:=x2-x1;
    ly:=y2-y1;
    //Fälle
    if count<=1 then begin  //Keine Linie
      px:=0;
      py:=0;
    end else if x1=x2 then begin //Steigung unendlich
      px:=1;
      py:=0;
    end else begin              //Vektor über Steigung
      //px*lx:=-py*ly da senkrecht
      px:=-ly/lx;
      py:=1;
      len:=sqrt(sqr(px)+sqr(py));
      px:=px/len;
```

```
      py:=py/len;
    end;
    //Vektor normieren
    len:=sqrt(sqr(lx)+sqr(ly));
    lx/=len;
    ly/=len;
    //Ausgangspunkte senkrecht verschieben, LINE_SPACE/2 Pixel pro Bindung
    //=> Symmetrie um Achse
    sizesub:=(abs(symHeight*ly)+abs(symWidth*lx))/2;
    sx:=x1+(sizesub+1)*lx-LINE_SPACE*px*(count-1)/2;
    sy:=y1+(sizesub+1)*ly-LINE_SPACE*py*(count-1)/2;
    ex:=x2-sizesub*lx-LINE_SPACE*px*(count-1)/2;
    ey:=y2-sizesub*ly-LINE_SPACE*py*(count-1)/2;
    //Bindungen nebeneinander zeichnen und dabei Verschiebung langsam umkehren
    if isnan(sx) or isnan(sy) or isnan(ex) or isnan(ey) then exit;
    for i:=1 to count do begin
      canvas.line(round(sx),round(sy),round(ex),round(ey));
      sx+=LINE_SPACE*px;
      sy+=LINE_SPACE*py;
      ex+=LINE_SPACE*px;
      ey+=LINE_SPACE*py;
    end;
  end;
var i,j:longint;
    totalBondUsed: longint;
begin
  with canvas do begin
    //Hintergrund
    Brush.color:=clWhite;
    Brush.Style:=bsSolid;
    FillRect(rect);
    Brush.Style:=bsClear;
    pen.color:=clBlack;
    font.Height:=symHeight;
    font.Style:=[fsBold];
    totalBondUsed:=0;
    //Atome zeichnen
    for i:=0 to molecule.atoms.count-1 do
      with tatom(molecule.atoms[i]) do begin
        if tatom(molecule.atoms[i])=selected then font.color:=clBlue
        else font.color:=clBlack;
        //Symbol
        TextOut(flatX-textWidth(ATOM_ABBREV[typ]) div 2,
                flatY-TextHeight(ATOM_ABBREV[typ]) div 2,
                ATOM_ABBREV[typ]);
```

```
        //Bindungen zeichnen
        totalBondUsed-=ATOM_CLOSED_BONDS[typ];
        for j:=1 to james do begin
          totalBondUsed+=bond[j].count;
          if index<bond[j].a.index then
            drawLines(flatX,flatY,bond[j].a.flatX,bond[j].a.flatY,bond[j].count);
        end;
      end;
    //Mögliche neue Bindung einzeichnen
    if (selected<>nil) and (status=sAddBond) then
      line(selected.flatX,selected.flatY,mx,my);
    //Hinweise an Benutzer ausgeben
    font.color:=clBlack;
    font.Style:=[];
    TextOut(10,10,'Anzahl Atome: '+IntToStr(molecule.atoms.count));
    with rect do
      TextOut(Right-TextWidth('Mit rechter Maustaste klicken für mehr Optionen')-3,
        bottom-TextHeight('Mp')-3,'Mit rechter Maustaste klicken für mehr Optionen');
    font.color:=clRed;
    if totalBondUsed<0 then
      TextOut(10,30,'Zu wenig Bindungen!');
    if totalBondUsed>0 then
      TextOut(10,30,'Zu viele Bindungen!');
  end;
end;

//Benutzer klickt
procedure TLewisRenderer.userMouseDown(x, y: longint; Button: TMouseButton);
var i:longint;
    newSelect:Tatom;
begin
  //Im Umkreis von 20 Pixeln nach Atomsymbol suchen
  newSelect:=nil;
  for i:=0 to molecule.atoms.count-1 do
    if sqr(tatom(molecule.atoms[i]).flatX-x)+
       sqr(tatom(molecule.atoms[i]).flatY-y) < 20 then begin
      newSelect:=tatom(molecule.atoms[i]);
      ox:=newSelect.flatX;
      oy:=newSelect.flatY;
      break;
    end;
  case status of
    sNormal: //Im Standardmodus auswählen
      selected:=newSelect;
```

```
    sAddBond: begin //Bindung zwischem ausgewählten und angeklickten Atom erzeugen
      if newSelect=nil then exit;
      status:=sNormal;
      molecule.addBond(newSelect,selected);
      molecule.GeoOptimizeTrivial; //Vorschaugeometrie aktualisieren
      selected:=nil;
    end;
  end;
  mx:=x;
  my:=y;
  paintBox.Refresh;
  if Button=mbRight then atomPopup.PopUp();
end;

//Benutzer hat die Maus bewegt
procedure TLewisRenderer.userMouseMove(x, y: longint; Shift: TShiftState);
begin
  case status of
    sNormal: if ssLeft in shift then begin //Atom wird verschoben
      if selected=nil then exit;
      selected.flatX:=ox+x-mx;
      selected.flatY:=oy+y-my;
      paintBox.Refresh;
    end;
    sAddBond: begin //Vorschaubindung zeigen
      mx:=x;
      my:=y;
      paintBox.Refresh;
    end;
  end;
end;

end.
```

E.5 atoms.pas

```
{Diese Datei enthält die Datenstruktur für ein Atom und elementare Information
 über diese
}
unit atoms;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils,math,extmath;

{Atomare Einheiten (von en.wikipedia):
length  Bohr radius   a0     5.291 772 108(18)*10-11 m
                             52.91 772 108(18)*10-12 m
                             0.5291 772 108 Â
mass    electron rest mass   me     9.109 3826(16)*10-31 kg
charge  elementary charge    e      1.602 176 53(14)*10-19 C
}
type
  TAtom=class;
  TBond=record            //Eine Bindung
    a:TAtom;              //zu einem Atom
    count: longint;       //Bindungszahl
  end;
  TBonds=array[1..4] of tbond;
  TAtom=class
  public
    index: longint;       //Index in TList
    typ: longint;         //Ordnungszahl
    bond:TBonds;          //kovalente bindungen
    james:longint;        //Anzahl Bindungen
    p: TVector3n;         //Raum Position

    flatX,flatY:  longint;  //2D Position (Lewis-Formel)

    firstOrbital: longint;  //Index des ersten Orbitals
  end;

const AT_H=1;             //Abkürzungen für Atome
      AT_C=6;
      AT_N=7;
      AT_O=8;
```

```
    //Abkürzungen als Strings
    ATOM_ABBREV:array[1..8]of string=('H','','','','','C','N','O');
    //Ausgeschriebene Namen
    ATOM_NAMES:array[1..8]of string=('Wasserstoff','','','','','Kohlenstoff',
                                     'Stickstoff','Sauerstoff');
    //Mögliche bindende Elektronenpaare
    ATOM_CLOSED_BONDS: array[1..8] of longint=(1,0,0,0,0,4,3,2);
    //Elektronegativität (Quelle: http://www.webelements.com/)
    EN:array[1..8] of number=(2.2,nan,nan,nan,nan,2.55,3.04,3.44);
    //Kovalenzradius (Quelle: de.wikipedia.org und http://www.webelements.com/)
    CovalentRadius:array[1..8] of number=(0.699,nan,nan,nan,nan,1.455,1.417,1.380);

implementation

end.
```

E.6 extmath.pas

```
{Dieser Programmteil bieten erweiterte Mathematikfunktionen und Datenstrukturen,
 die in normalem Pascal nicht vorhanden sind}
unit extmath;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils,math;
type
  {Strukturen für Vektoren und Matrizen}
  number = double;//extended;
  TVector3n=array['x'..'z'] of number;
  TVector3f=array['x'..'z'] of single;
  TVectorn=array of number;
  TVectorb=array of boolean;
  TVectori=array of longint;
  TMatrix=array of array of number;
  TSymMatrix=array of array of number; //a[i,j] existiert-> i<=j (obere Dreiecksmat.)

  {Vorberechnete Funktionen}
//Einfache Fakultät
const LOOK_UP_FAK:array[0..11] of longint=(1,1,2,6,24,120,720,5040,40320,
                                           362880,3628800,39916800);
//Doppelte Fakultät (2n-1)!!
const LOOK_UP_FAK2:array[-1..57] of number=
 (1,1,{1:}1,2,3,8,15,48,105,384,945,3840,10395,46080,135135,645120,2027025,10321920,
  34459425,185794560,654729075,3715891200,13749310575,81749606400,316234143225,
  1961990553600,7905853580625,51011754393600,213458046676875,1428329123020800,
  6190283353629375,42849873690624000,191898783962510625,1371195958099968000,
  6332659870762850625,46620662575398912000, 221643095476699771875,
  1678343852714360832000,8200794532637891559375, 637770660340314561616000,
  3198309867728777708156 25,25510826561258284646400 00, 131130704576879886034406 25,
1071454715572847955148800 00,5638620296805835099479468 75,4714400748520531002654720000,
  2537379133562625794765760 9375,2168624344319444261221171 20000,
  1192568192774434123539907640625,1040939685273333245386162176 0000,
  5843584144594727205345547439 0625, 5204698426366666226930810880 00000,
  2980227913743310874726229193921875, 2706443181710666438004021657 6000000,
  1579520794283954763604901472778593 75,1461479318123759876522171695104000000,
  8687364368561751199826958100282265625,8184284181493055308524161492582400 0000,
  4951797690080198183901366117160891406 25);
{Zweier Potenzen}
```

```
const LOOK_UP_2POWER:array[-13..61] of number=
  (1/8192,1/4096,1/2048,1/1024,1/512,1/256,1/128,1/64,1/32,1/16,1/8,1/4,1/2,
   1,2,4,8,16,32,64,128,256,512,1024,2048,4096,8192,16384,32768,65536,131072,
   262144,524288,1048576,2097152,4194304,8388608,16777216,33554432, 67108864,
   134217728,268435456,536870912,1073741824, 2147483648,4294967296,8589934592,
   17179869184,34359738368,68719476736,137438953472,274877906944,549755813888,
   1099511627776,2199023255552,4398046511104,8796093022208,17592186044416,
   35184372088832,70368744177664,140737488355328,281474976710656,562949953421312,
   1125899906842624,2251799813685248,4503599627370496,9007199254740992,
   18014398509481984,36028797018963968,72057594037927936,144115188075855872,
   288230376151711744,576460752303423488,1152921504606846976,2305843009213693952
  );
//Binomialkoeffizienten
const LOOK_UP_PASCAL:array[0..11,0..11] of longint= //Binomialkoeffizienten
    ((                          1,              0,0,0,0,0,0,0,0,0,0),
     (                        1,  1,             0,0,0,0,0,0,0,0,0),
     (                      1,  2,  1,           0,0,0,0,0,0,0,0),
     (                    1,  3,  3,  1,         0,0,0,0,0,0,0),
     (                  1,  4,  6,  4,  1,       0,0,0,0,0,0),
     (                1,  5, 10, 10,  5,  1,     0,0,0,0,0),
     (              1,  6, 15, 20, 15,  6,  1,   0,0,0,0),
     (            1,  7, 21, 35, 35, 21,  7,  1,   0,0,0),
     (          1,  8, 28, 56, 70, 56, 28,  8,  1,   0,0),
     (        1,  9, 36, 84, 126,126, 84,36,  9,  1,  0,0 ),
     (  1, 10, 45,120,210,252,210,120, 45, 10, 1, 0),
     (1, 11, 55, 165,330,462,462,330,165,55, 11,  1));
{Null}
const epsilon16=1e-16;
const nullvector3n: TVector3n=(0,0,0);
      nullvector3ic:array['x'..'z'] of longint=(0,0,0);

//===Lineare Algebra===
//-Vektorrechnung-
//Elementare Vektorfunktionen (s. Formelsammlung)
function vec2f(const v: TVector3n): TVector3f;
procedure vecscale(var v: TVector3n; const scale: number);inline;overload;
function vecscale(const scale: number;const v: TVector3n): TVector3n;inline;overload;
function vecadd(const v1,v2: TVector3n): TVector3n;inline;
function vecsub(const v1,v2: TVector3n):TVector3n;inline;
function vecsub(const v1,v2: TVector3f):TVector3f;inline;
function veclensqr(const v: TVector3n):number;inline;overload;
function veclensqr(const v: TVector3f):single;inline;overload;
function vecdist(const v1,v2: TVector3n):number;inline;overload; //Vektordistanz
procedure vecnormalize(var v:TVector3n);                    //Normalisieren
function vecout(const v: TVectorn): string;
```

```
function mirrorVec(const v,n:TVector3n):TVector3n; //Vektorspiegelung an Ebene
//Umrechnen in Kugelkoordinaten                                              90
procedure vec2sphere(const v:TVector3n;var r,phi,theta: number);

//–Matrixrechnung—
procedure matinit(var M:TMatrix;const r,c:longint);
//Erzeugen von Rotationsmatrizen
function createRotationMat3(const alpha:number; const axe: char):TMatrix;
function createRotationMat3(const theta: number; const axe: TVector3n): TMatrix;
//Löschen einer Zeile
procedure materaseLine(var M:TMatrix; const l:longint);inline;
//Matrix mal Vektor                                                          100
function matVecTrans(const M:TMatrix;const v: TVector3n):TVector3n;
//Untere Dreiecksmatrix mit oberer füllen
procedure matsym2full(var M: TMatrix);
//Matrixmultiplikation
procedure matmul(const A,B:TMatrix;var R: TMatrix);
procedure matmul_sym(const A,B:TSymMatrix;var R: TSymMatrix);
procedure matmul_withtransA(const A,B:TMatrix;var R: TMatrix);
//Berechnung von A^-0.5
procedure matinvsqrt_sym(var A:TSymMatrix;var T1,T2: TMatrix; var Tvb: TVectorb;
                         var Tvi: TVectori; var Tv: TVectorn; var R: TSymMatrix);  110
//Diagonalisieren einer Matrix
procedure diag_sym(var M: TSymMatrix; var changed: TVectorb;var ind: TVectori;
                   var eigenvalues:TVectorn; var eigenvectors: TMatrix);
function matout(const mat: TMatrix): string;

//—Analysis—
//base^exponent
function intpower00(base : float;const exponent : Integer) : float; //0^0 = 0
function intpower01(base : float;const exponent : Integer) : float; //0^0 = 1
//Summe der Binomialkoeffizienten                                            120
function binomSum(const j,l,m:longint; const a,b:number):number;
//Berechnet die Fehlerfunktion
function erf(const v: number;const delta: number=epsilon16): number;

procedure assertcomp(n1,n2:number;s:string;prec:number=0.000001);inline;
implementation
uses dialogs,windows;
//Wandelt einen Vektor in einen 3 float Vektor um
function vec2f(const v: TVector3n): TVector3f;
begin                                                                         130
  result['x']:=single(v['x']);
  result['y']:=single(v['y']);
  result['z']:=single(v['z']);
```

```
end;
//Elementare Vektorrechnung
procedure vecscale(var v: TVector3n; const scale: number); inline;
begin
  v['x']:=scale*v['x'];
  v['y']:=scale*v['y'];
  v['z']:=scale*v['z'];
end;

function vecscale(const scale: number; const v: TVector3n): TVector3n; inline;
begin
  Result:=v;
  vecscale(result,scale);
end;

function vecadd(const v1, v2: TVector3n): TVector3n; inline;
begin
  result['x']:=v1['x']+v2['x'];
  result['y']:=v1['y']+v2['y'];
  result['z']:=v1['z']+v2['z'];
end;

//Subtrahiert zwei Vektoren
function vecsub(const v1,v2: TVector3n):TVector3n;inline;
begin
  result['x']:=v1['x']-v2['x'];
  result['y']:=v1['y']-v2['y'];
  result['z']:=v1['z']-v2['z'];
end;

function vecsub(const v1, v2: TVector3f): TVector3f; inline;
begin
  result['x']:=v1['x']-v2['x'];
  result['y']:=v1['y']-v2['y'];
  result['z']:=v1['z']-v2['z'];
end;

//Berechnet das Quadrat der Länge des Vektors
function veclensqr(const v: TVector3n):number;inline;
begin
  result:=sqr(v['x'])+sqr(v['y'])+sqr(v['z']);
end;

function veclensqr(const v: TVector3f): single; inline;
begin
```

```
  result:=sqr(v['x'])+sqr(v['y'])+sqr(v['z']);
end;

function vecdist(const v1, v2: TVector3n): number; inline;
begin
  result:=sqrt(veclensqr(vecsub(v1,v2)));
end;

function vecout(const v: TVectorn): string;
var i:integer;
begin
  result:='';
  for i:=0 to high(v) do
    result:=result+format('%2.4g',[v[i]])+#9;
end;

//Vektor v an Ebene mit Normalenvektor n durch 0,0,0 spiegeln
function mirrorVec(const v, n: TVector3n): TVector3n;
var d:number;
begin
  //Der Abstand von v und v' zur Ebene ist gleich
  //v-d*n=v'+d*n
  //v'=v-2*d*n
  d:=v['x']*n['x']+v['y']*n['y']+v['z']*n['z'];
  result:=vecsub(v,vecscale(2*d,n));
end;

//Umrechnen in Kugelkoordinaten
procedure vec2sphere(const v: TVector3n; var r, phi, theta: number);
begin
  r:=sqrt(veclensqr(v));
  phi:=arccos(v['z']/r);
  theta:=arctan2(v['y'],v['x']);
end;

procedure vecnormalize(var v: TVector3n);
var len: number;
begin
  len:=sqrt(veclensqr(v));
  vecscale(v,1/len);
end;

//Wandelt die Matrix in ein für Menschen lesbares Format um
function matout(const mat: TMatrix): string;
```

```
const useStyle=2;
const STYLE:array[1..2,0..4] of string=(('','',#9,#13#10,''), //Menschen
                                        ('.','[[',',',']','[',']]'));//Derive
var i,j:longint;
begin
  if style[usestyle][0]<>'' then DecimalSeparator:=style[usestyle][0][1];
  result:=STYLE[useStyle][1];                                                  230
  for i:=0 to high(mat) do begin
    for j:=0 to high(mat[i]) do begin
      result:=result+format('%2.4g',[mat[i,j]]);
      if j<>high(mat[i]) then
        result+=STYLE[useStyle][2];
    end;
    if i<>high(mat) then
      result:=result+STYLE[useStyle][3];
  end;
  result:=result+STYLE[useStyle][4];                                           240
end;

//Berechnet base^exponent schnell und gibt 0 für 0^0 zurück
function intpower00(base : float;const exponent : Integer) : float;
begin
 if (base=0) and (exponent=0) then exit(0)
 else exit(intpower(base,exponent));
end;

//Berechnet base^exponent schnell und gibt 1 für 0^0 zurück                    250
function intpower01(base : float;const exponent : Integer) : float;
begin
 if (base=0) and (exponent=0) then exit(1)
 else exit(intpower(base,exponent));
end;


//Berechnet den Faktor vor dem Summanden mit dem angegebenen Exponenten
//     l        m     l+m      l+m-1                    0  l   m
//(x+a) *(x+b) = x   + x         (l*a+m*b) + ... + x  a  b                    260
//
//Also, binomSum(l+m,l.m) = 1, binomSum(l+m-1,l.m) = l*a+m,
//      ...                                 binomSum(0,l.m) = a^l * b^m,
function binomSum(const j,l,m:longint; const a,b:number):number;
var i,f:longint;
    ac,bc:number;
begin
  if j>l+m then exit(0);
```

```
  if j=l+m then exit(1);
  if (a=0) and (b=0) then exit(0);
  if a=0 then
    if l+m-j>m then exit(0)
    else exit(LOOK_UP_PASCAL[m,l+m-j]* intpower(b,l+m-j));
  if b=0 then
    if l+m-j>l then exit(0)
    else exit(LOOK_UP_PASCAL[l,l+m-j]*intpower(a,l+m-j));


  result:=0;
  f:=max(0,j-m);
  ac:=intpower01(a,l-f);
  bc:=intpower01(b,m+f-j);
  for i:=f to min(j,l) do begin
    result:=result+LOOK_UP_PASCAL[l,i]*LOOK_UP_PASCAL[m,j-i]*ac*bc;
    ac/=a;
    bc*=b;
  end;
end;

//Berechnet die Fehlerfunktion
// v      -x^2
// S dx e
// 0
//nach http://mathworld.wolfram.com/Erf.html
function erf(const v: number;const delta: number): number;
var change,value,fak,v_power,v_sqr,n: number;
    sign: boolean;
begin
  if IsZero(v) then exit(0)
  else if SameValue(v,1) then exit(0.842700792950719442 9276)
  else if v>3.8325063453686 then exit(1)
  else if v < 0 then exit(-erf(-v,delta))
  else begin
    value:=v;
    fak:=1;
    v_sqr:=v*v;
    v_power:=v*v_sqr;
    sign:=true;
    n:=1;
    repeat
      change:=v_power/(fak*(2*n+1));
      if sign then value:=value-change
      else value:=value+change;
```

```
      v_power:=v_power*v_sqr;
      sign:=not sign;
      n:=n+1;
      fak:=fak*n;
    until abs(change)<delta;
    result:=value*2/sqrt(pi);
  end;
end;

procedure assertcomp(n1, n2: number; s: string;prec:number=0.000001);inline;
begin
  assert(samevalue(n1,n2,prec),s+#13#10+floattostr(n1)+'<>'+floattostr(n2));
end;

//Einheitsmatrix erstellen
procedure matinit(var M: TMatrix; const r, c: longint);
var i:longint;
begin
  SetLength(M,r,c);
  for i:=0 to r-1 do materaseLine(m,i);
  for i:=0 to min(r-1,c-1) do
    m[i,i]:=1;
end;

//Triviale Rotation um Achsenkoordinaten
//nach http://www.grundstudium.info/animation/node14.php
function createRotationMat3(const alpha:number; const axe: char):TMatrix;
var s,c:number;
begin
  s:=sin(alpha);
  c:=cos(alpha);
  matinit(result,3,3);
  case axe of
    'x':begin
      result[1,1]:=c;
      result[1,2]:=-s;
      result[2,1]:=s;
      result[2,2]:=c;
    end;
    'y':begin
      result[0,0]:=c;
      result[0,2]:=-s;
      result[2,0]:=s;
      result[2,2]:=c;
    end;
```

```
    'z':begin
      result[0,0]:=c;
      result[0,1]:=-s;
      result[1,0]:=s;
      result[1,1]:=c;
    end;
  end;
end;

//Rotation um eine beliebige Achse
//Nach http://www.grundstudium.info/animation/node14.php
function createRotationMat3(const theta: number; const axe: TVector3n): TMatrix;
var {Rx,Ry,Rz,Ry_inv,Rx_inv: TMatrix;
    alpha,beta,d:number;}
    si,co:number;
begin
  if (axe['y']=0)and(axe['z']=0) then
    exit(createRotationMat3(theta,'x'));
  si:=sin(theta);
  co:=cos(theta);
  setlength(result,3,3);

  result[0, 0] := ((1-co) * sqr(axe['x'])) + co;
  result[0, 1] := ((1-co) * axe['x'] * axe['y']) - (axe['z'] * si);
  result[0, 2] := ((1-co) * axe['z'] * axe['x']) + (axe['y'] * si);

  result[1, 0] := ((1-co) * axe['x'] * axe['y']) + (axe['z'] * si);
  result[1, 1] := ((1-co) * sqr(axe['y'])) + co;
  result[1, 2] := ((1-co) * axe['y'] * axe['z']) - (axe['x'] * si);

  result[2, 0] := ((1-co) * axe['z'] * axe['x']) - (axe['y'] * si);
  result[2, 1] := ((1-co) * axe['y'] * axe['z']) + (axe['x'] * si);
  result[2, 2] := ((1-co) * sqr(axe['z'])) + co;
end;

//Elementare Matrixrechnung
procedure materaseLine(var M: TMatrix; const l: longint);inline;
begin
  fillchar(M[l][0],length(M[l])*sizeof(M[l][0]),0);
end;

function matVecTrans(const M: TMatrix; const v: TVector3n):TVector3n;
var i,j:longint;
    c:char;
begin
```

```
  for c:='x' to 'z' do begin
    result[c]:=0;
    for j:=0 to 2 do
      result[c]+=M[ord(c)-ord('x')][j]*v[chr(ord('x')+j)];
  end;
end;

procedure matsym2full(var M: TMatrix);
var i,j: longint;
begin
  assert(length(M)>0,'Matrix leer');
  assert(length(M[0])=length(M),'nicht symmetrisch');
  for i:=0 to high(M) do
    for j:=i+1 to high(M) do
      M[j,i]:=M[i,j];
end;

procedure matmul(const A, B: TMatrix; var R: TMatrix);
var i,j,k:longint;
begin
  assert(length(A[0])=length(B),'Ungültige Größe für Multiplikation');
  assert(length(R)=length(B[0]),'Ungültige Größe für Multiplikation');
  assert(length(R[0])=length(B[0]),'Ungültige Größe für Multiplikation');

  for i:=0 to high(A) do
    for j:=0 to high(B[0]) do begin
      r[i,j]:=0;
      for k:=0 to high(B) do
        r[i,j]+=A[i,k]*B[k,j];
    end;
end;

procedure matmul_sym(const A, B: TMatrix; var R: TMatrix);
var i,j,k:longint;
begin
  assert(length(A)=length(B),'Ungültige Größe für Multiplikation');
  assert(length(A)=length(A[0]),'Ungültige Größe für Multiplikation');
  assert(length(A)=length(B[0]),'Ungültige Größe für Multiplikation');
  assert(length(A)=length(R),'Rückgabe Matrix falsch');
  assert(length(A)=length(R[0]),'Rückgabe Matrix falsch');

  for i:=0 to high(A) do
    for j:=i to high(A) do begin
      //Berechnung ist trivial, aber trickreich
      //Für jedes Koordinatenpaar [i,j] muss gelten i<=j
```

```
      r[i,j]:=0;
      for k:=0 to i do
        r[i,j]+=A[k,i]*B[k,j];
      for k:=i+1 to j do
        r[i,j]+=A[i,k]*B[k,j];
      for k:=j+1 to high(A) do
        r[i,j]+=A[i,k]*B[j,k];
    end;
end;

//Multiplisiert A mit der Transponierten von B
procedure matmul_withtransA(const A, B: TMatrix; var R: TMatrix);
var i,j,k:longint;
begin
  assert(length(A)>0,'Matrix leer');
  assert(length(A[0])=length(B[0]),'Ungültige Größe für Multiplikation');
  assert(length(R)=length(B[0]),'Ungültige Größe für Multiplikation');
  assert(length(R[0])=length(B[0]),'Ungültige Größe für Multiplikation');
  for i:=0 to high(A) do
    for j:=0 to high(B) do begin
      r[i,j]:=0;
      for k:=0 to high(A[0]) do
        r[i,j]+=A[k,i]*B[k,j];
    end;
end;

//Berechnet R=A^-0.5
//T1,T2,Tv sind temporär, müssen die gleiche Größe haben
//Nach Cook, Ab Initio Valence Calculations in Chemistry
procedure matinvsqrt_sym(var A:TSymMatrix;var T1,T2: TMatrix; var Tvb: TVectorb;
                         var Tvi: TVectori; var Tv: TVectorn; var R: TSymMatrix);
var ev: TVectorn;
    i,j:longint;
begin
  assert(length(A)>0,'Matrix leer');
  assert(length(A[0])=length(A),'Ungültige Größe für M^-0.5');
  assert(length(R)=length(A),'Ungültige Größe für M^-0.5: r');
  assert(length(R[0])=length(A),'Ungültige Größe für M^-0.5: r');
  assert(length(T1)=length(A),'Ungültige Größe für M^-0.5: t1');
  assert(length(T1[0])=length(A),'Ungültige Größe für M^-0.5: t1');
  assert(length(T2)=length(A),'Ungültige Größe für M^-0.5: t2');
  assert(length(T2[0])=length(A),'Ungültige Größe für M^-0.5: t2');

  diag_sym(A,Tvb,Tvi,Tv,T1);
  //ev:=ev^-1/2
```

```
  for i:=0 to high(Tv) do
    Tv[i]:=1/sqrt(Tv[i]);
  //T2:=T1*Tv;
  for i:=0 to high(T1) do
    for j:=0 to high(T1) do
      T2[i,j]:=T1[i,j]*Tv[i];
  //R:=temp2*temp1;                                                    500
  matmul_withtransA(t1,t2,r);
end;

{//Diagonalisiert die Matrix M
//Die obere Seite von M wird zerstört
//Die Eigenvektoren sind in der Matrix Zeilenvektoren
//von http://en.wikipedia.org/wiki/Jacobi_eigenvalue_algorithm
//version 20:23, 22 October 2006
//version-url:
http://en.wikipedia.org/w/index.php?title=Jacobi_eigenvalue_algorithm&oldid=83064407  510
}
procedure diag_sym(var M: TSymMatrix; var changed: TVectorb;var ind: TVectori;
                   var eigenvalues:TVectorn; var eigenvectors: TMatrix);
  //index of largest element in row k
  function maxind(k: longint): longint;
  var i:longint;
  begin
    result:=k+1;
    for i := k+2 to high(M[k]) do
      if abs(M[k,i]) > abs(M[k,result]) then result:=i;                  520
  end;

var state:longint;
  procedure update(k:longint; t: number); // update ek and its status
  var y:number;
  begin
    y := eigenvalues[k];
    eigenvalues[k] := y+t;
    if changed[k] and (IsZero(t)) then begin
      changed[k] := false;                                               530
      state := state-1;
    end else if (not changed[k]) and (not IsZero(t)) then begin
      changed[k] := true;
      state := state+1
    end;
  end;

  procedure rotate(k,l,i,j : longint;s,c:number); // perform rotation of Sij, Skl
```

```
  var Mkl,Mij:number;
  begin
   {|   |     |      ||   |
    |Skl|     |c  -s||Skl|
    |   | := |      ||   |
    |Sij|     |s   c||Sij|
    |   |     |      ||   |}
    Mkl:=M[k,l];
    Mij:=M[i,j];
    M[k,l]:=c*Mkl−s*Mij;
    M[i,j]:=s*Mkl+c*Mij;
  end;

var
  i, j, k, l: longint;
  s, c, t, p, y, Eki, Eli: number;
begin
  assert(length(M)>0,'Matrix leer');
  assert(length(M)=length(M[0]),'Nur quadratische Matrizen werden diagonalisiert');
  assert(length(eigenvectors)=length(M),'Größe falsch');
  assert(length(eigenvectors[0])=length(M),'Größe falsch');
  assert(length(eigenvalues)=length(M),'Größe falsch');
  assert(length(ind)=length(M),'Größe falsch');
  assert(length(changed)=length(M),'Größe falsch');
  for i:=0 to high(eigenvectors) do begin
    fillchar(eigenvectors[i,0],length(M)*sizeof(eigenvectors[i,0]),0);
    eigenvectors[i,i]:=1;
  end;
  state := length(M);
  for k := 0 to high(M) do begin
    ind[k] := maxind(k);
    eigenvalues[k] :=M[k,k];
    changed[k] := true;
  end;
  //Berechnung
  while state>0 do begin// next rotation
    l := 0; // find index (k,l) of pivot p
    for k := 1 to high(M)−1 do
      if abs(M[k,ind[k]])> abs(M[l,ind[l]]) then l := k;
    k := l;l := ind[k];p := M[k,l];
    if iszero(p) then break;
    // calculate c = cos phi, s = sin phi
    y := (eigenvalues[l]−eigenvalues[k])/2;
    t := abs(y)+sqrt(p*p+y*y);
    s := sqrt(p*p+t*t);
```

```
    c := t/s; s := p/s; t := p*p/t;
    if y<0 then begin
      s := −s;
      t := −t;
    end;
    M[k,l] := 0.0; update(k,−t); update(l,t);
    // rotate rows and columns k and l                                    590
    for i := 0 to k−1 do rotate(i,k,i,l,s,c);
    for i := k+1 to l−1 do rotate(k,i,i,l,s,c);
    for i := l+1 to high(M) do rotate(k,i,l,i,s,c);
    // rotate eigenvectors
    for i := 0 to high(eigenvectors) do begin
      {? ? ? ?? ?
      ?Eki? ?c -s??Eki?
      ? ? := ? ?? ?
      ?Eli? ?s c??Eli?
      ? ? ? ?? ?}                                                        600
      eki:=eigenvectors[k,i];
      eli:=eigenvectors[l,i];
      eigenvectors[k,i]:=c*eki−s*eli;
      eigenvectors[l,i]:=c*eli+s*eki;
    end;
    // rows k, l have changed, update rows indk, indl
    ind[k] := maxind(k); ind[l] := maxind(l);
  end;
end;
end.                                                                     610
```

E.7 gtf.pas

```
{Dieser Programmteil enthält die Datenstrukturen für Gauss Type Functions
 und Berechnungsfunktionen für die dabei möglichen Integrale

 Die Funktionen folgen Cook, Ab Initio Valence Calculations in Chemistry
 und Carsky/Urban, Ab Initio Calculations
 }
unit gtf;

{$mode objfpc}{$H+}
{$define performtests}

interface

uses
  Classes, SysUtils,math,extmath,Dialogs;

type
  //Datenstruktur für eine allgemeine Gauss Type Function
  TGTF=record
    e:array['x'..'z'] of longint; //Exponenten
    n:TVector3n;                  //KO-System Nullpunkt
    alpha: number;                //Exponent a
    pre: number;                  //Vorfaktor b
    //Funktion der Form
    {       xe     ye     ze    -a*r^2
      b*(x-x0) (y-y0) (z-z0) e
    }
  end;

  //Unskalierte GTF am Koordinatennullpunkt
  TFreeGTF=record
    e:array['x'..'z'] of longint; //Exponenten
    alpha: number; //Exponent a
    //Funktion der Form
    { xe  ye  ze  -a*r^2
      x   y   z   e
    }
  end;

  //Datenstruktur für ein vorberechnetes Überlappungsintegral
  TGTFPreCalculatedOverlap=record
    factor,                   //Vorfaktor
    alphasum,                 //alpha1 + g.alpha2
```

```
    alphamulprosum: number; //alpha1*alpha2/alphasum
    f,g:TFreeGTF;          //freie Orbitale
  end;

  //Datenstruktur für ein vorberechnetes kinetisches Energieintegral
  TGTFPreCalculatedKinetic=record
    //Zusammengesetzt aus overlaps
    overlap: array[0..3] of TGTFPreCalculatedOverlap;
  end;

  //Datenstruktur für ein vorberechnetes Kernanziehungsintegral
  TGTFPreCalculatedNuclearAttraction=record
    factor: number;              //Vorfaktor
    f,g:TFreeGTF;                //freie Orbitalfunktionen
    coeffsum: longint;           //Die Summe alle Koeffizienten (fi.e[c])
    a_cache:array['x'..'z',0..4,0..2,0..2] of number; //Vorberechnete Faktoren
  end;

  //Datenstruktur für ein vorberechnetes Elektronenwechselwirkungsintegral
  TGTFPreCalculatedElectronRepulsion=record
    factor: number;
    f,g: TFreeGTF;
    alphasum: number; //Die Summe der alphas und Kehrwerte
    coeffsum: longint;//Die Summe alle Koeffizienten (fi.e[c])
    b_cache: array['x'..'z',0..4,0..4,0..2,0..2,0..4] of number; //Vorberechnung
    a_power: array[-4..0] of number; //Potenzen von alphasum
  end;

//Berechnet das Integral
//  1      2v  -ts^2
// S  ds s    e
//  0
function f_erf_like(const v:longint;const t,erf_sqrt_t:number):number;overload;
function f_erf_like(const v:longint;const t:number):number;inline;overload;

//Berechnet den Mittelpunkt zweier GTF-Orbitale
function midpoint(const a1,a2:number;const p1,p2:TVector3n):TVector3n;inline;

//Berechnet einen normalisiertenden Vorfaktor
function normalizeFactorGTF(const l,m,n:longint;const alpha:number): number;
        inline;overload;

//Berechnet das Überlappungsintegral
function preCalculateOverlapIntegral(const factor:number;const f,g:TFreeGTF):
        TGTFPreCalculatedOverlap;
```

```
function overlapIntegral(const pre: TGTFPreCalculatedOverlap;
                         const p1,p2: TVector3n): number;
//Berechnet das Integral der kinetischen Energie
function preCalculateKineticIntegral(const factor:number;const f,g:TFreeGTF):
        TGTFPreCalculatedKinetic;
function kineticIntegral(const pre: TGTFPreCalculatedKinetic;
                         const p1,p2: TVector3n): number;
//Berechnet die Energie der Anziehung zwischen Kern und Orbital
function preCalculateNuclearAttractionIntegral(const factor:number;
                              const f,g:TFreeGTF):TGTFPreCalculatedNuclearAttraction;
function nuclearAttractionIntegral(const pre:TGTFPreCalculatedNuclearAttraction;
                                   const p1,p2,kern: TVector3n):number;
//Berechnet die durch die Abstoßung zwischen Elektronen gegebene Energie
function preCalculateElectronRepulsionIntegral(const factor:number;
                             const f,g:TFreeGTF):TGTFPreCalculatedElectronRepulsion;
function electronRepulsionIntegral(const pre12,pre34:
            TGTFPreCalculatedElectronRepulsion;const p1,p2,p3,p4:TVector3n):number;
implementation


//Berechnet das Integral
//  1      2v  -ts^2
// S  ds s    e
//  0
//nach http://mathworld.wolfram.com/GaussianIntegral.html
//     und Beispielsrechnungen mit Derive
function f_erf_like(const v:longint;const t,erf_sqrt_t:number):number;overload;
var temp: number;
    i:longint;
begin
  if isZero(t) then exit(1/(2*v+1))
  else begin
    assert(t>0,'Negative Zahlen sind in f_erf_like nicht implementiert');
    if v>=1 then begin
      temp:=1/t;
      result:=0;
      for i:=v-1 downto 0 do begin
        result-=LOOK_UP_FAK2[2*v-1]/(LOOK_UP_2POWER[v-i]*LOOK_UP_FAK2[2*i+1])*tem
        temp/=t;
      end;
      result:=exp(-t)*result;
    end else result:=0;
     result+=LOOK_UP_FAK2[2*v-1]*sqrt(pi)*erf_sqrt_t/((LOOK_UP_2POWER[v+1])*
             power(t,v+0.5));
  end;
```

```
end;

function f_erf_like(const v:longint;const t:number):number;inline;overload;
begin
  if isZero(t) then exit(1/(2*v+1))
  else exit(f_erf_like(v,t,erf(sqrt(t))));
end;                                                                    140

//Berechnet den Mittelpunkt zweier GTF-Orbitale als Vektor
//nach Cook, Ab Initio Valence Calculations in Chemistry
function midpoint(const a1,a2:number;const p1,p2:TVector3n):TVector3n;inline;
var a12:number;
begin
  a12:=a1+a2;
  result['x']:=(a1*p1['x']+a2*p2['x'])/a12;
  result['y']:=(a1*p1['y']+a2*p2['y'])/a12;
  result['z']:=(a1*p1['z']+a2*p2['z'])/a12;                            150
end;

//Berechnet einen normalisierenden Vorfaktor für eine GTF
//nach Cook, Ab Initio Valence Calculations in Chemistry und Überprüfungen mit Derive
function normalizeFactorGTF(const l,m,n:longint;const alpha:number): number;
  inline;overload;
begin
  result:=sqrt((intpower(4,l+m+n)*sqrt(8)*power(alpha,l+m+n+1.5))/
              (LOOK_UP_FAK2[l]*LOOK_UP_FAK2[m]*LOOK_UP_FAK2[n]*sqrt(pi*pi*pi)));
end;                                                                   160

//Berechnet das Überlappungsintegral
function preCalculateOverlapIntegral(const factor:number;const f,g:TFreeGTF):
        TGTFPreCalculatedOverlap;
begin
  result.alphasum:=f.alpha+g.alpha;
  result.alphamulprosum:=f.alpha*g.alpha/result.alphasum;
  result.factor:=factor*power(pi/result.alphasum, 1.5);
  result.f:=f;
  result.g:=g;                                                         170
end;

//Berechnet ein verschobenes Überlappungsintegral
function overlapIntegral(const pre: TGTFPreCalculatedOverlap;
                         const p1,p2: TVector3n): number;
  function asym(const c: char):number;
  const LOOK_UP_FACTOR:array[0..5] of number=(1,1/2,3/4,15/8,105/16,945/32);
  var i:longint;
```

```
      factor: number;
  begin
    case (pre.f.e[c]+pre.g.e[c])div 2 of
      0: if pre.f.e[c]=pre.g.e[c] then result:=1 //beide 0
         else if pre.f.e[c]>=1 then
           result:=(pre.f.alpha*p1[c]+pre.g.alpha*p2[c])/pre.alphasum-p1[c]//ortho.
         else result:=(pre.f.alpha*p1[c]+pre.g.alpha*p2[c])/pre.alphasum-p2[c];
      1..5: begin
        result:=0;
        for i:=0 to (pre.f.e[c]+pre.g.e[c]) div 2 do begin
         factor:=binomSum(2*i,pre.f.e[c],pre.g.e[c],
                          (pre.f.alpha*p1[c]+pre.g.alpha*p2[c])/pre.alphasum-p1[c],
                          (pre.f.alpha*p1[c]+pre.g.alpha*p2[c])/pre.alphasum-p2[c]);
         Result:=result+factor*LOOK_UP_FACTOR[i]/intpower(pre.alphasum,i);
        end;
      end;
      else assert(false,'Orbitaltyp wird nicht unterstützt');
    end;
  end;
begin
  result:=pre.factor;
  result*=exp(-veclensqr(vecsub(p1,p2))*pre.alphamulprosum);
  result*=asym('x')*asym('y')*asym('z');
end;

//Berechnet das Integral der kinetischen Energie
function preCalculateKineticIntegral(const factor: number; const f,
  g: TFreeGTF): TGTFPreCalculatedKinetic;
var temp: TFreeGTF;
    c:char;
begin
  result.overlap[0]:=preCalculateOverlapIntegral(
      factor*g.alpha*(2*(g.e['x']+g.e['y']+g.e['z'])+3),f,g);
  temp:=g;
  for c:='x' to 'z' do begin
    temp.e[c]+=2;
    result.overlap[ord(c)-ord('x')+1]:=
      preCalculateOverlapIntegral(-factor*2*sqr(g.alpha),f,temp);
    temp.e[c]-=2;
  end;
end;

function kineticIntegral(const pre: TGTFPreCalculatedKinetic;
  const p1, p2: TVector3n): number;
var i:longint;
```

```
begin
  result:=0;
  for i:=0 to 3 do
    result+=overlapIntegral(pre.overlap[i],p1,p2);
end;

//Berechnet die Energie der Anziehung zwischen Kern und Orbital
function preCalculateNuclearAttractionIntegral(const factor:number;
                                const f,g:TFreeGTF):TGTFPreCalculatedNuclearAttraction;
var c:char;
    i,r,u:longint;
begin
  assert((f.alpha>0)and(g.alpha>0),'Falsche Alphawerte');
  Result.factor:=factor*2*pi/(f.alpha+g.alpha);
  result.f:=f;
  result.g:=g;
  result.coeffsum:=f.e['x']+g.e['x']+f.e['y']+g.e['y']+f.e['z']+g.e['z'];
  for c:='x' to 'z' do begin
    assert(f.e[c]+g.e[c]<=2,
          'Orbitaltyp wird nicht unterstützt'); //geht nur mit s,d,p
    for i:=0 to f.e[c]+g.e[c] do
      for r:=0 to i div 2 do
        for u:=0 to (i - 2*r) div 2 do begin
          if odd(i+u) then result.a_cache[c,i,r,u]:=-1
          else result.a_cache[c,i,r,u]:=1;
          result.a_cache[c,i,r,u]*=intpower(0.25/(f.alpha+g.alpha),r+u)*
             LOOK_UP_FAK[i]/(LOOK_UP_FAK[r]*LOOK_UP_FAK[u]*LOOK_UP_FAK[i-2*r-2*u])
        end;
  end;
end;

function nuclearAttractionIntegral(const pre: TGTFPreCalculatedNuclearAttraction;
                                   const p1, p2, kern: TVector3n): number;
var a_cache:array['x'..'z',0..4,0..2,0..2] of number;
    f_cache:array[0..12] of number;
    c:char;
    tau: number;
    i,r,u,j,s,v,k,t,w: longint;
    P: TVector3n;
begin
  P:=midpoint(pre.f.alpha,pre.g.alpha,p1,p2);
  for c:='x' to 'z' do
    for i:=0 to pre.f.e[c]+pre.g.e[c] do
      for r:=0 to i div 2 do
        for u:=0 to (i - 2*r) div 2 do begin
```

```
        a_cache[c,i,r,u]:=pre.a_cache[c,i,r,u]*
                          binomSum(i,pre.f.e[c],pre.g.e[c],P[c]-p1[c],P[c]-p2[c]);
        if (i-2*(r+u)<>0) or (P[c]-kern[c]<>0) then //TODO: Wirklich = 1
          a_cache[c,i,r,u]*=power((P[c]-kern[c]),i-2*(r+u));//TODO: vielleicht abs?
      end;

  //Vorberechnung aller F_v(t) Terme in f_cache
  tau:=(pre.f.alpha+pre.g.alpha)*veclensqr(vecsub(kern,P));
  for i:=0 to pre.coeffsum do
    f_cache[i]:=f_erf_like(i,tau);

  //Zusammenfügen der Caches
  result:=0;
  for i:=0 to pre.f.e['x']+pre.g.e['x'] do
   for r:=0 to i div 2 do
    for u:=0 to (i - 2*r) div 2 do
     for j:=0 to pre.f.e['y']+pre.g.e['y'] do
      for s:=0 to j div 2 do
       for v:=0 to (j - 2*s) div 2 do
        for k:=0 to pre.f.e['z']+pre.g.e['z'] do
         for t:=0 to k div 2 do
          for w:=0 to (k - 2*t) div 2 do
           result+=a_cache['x',i,r,u]*a_cache['y',j,s,v]*
                   a_cache['z',k,t,w]*f_cache[i+j+k-2*(r+s+t)-u-v-w];

  //Vorfaktoren
  with pre do
    result*=exp(-(f.alpha*g.alpha/(f.alpha+g.alpha)) *
            veclensqr(vecSub(p1,p2)))*factor;
end;

//Berechnet die durch die Abstoßung zwischen Elektronen gegebene Energie
function preCalculateElectronRepulsionIntegral(const factor: number; const f,
  g: TFreeGTF): TGTFPreCalculatedElectronRepulsion;
var c:char;
    i:longint;
    i1,i2,r1,r2,u:longint;
begin
  for c:='x' to 'z' do
    assert((f.e[c]<=1)and(g.e[c]<=1),
           'Orbitaltyp nicht erlaubt');
  result.f:=f;
  result.g:=g;
  result.alphasum:=f.alpha+g.alpha;
  result.factor:=factor/result.alphasum;
```

```
  result.coeffsum:=f.e['x']+g.e['x']+
                   f.e['y']+g.e['y']+
                   f.e['z']+g.e['z'];

  Result.a_power[0]:=1;
  for i:=-1 downto low(Result.a_power) do
    Result.a_power[i]:=Result.a_power[i+1]/result.alphasum;

  for c:='x' to 'z' do
   for i1:=0 to f.e[c]+g.e[c] do
    for i2:=0 to high(result.b_cache[c,i1]) do
     for r1:=0 to i1 div 2 do
      for r2:=0 to i2 div 2 do
       for u:=0 to (i1+i2)div 2-r1-r2 do with result do begin
         //Ort unabhängig
         if odd(i1+u) then b_cache[c,i1,i2,r1,r2,u]:=-1 //Änderung i2->i1 WARUM??
         else b_cache[c,i1,i2,r1,r2,u]:=1;
         b_cache[c,i1,i2,r1,r2,u]*=
           a_power[r1-i1]

           *LOOK_UP_FAK[i1]*LOOK_UP_FAK[i2]
           *LOOK_UP_2POWER[2*(r1+r2-i1-i2)]
           *LOOK_UP_FAK[i1+i2-2*(r1+r2)]
           /(LOOK_UP_FAK[r1]*LOOK_UP_FAK[r2]*
             LOOK_UP_FAK[i1-2*r1]*LOOK_UP_FAK[i2-2*r2]*
             LOOK_UP_FAK[u]*LOOK_UP_FAK[i1+i2-2*(r1+r2+u)]);
       end;
end;

function electronRepulsionIntegral(const pre12,pre34:
         TGTFPreCalculatedElectronRepulsion;
  const p1,p2,p3,p4:TVector3n):number;
var P12,P34: TVector3n;
    c:char;
    //Schleifenvariablen
    i,i1,i2,r1,r2,u,j1,j2,s1,s2,v,k1,k2,t1,t2,w:longint;
    //Vorberechnete Werte
    pq_power,hf12_cache,hf34_cache:array[0..4]of number;
    b_cache: array['x'..'z',0..4,0..4,0..2,0..2,0..4] of number;
    f_cache: array[0..24] of number;
    delta_power: array[-4..0] of number;
    tau,erf_sqrt_tau,pq,delta:number;
begin
  P12:=midpoint(pre12.f.alpha,pre12.g.alpha,p1,p2);
  P34:=midpoint(pre34.f.alpha,pre34.g.alpha,p3,p4);
```

```
tau:=pre12.alphasum*pre34.alphasum*veclensqr(vecsub(P12,P34))/
                                        (pre12.alphasum+pre34.alphasum);
if pre12.coeffsum+pre34.coeffsum=0 then result:=f_erf_like(0,tau)
else begin //Mindestens ein p-Orbital
  delta:=0.25*(1/pre12.alphasum+1/pre34.alphasum);
  delta_power[0]:=1;
  for i:=-1 downto low(delta_power) do
    delta_power[i]:=delta_power[i+1]/delta;

  erf_sqrt_tau:=erf(sqrt(tau));
  for i:=0 to pre12.coeffsum+pre34.coeffsum do
    f_cache[i]:=f_erf_like(i,tau,erf_sqrt_tau);
  for c:='x' to 'z' do begin
    for i:=0 to pre12.f.e[c]+pre12.g.e[c] do
      hf12_cache[i]:=binomSum(i,pre12.f.e[c],pre12.g.e[c],
                              P12[c]-p1[c],P12[c]-p2[c]);
    for i:=0 to pre34.f.e[c]+pre34.g.e[c] do
      hf34_cache[i]:=binomSum(i,pre34.f.e[c],pre34.g.e[c],
                              P34[c]-p3[c],P34[c]-p4[c]);
    pq:=P12[c]-P34[c];
    pq_power[0]:=1;
    for i:=1 to high(pq_power) do
      pq_power[i]:=pq_power[i-1]*pq;

    //B cache füllen
    for i1:=0 to pre12.f.e[c]+pre12.g.e[c] do
     for i2:=0 to pre34.f.e[c]+pre34.g.e[c]do
      for r1:=0 to i1 div 2 do
       for r2:=0 to i2 div 2 do
        for u:=0 to (i1+i2)div 2-r1-r2 do
          b_cache[c,i1,i2,r1,r2,u]:=pre12.b_cache[c,i1,i2,r1,r2,u]*
                                    pre34.a_power[r2-i2]*
            hf12_cache[i1]*hf34_cache[i2]*pq_power[i1+i2-2*(r1+r2+u)]*
            delta_power[2*(r1+r2)-i1-i2+u];
  end;

  //Caches multiplizieren
  Result:=0;
  for i1:=0 to pre12.f.e['x']+pre12.g.e['x'] do
  for i2:=0 to pre34.f.e['x']+pre34.g.e['x']do
  for r1:=0 to i1 div 2 do
  for r2:=0 to i2 div 2 do
  for u:=0 to (i1+i2)div 2-r1-r2 do
    for j1:=0 to pre12.f.e['y']+pre12.g.e['y'] do
    for j2:=0 to pre34.f.e['y']+pre34.g.e['y']do
```

```
      for s1:=0 to j1 div 2 do
      for s2:=0 to j2 div 2 do
      for v:=0 to (j1+j2)div 2−s1−s2 do
        for k1:=0 to pre12.f.e['z']+pre12.g.e['z'] do
        for k2:=0 to pre34.f.e['z']+pre34.g.e['z']do
        for t1:=0 to k1 div 2 do
        for t2:=0 to k2 div 2 do                                          410
        for w:=0 to (k1+k2)div 2−t1−t2 do
          result+=b_cache['x',i1,i2,r1,r2,u]*
                  b_cache['y',j1,j2,s1,s2,v]*
                  b_cache['z',k1,k2,t1,t2,w]*
                  f_cache[i1+i2+j1+j2+k1+k2
                          −2*(r1+r2+s1+s2+t1+t2)
                          −u−v−w];
  end;
  result*=pre12.factor*pre34.factor*2*pi*pi*sqrt(pi/(pre12.alphasum+pre34.alphasum));
  result*=exp(−pre12.f.alpha*pre12.g.alpha*veclensqr(vecsub(p1,p2))/pre12.alphasum   420
              −pre34.f.alpha*pre34.g.alpha*veclensqr(vecsub(p3,p4))/pre34.alphasum)
end;

end.
```

E.8 stosim.pas

```
{Dieser Programmteil enthält Routinen zur Berechnung von Slater Type Orbitals
 durch Approximation mit GTFs (STO3G-Basis).
 Die Orbitale werden dabei nicht einmal umgeformt, sondern erst bei Berechnung.
 Zusammen mit der teilweisen Vorberechnung dieser Integralen liefert dies
 einen kleinen Geschwindigkeitsvorteil.
}
unit stosim;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils,math,extmath,atoms,gtf;

type
  //Slatertypfunktion
  TSTO=record
    atom: TAtom; //Am Atom
    typ: longint; //(1: 1s; 2: 2s; 3: 2px; 4: 2py; 5: 2pz);
  end;

const QN: array[1..5,1..3] of longint= //Quantenzahlen von Orbitaltyp
      ((1,0,0),(2,0,0),(2,1,-1), (2,1,0), (2,1,-1));

//Standardmäßig verfügbare Orbitale je Atom (Alle bis zu 2p, außer bei H, da 1s)
const OrbitalCount:array[1..8] of longint=(1,0,0,0,0,5,5,5);

//Liefert eine STO zurück
function createSTO(const atom: TAtom; typ: longint): TSTO;
//Berechnet das vollständige Überlappungsintegral
function overlapIntegral(const f,g: TSTO): number;
//Berechnet das Integral der kinetischen Energie
function kineticEnergyIntegral(const f,g:TSTO):number;
//Berechnet die Energie der Anziehung zwischen Kern und Orbital
function nuclearAttractionIntegral(const f,g:TSTO; const atom: TAtom):number;
//Berechnet die durch die Abstoßung zwischen Elektronen gegebene Energie
function electronRepulsionIntegral(const f1,f2,f3,f4:TSTO):number;

var initializated:boolean=false;
//Berechnet die möglichen Integrale teilweise
procedure initUnit;
implementation
```

```
const
  //Slater Type Orbital Parameter (nach JCP38,2686)
  STO_Params: array[1..8] of array [1..3] of number=(
    //1s     2s       2p
    (1, nan, nan),          //H
    (nan, nan, nan),        //He
    (nan, nan, nan),        //Li
    (3.6848, 0.9560, nan),  //Be
    (nan, nan, nan),        //B
    (5.6727, 1.6083, 1.5679),//C
    (6.6651, 1.9237, 1.9170),//N
    (7.6579, 2.2458, 2.2266) //O
  );
  //STO zu GTO Konvertierungsfaktoren (nach Cook)
  STO3G: array[1..3,1..3,1..2] of number=
  (((0.154321,2.22766),(0.535328,0.40577),(0.444635,0.1098175)), //1s
   ((-0.0599447,2.58158),(0.596039,0.156762),(0.458179,0.0601833)), //2s
   ((0.162395,0.919238),(0.566171,0.235919),(0.422307,0.0800981))) //2p
  ;

var
  //Berechnete Parameter für die GTF Zerlegung
  STO3G_GTFPreFactors: array[1..8,1..3,1..3] of number;
  STO3G_GTF:array[1..8,1..5,1..3] of TFreeGTF;
  //Vorberechnete Integrale
  STO3G_PC_Overlap: array[1..8,1..5,1..3,1..8,1..5,1..3] of TGTFPreCalculatedOverlap;
  STO3G_PC_Kinetic: array[1..8,1..5,1..3,1..8,1..5,1..3] of TGTFPreCalculatedKinetic;
  STO3G_PC_Nuclear: array[1..8,1..5,1..3,1..8,1..5,1..3] of
                                              TGTFPreCalculatedNuclearAttraction;
  STO3G_PC_Repulsion: array[1..8,1..5,1..3,1..8,1..5,1..3] of
                                              TGTFPreCalculatedElectronRepulsion;
  //Integrale am gleichen Zentrum sind Positionsunabhängig
  STO3G_PC_SimpleOverlap,STO3G_PC_SimpleKinetic,
  STO3G_PC_SimpleNuclear:array[1..8,1..5,1..5] of number;
  //...s.r.[i,j,k,l]: i>=j, k>=l!
  STO3G_PC_SimpleRepulsion:array[1..8,1..5,1..5,1..5,1..5] of number;

function createSTO(const atom: TAtom; typ: longint): TSTO;
begin
  result.atom:=atom;
  result.typ:=typ;
end;

//Berechnet das vollständige Überlappungsintegral
//wenn beide Orbitale am selben Atom sind, wird das vorberechnete Ergebnis geliefert
```

```
//ansonsten die GTF Funktionen aufgerufen
function overlapIntegral(const f, g: TSTO): number;
var i,j:longint;
begin
  if f.atom=g.atom then exit(STO3G_PC_SimpleOverlap[f.atom.typ,f.typ,g.typ]);
  result:=0;
  for i:=1 to 3 do
    for j:=1 to 3 do
      result+=gtf.overlapIntegral(STO3G_PC_Overlap[f.atom.typ,f.typ,i,
                                                   g.atom.typ,g.typ,j],f.atom.p,g.atom.p);
end;

//Berechnet das Integral der kinetischen Energie
//wenn beide Orbitale am selben Atom sind, wird das vorberechnete Ergebnis geliefert
//ansonsten die GTF Funktionen aufgerufen
function kineticEnergyIntegral(const f, g: TSTO): number;
var i,j:longint;
begin
  if f.atom=g.atom then exit(STO3G_PC_SimpleKinetic[f.atom.typ,f.typ,g.typ]);
  result:=0;
  for i:=1 to 3 do
    for j:=1 to 3 do
      result+=gtf.kineticIntegral(STO3G_PC_Kinetic[f.atom.typ,f.typ,i,
                                  g.atom.typ,g.typ,j],f.atom.p,g.atom.p);
end;

//Berechnet die Energie der Anziehung zwischen Kern und Orbital
//wenn alle am Atom sind, wird das vorberechnete Ergebnis geliefert
//ansonsten die GTF Funktionen aufgerufen
function nuclearAttractionIntegral(const f, g: TSTO; const atom: TAtom): number;
var i,j:longint;
begin
  if (f.atom=g.atom) and (atom=f.atom) then
    exit(STO3G_PC_SimpleNuclear[f.atom.typ,f.typ,g.typ]);
  result:=0;
  for i:=1 to 3 do
    for j:=1 to 3 do
      result+=gtf.nuclearAttractionIntegral(
        STO3G_PC_Nuclear[f.atom.typ,f.typ,i,g.atom.typ,g.typ,j],
        f.atom.p,g.atom.p,atom.p);
  result*=atom.typ;
end;

//Berechnet die durch die Abstoßung zwischen Elektronen gegebene Energie
//wenn alle Orbitale am selben Atom sind, wird das vorberechnete Ergebnis geliefert
```

```
//ansonsten die GTF Funktionen aufgerufen
function electronRepulsionIntegral(const f1, f2, f3, f4: TSTO): number;
var i,j,k,l:longint;
begin
  if (f1.atom=f2.atom) and (f2.atom=f3.atom) and (f3.atom=f4.atom) then
    exit(STO3G_PC_SimpleRepulsion[f1.atom.typ,max(f1.typ,f2.typ),min(f1.typ,f2.typ),
                                  max(f3.typ,f4.typ),min(f3.typ,f4.typ)]);

  result:=0;
  for i:=1 to 3 do
   for j:=1 to 3 do
    for k:=1 to 3 do
     for l:=1 to 3 do
      result+=gtf.electronRepulsionIntegral(
        STO3G_PC_Repulsion[f1.atom.typ,f1.typ,i,f2.atom.typ,f2.typ,j],
        STO3G_PC_Repulsion[f3.atom.typ,f3.typ,k,f4.atom.typ,f4.typ,l],
        f1.atom.p,f2.atom.p,f3.atom.p,f4.atom.p);
end;

//Berechnet die möglichen Integrale teilweise
procedure initUnit;
const OX:array[1..5]of longint=(1,2,3,3,3); //Orbitalzuordnung
var i,j,k,l,m: longint;
    i1,i2,o1,o2,o3,o4,g1,g2:longint;
    f,g:TFreeGTF;
begin
  if initializated then exit;
  initializated:=true;
  //Berechnung der GTF Vorfaktorparameter
  for i:=1 to 8 do //Atomart
   for j:=1 to 3 do //Orbitalart(1s,2s,2p)
    for k:=1 to 3 do begin//Index der GTF
      if isnan(STO_Params[i,j]) then continue;
      STO3G_GTFPreFactors[i,j,k]:=STO3G[j,k,1]*
                                  normalizeFactorGTF(j div 3,0,0,
                                  STO3G[j,k,2]*sqr(STO_Params[i,j]));
    end;
  //Berechnet freie GTF Funktionen
  for i:=1 to 8 do
   for j:=1 to Orbit"alCount[i] do
    for k:=1 to 3 do begin
      STO3G_GTF[i,j,k].alpha:=STO3G[OX[j],k,2]*sqr(STO_Params[i,OX[j]]);
      STO3G_GTF[i,j,k].e:=nullvector3ic;
      if j=3 then STO3G_GTF[i,j,k].e['x']:=1;
      if j=4 then STO3G_GTF[i,j,k].e['y']:=1;
```

```
    if j=5 then STO3G_GTF[i,j,k].e['z']:=1;
  end;
//teilweise Vorberechnung der möglichen Integrale
for i1:=1 to 8 do //Atomart
 for o1:=1 to OrbitalCount[i1] do //Orbitalart(1s,2s,2px,2py,2pz)
  for g1:=1 to 3 do //GTF-Index
   for i2:=1 to 8 do
    for o2:=1 to OrbitalCount[i2] do
     for g2:=1 to 3 do begin
       STO3G_PC_Overlap[i1,o1,g1,i2,o2,g2]:=preCalculateOverlapIntegral(
          STO3G_GTFPreFactors[i1,OX[o1],g1]*STO3G_GTFPreFactors[i2,OX[o2],g2],
          STO3G_GTF[i1,o1,g1],STO3G_GTF[i2,o2,g2]);
       STO3G_PC_Kinetic[i1,o1,g1,i2,o2,g2]:=preCalculateKineticIntegral(
          STO3G_GTFPreFactors[i1,OX[o1],g1]*STO3G_GTFPreFactors[i2,OX[o2],g2],
          STO3G_GTF[i1,o1,g1],STO3G_GTF[i2,o2,g2]);
       STO3G_PC_Nuclear[i1,o1,g1,i2,o2,g2]:=preCalculateNuclearAttractionIntegral(
          STO3G_GTFPreFactors[i1,OX[o1],g1]*STO3G_GTFPreFactors[i2,OX[o2],g2],
          STO3G_GTF[i1,o1,g1],STO3G_GTF[i2,o2,g2]);
      STO3G_PC_Repulsion[i1,o1,g1,i2,o2,g2]:=preCalculateElectronRepulsionIntegral(
          STO3G_GTFPreFactors[i1,OX[o1],g1]*STO3G_GTFPreFactors[i2,OX[o2],g2],
          STO3G_GTF[i1,o1,g1],STO3G_GTF[i2,o2,g2]);
     end;
//vollständige Vorberechnungen aller Möglichkeiten an einem Atom
for i:=1 to 8 do
 for o1:=1 to OrbitalCount[i] do
  for o2:=1 to OrbitalCount[i] do begin
    STO3G_PC_SimpleOverlap[i,o1,o2]:=0;
    for j:=1 to 3 do for k:=1 to 3 do
      STO3G_PC_SimpleOverlap[i,o1,o2]+=gtf.overlapIntegral(
        STO3G_PC_Overlap[i,o1,j,i,o2,k],nullvector3n,nullvector3n);

    STO3G_PC_SimpleKinetic[i,o1,o2]:=0;
    for j:=1 to 3 do for k:=1 to 3 do
      STO3G_PC_SimpleKinetic[i,o1,o2]+=gtf.kineticIntegral(
        STO3G_PC_Kinetic[i,o1,j,i,o2,k],nullvector3n,nullvector3n);

    STO3G_PC_SimpleNuclear[i,o1,o2]:=0;
    for j:=1 to 3 do for k:=1 to 3 do
      STO3G_PC_SimpleNuclear[i,o1,o2]+=gtf.nuclearAttractionIntegral(
        STO3G_PC_Nuclear[i,o1,j,i,o2,k],nullvector3n,nullvector3n,nullvector3n);
    STO3G_PC_SimpleNuclear[i,o1,o2]*=i;

    for o3:=1 to o1 do
      for o4:=1 to o2 do begin
        STO3G_PC_SimpleRepulsion[i,o1,o3,o2,o4]:=0;
```

```
          for j:=1 to 3 do for k:=1 to 3 do
           for l:=1 to 3 do for m:=1 to 3 do
            STO3G_PC_SimpleRepulsion[i,o1,o3,o2,o4]+=gtf.electronRepulsionIntegral(
              STO3G_PC_Repulsion[i,o1,j,i,o3,k],STO3G_PC_Repulsion[i,o2,l,i,o4,m],
              nullvector3n,nullvector3n,nullvector3n,nullvector3n);
        end;
                                                                                  230
    end;
end;

initialization
  initUnit;
end.
```

E.9 molecules.pas

```
{Dieser Programmteil enthält alle wesentlichen Funktionen zur Veränderung von
 Molekülen und zur Berechnung ihrer Eigenschaften
 w
}
unit molecules;

{$mode objfpc}{$H+}



interface

uses
  Classes, SysUtils,windows,math,extmath,atoms,stosim;
type
  { TMolecule }
  //Ermöglicht Fortschrittsanzeige bei Energieberechnung
  TEnergyProgress=procedure(sender: tobject;energy:number;steps: longint;
                                    var cancel: boolean) of object;
  //DIE Molekülklasse
  TMolecule=class
  private
    orbitals:array of TSTO;          //vorhandenen AO-Orbitale
    elektrons: longint;              //Anzahl Elektronen
    //Temporäre Speicher für die SCF Energieberechnung
    _overlapSymMat:TSymMatrix;       //Überlappungsintegrale
    _orthoOverlapSymMat:TSymMatrix; // " hoch -0.5
    _oneElectronEnergySymMat:TSymMatrix; //kinetische+Kernenergie
    _coefMat:TMatrix;          //Koeffizienten i-te MO = Summe (c * j-te AO) T
    _denseSymMat:TSymMatrix;         //Dichteverteilung
    _fockSymMat:TSymMatrix;          //Fockmatrix
    _tempMat1,_tempMat2: TMatrix;    //temporärer
    _repulsion: array of array of TMatrix;//Elektronenwechselwirkungen

    //Sicherungen der obrigen Matrizen
    _saveOverlap:TMatrix;
    _saveRepulsion:array of array of TMatrix;

    _energies,_tempVecN: tvectorn; //MO-Energie und temporär
    _tempVecI: TVectori;           //              "
    _tempVecB: TVectorb;           //              "
    _lastEnergies:array of tvectorn;//Die letzten bisher aufgetreten MO-Energien
    _sortpos: array of longint;    //für Sortieralgorithmus
```

```
    _sorted: array of boolean;    // "    "

    //Rät die Koeffizientenmatrix
    procedure guessCoefficients;
    //Berechnet den Schwerpunkt neu
    procedure geometryChanged();
  public
    center: TVector3n; //Schwerpunkt
    atoms: TList;      //Atomliste
    constructor create;
    destructor destroy;override;

    //Bewegt das Molekül in eine feste Lage
    procedure normalizeMove;
    //Initialisiert das SCF Verfahren(Orbitalberechnung und Speicherreservierung)
    procedure initMOSCFEnergyCalculation;
    //Berechnet die Energie mit SCF
    function calculateEnergy(oneAtomChange: longint=-1):number;
    //Optimiert die Molekülenergie bis SCF eine minimale liefert
    procedure GeoOptimize(maxsteps:longint=$FFFFFF;progress:TEnergyProgress=nil);
    //Optimiert das Molekül auf triviale Weise mit dem Tetraedermodell
    procedure GeoOptimizeTrivial;

    //Überprüft ob alle Schalen geschlossen sind
    function closedShell():boolean;

    function addAtom(const typ: longint):tatom; //Fügt ein Atom in die Liste ein
    function addAtom(const typ: longint;const x,y,z: number):tatom;//an Position
    procedure deleteAtom(a:Tatom); //Löscht ein Atom
    procedure deleteBonds(a:Tatom); //Löscht alle Bindungen am Atom
    procedure deleteBonds(a,b:Tatom); //Löscht alle Bindungen zwischen Atomen
    procedure addBond(a,b:tatom;const count:longint=1);//Fügt eine Bindung hinzu

    //Speichert und läd das Molekül
    procedure saveToFile(fn: string);
    procedure loadFromFile(fn: string);
end;

implementation
{ TMolecule }

constructor TMolecule.create;
begin
  atoms:=TList.create;
end;
```

```
destructor TMolecule.destroy;
var i:longint;
begin
  for i:=0 to atoms.count-1 do
    tatom(atoms[i]).free;
  atoms.free;

  inherited destroy;
end;

//Dreht das gesamte Molekül, so dass:
//   Atom 0 am Nullpunkt liegt
//   Atom 1 auf der x-Achse (y=0,z=0)
//   Atom 2 auf der xy-Ebene (z=0)
procedure TMolecule.normalizeMove;
var c:char;
    i:longint;
    phi,theta,r:number;
    rotMat:TMatrix;
begin
  //Bewege Atom 0 zu 0
  for c:='x' to 'z' do
    if tatom(atoms[0]).p[c]<>0 then
      for i:=atoms.count-1 downto 0 do
        tatom(atoms[i]).p[c]-=tatom(atoms[0]).p[c];
  //Rotiere Atom 1 auf x
  vec2sphere(tatom(atoms[1]).p,r,phi,theta);
  SetLength(rotMat,3,3);
  matmul(createRotationMat3(phi-pi/2,'y'),createRotationMat3(-theta,'z'),rotMat);
  for i:=0 to atoms.count-1 do
    tatom(atoms[i]).p:=matVecTrans(rotMat,tatom(atoms[i]).p);
  if atoms.count>2 then begin
    //Rotiere Atom 2 auf xy
    with tatom(atoms[2]) do
      rotMat:=createRotationMat3(-arctan2(p['z'],p['y']),'x');
    for i:=0 to atoms.count-1 do
      tatom(atoms[i]).p:=matVecTrans(rotMat,tatom(atoms[i]).p);
  end;

end;

//Rät eine einigermaßen gute Koeffizientenmatrix
procedure TMolecule.guessCoefficients;
var pos: longint; //Anzahl der bereits festgelegten Werten
```

```
  procedure addAOsToMO(const atom: tatom);inline;
  begin with atom do
    case typ of
      AT_H: _coefMat[pos][firstOrbital]:=1; //H Atome geben 1s in die Bindung
      else begin
        //Alle anderen Atome werden als sp3-Hybridisierung probiert
        //(sollte es nicht stimmen, ändert das SCF Verfahren dies).
        _coefMat[pos][firstOrbital+1]:=1;
        _coefMat[pos][firstOrbital+2]:=1;
        _coefMat[pos][firstOrbital+3]:=1;
        _coefMat[pos][firstOrbital+4]:=1;
      end;
    end;
  end;
var i,j,k: longint;
begin
  pos:=0;
  //Ein MO für jede Bindung
  for i:=0 to atoms.count - 1 do
    for j:=1 to TAtom(atoms[i]).james do
      if TAtom(atoms[i]).bond[j].a.index>TAtom(atoms[i]).index then
        for k:=1 to TAtom(atoms[i]).bond[j].count do begin
          materaseLine(_coefMat,pos);
          addAOsToMO(TAtom(atoms[i]));
          addAOsToMO(TAtom(atoms[i]).bond[j].a);
          pos+=1;
        end;
  //Übriggebliebene MO besetzten:je 1s für C,N,O + 1 MO für N und 2 MOs für O
  for i:=0 to atoms.count - 1 do begin
    if TAtom(atoms[i]).typ=AT_H then continue;
    materaseLine(_coefMat,pos);
    _coefMat[pos][TAtom(atoms[i]).firstOrbital]:=1;
    pos+=1;
    case TAtom(atoms[i]).typ of
      AT_N: begin
        //Probiere 2s
        materaseLine(_coefMat,pos);
        _coefMat[pos][TAtom(atoms[i]).firstOrbital+1]:=1;
        pos+=1;
      end;
      AT_O: begin
        //Probiere 2s
        materaseLine(_coefMat,pos);
        _coefMat[pos][TAtom(atoms[i]).firstOrbital+1]:=1;
        pos+=1;
```

```
        //und 2p-Verschmelzung
        materaseLine(_coefMat,pos);
        _coefMat[pos][TAtom(atoms[i]).firstOrbital+2]:=1;
        _coefMat[pos][TAtom(atoms[i]).firstOrbital+3]:=1;
        _coefMat[pos][TAtom(atoms[i]).firstOrbital+4]:=1;
        pos+=1;
      end;
    end;
  end;
  assert(2*pos=elektrons,'Geschätze MO-Anzahl ungültig: '+inttostr(pos));
end;

//Berechnet den Schwerpunkt neu
procedure TMolecule.geometryChanged();
var i:longint;
begin
  center:=nullvector3n;
  for i:=0 to atoms.count-1 do
    center:=vecadd(center,tatom(atoms[i]).p);
  vecscale(center,1/atoms.count);
end;

//Initialisiert das SCF Verfahren (Orbitalberechnung und Speicherreservierung)
procedure TMolecule.initMOSCFEnergyCalculation;
var i,j:longint;
begin
  stosim.initUnit;;
  //Orbitalberechnung über Auffruf der Funktionen von stosim
  SetLength(orbitals,0);
  elektrons:=0;
  for i:=0 to atoms.count-1 do
    with TAtom(atoms[i]) do begin
      firstOrbital:=length(orbitals);
      setlength(orbitals,length(orbitals)+OrbitalCount[typ]);
      for j:=1 to OrbitalCount[typ] do
        orbitals[firstOrbital+j-1]:=createSTO(tatom(atoms[i]),j);
      elektrons+=typ;
    end;
  //Speicherplatzreservierung
  setlength(_overlapSymMat,length(orbitals),length(orbitals));
  setlength(_orthoOverlapSymMat,length(orbitals),length(orbitals));
  setlength(_oneElectronEnergySymMat,length(orbitals),length(orbitals));
  setlength(_coefMat,length(orbitals),length(orbitals));
  setlength(_denseSymMat,length(orbitals),length(orbitals));
```

```
  setlength(_fockSymMat,length(orbitals),length(orbitals));
  setlength(_tempMat1,length(orbitals),length(orbitals));
  setlength(_tempMat2,length(orbitals),length(orbitals));
  setlength(_repulsion,length(orbitals),length(orbitals),
                       length(orbitals),length(orbitals));
  setlength(_energies,length(orbitals));
  setlength(_tempVecN,length(orbitals));                                    230
  setlength(_tempVecI,length(orbitals));
  setlength(_tempVecB,length(orbitals));
  setlength(_lastEnergies,3,length(orbitals));
  setlength(_sortpos,length(orbitals));
  setlength(_sorted,length(orbitals));
  SetLength(_saveOverlap,length(orbitals),length(orbitals));
  SetLength(_saveRepulsion,length(orbitals),length(orbitals),
                           length(orbitals),length(orbitals));
end;
                                                                            240
//Berechnet die Energie mit dem SCF-Verfahren
function TMolecule.calculateEnergy(oneAtomChange: longint=-1): number;
const PREC=1e-10;                    //Zielpräzision
var
  i,j,k,l:longint;                   //Schleifenvariablen
  energyDiff,energyGain,tempNumber: number;//Energiedifferenz zum vorherigen
  tempVecPointer: TVectorn;          //Zeiger auf Energiearray
  oneAtomFirstOrbital,oneAtomLastOrbital:longint;//Orbitalgrenzen vom Atom
begin
  //=========================Berechnen der Integrale==================
  if oneAtomChange>=0 then begin
    //Es wurde ein Atom angegeben, dass als einzigstes verändert wurde
    //Orbitale des Atoms suchen
    oneAtomFirstOrbital:=tatom(atoms[oneAtomChange]).firstOrbital;
    oneAtomLastOrbital:=oneAtomFirstOrbital+
                        OrbitalCount[tatom(atoms[oneAtomChange]).typ]-1;
    //Alle Über. und Elek. Integrale berechnen, in denen das Atom vorkommt
    //Überlappungsmatrix
    //wiederherstellen
    for i:=0 to high(orbitals) do                                           260
      for j:=i to high(orbitals) do
        _overlapSymMat[i,j]:=_saveOverlap[i,j];
    //berechnen
    for i:=0 to oneAtomLastOrbital do
      for j:=i to high(orbitals) do
        if (i>=oneAtomFirstOrbital) or
          ((j>=oneAtomFirstOrbital)and(j<=oneAtomLastOrbital)) then
        _overlapSymMat[i,j]:=overlapIntegral(orbitals[i],orbitals[j]);
```

```
  //Elektronenwechselwirkungen
  for i:=0 to oneAtomLastOrbital do
   for j:=i to high(orbitals) do
    for k:=i to high(orbitals) do
     for l:=k to high(orbitals) do
      if (i>=oneAtomFirstOrbital) or
         ((j>=oneAtomFirstOrbital) and (j<=oneAtomLastOrbital)) or
         ((k>=oneAtomFirstOrbital) and (k<=oneAtomLastOrbital)) or
         ((l>=oneAtomFirstOrbital) and (l<=oneAtomLastOrbital)) then begin
           _saveRepulsion[i,j,k,l]:=_repulsion[i,j,k,l];
           _repulsion[i,j,k,l]:=electronRepulsionIntegral(orbitals[i],
                                  orbitals[j],orbitals[k],orbitals[l]);
      end;
end else begin
  //Alle Atome wurden geändert (oder sind unbekannt)
  //Überlappungsintegrale
  for i:=0 to high(orbitals) do
    for j:=i to high(orbitals) do begin
      _overlapSymMat[i,j]:=overlapIntegral(orbitals[i],orbitals[j]);
      _saveOverlap[i,j]:=_overlapSymMat[i,j]; //sichern
    end;

  //Elektronenwechselwirkungsintegrale
  for i:=0 to high(_repulsion) do
   for j:=i to high(_repulsion[i]) do
    for k:=i to high(_repulsion[i,j]) do
     for l:=k to high(_repulsion[i,j,k]) do
       _repulsion[i,j,k,l]:=electronRepulsionIntegral(orbitals[i],orbitals[j],
                                  orbitals[k],orbitals[l]);

  //Koeffizienten raten
  guessCoefficients;
end;

//Kinetikintegrale immer berechnen
for i:=0 to high(orbitals) do
  for j:=i to high(orbitals) do
    _oneElectronEnergySymMat[i,j]:=kineticEnergyIntegral(orbitals[i],
                                  orbitals[j]);

//Kernanziehungsintegrale immer berechnen
for k:=0 to atoms.count-1 do
  for i:=0 to high(orbitals) do
```

```
    for j:=i to high(orbitals) do
      _oneElectronEnergySymMat[i,j]-=
        nuclearAttractionIntegral(orbitals[i],orbitals[j],tatom(atoms[k]));



//orthoOverlap = overlap^-0.5                                              320
//Achtung: overlap wird dabei zerstört (deshalb wurde es gesichert)
matinvsqrt_sym(_overlapSymMat,_tempMat1,_tempMat2,_tempVecB,_tempVecI, _tempVecN,
               _orthoOverlapSymMat);



//==========================Optimierung der MOs==================
//Löschen der letzten Energien
fillchar(_energies[0],sizeof(_energies[0])*length(_energies),0);
for i:=0 to high(_lastEnergies) do
  fillchar(_lastEnergies[i,0],                                           330
           sizeof(_lastEnergies[i,0])*length(_lastEnergies[i]),0);
repeat
  //Berechnung der Dichtematrix
  for i:=0 to high(orbitals) do
    for j:=i to high(orbitals) do begin
      _denseSymMat[i,j]:=0;
      for k:=0 to elektrons div 2-1 do
        _denseSymMat[i,j]+=_coefMat[k,i]*_coefMat[k,j];
    end;
                                                                           340
  //Berechnung der Fockmatrix
  for i:=0 to high(orbitals) do
    for j:=i to high(orbitals) do begin
      _fockSymMat[i,j]:=_oneElectronEnergySymMat[i,j];
      for k:=0 to i do begin
        for l:=k+1 to i do
          _fockSymMat[i,j]+=_denseSymMat[k,l]*(4*_repulsion[k,l,i,j]-
                                    _repulsion[k,i,l,j]-_repulsion[k,j,l,i]);
        for l:=i+1 to j do
          _fockSymMat[i,j]+=_denseSymMat[k,l]*(4*_repulsion[k,l,i,j]-   350
                                    _repulsion[k,i,l,j]-_repulsion[k,j,i,l]);
        for l:=j+1 to high(orbitals) do
          _fockSymMat[i,j]+=_denseSymMat[k,l]*(4*_repulsion[k,l,i,j]-
                                    _repulsion[k,i,j,l]-_repulsion[k,j,i,l]);
        _fockSymMat[i,j]+=_denseSymMat[k,k]*(2*_repulsion[k,k,i,j]-
                                        _repulsion[k,i,k,j]);
      end;
      for k:=i+1 to j do begin
```

```
      for l:=k+1 to j do
        _fockSymMat[i,j]+=_denseSymMat[k,l]*(4*_repulsion[i,j,k,l]-_repulsion[i,k,l,j]-_repulsion[i,l
      for l:=j+1 to high(orbitals) do
        _fockSymMat[i,j]+=_denseSymMat[k,l]*(4*_repulsion[i,j,k,l]-_repulsion[i,k,j,l]-_repulsion[i,l
      _fockSymMat[i,j]+=_denseSymMat[k,k]*(2*_repulsion[i,j,k,k]-_repulsion[i,k,k,j]);
    end;
    for k:=j+1 to high(orbitals) do begin
      for l:=k+1 to high(orbitals) do
        _fockSymMat[i,j]+=_denseSymMat[k,l]*(4*_repulsion[i,j,k,l]-
                        _repulsion[i,k,j,l]-_repulsion[i,l,j,k]);
      _fockSymMat[i,j]+=_denseSymMat[k,k]*(2*_repulsion[i,j,k,k]-
                                          _repulsion[i,k,j,k]);
    end;
  end;

  //Fockmatrix in eine ganze Matrix umwandeln zur Multiplikation
  //mit der OrthoOverlapMatrix (Ergebnis ist nicht symmetrisch)
  matsym2full(_fockSymMat);
  //Fockmatrix mal S^-0.5 in tempMat speichern
  matmul(_fockSymMat,_orthoOverlapSymMat,_tempMat1);
  //S^-0.5 mal tempMat in fock speichern
  matmul(_orthoOverlapSymMat,_tempMat1,_fockSymMat);

  //Berechnung der Eigenvektoren der Fockmatrix
  diag_sym(_fockSymMat,_tempVecB,_tempVecI,_tempVecN,_coefMat);
  //Verformung der Eigenverte mit S^-0.5 für endgültig richtige Werte
  matmul(_coefMat,_orthoOverlapSymMat,_tempMat1);

  //Sortieren, da nur die kleinsten elektrons/2 zählen
  for i:=0 to high(_tempVecN) do
    _sortpos[i]:=i;
  for i:=1 to high(_tempVecN) do
    for j:=i-1 downto 0 do
      if _tempVecN[_sortpos[j]]>_tempVecN[_sortpos[j+1]] then begin
        k:=_sortpos[j];
        _sortpos[j]:=_sortpos[j+1];
        _sortpos[j+1]:=k;
      end;
  for i:=0 to elektrons div 2-1 do begin
    _energies[i]:=_tempVecN[_sortpos[i]];
    move(_tempMat1[_sortpos[i],0],_coefMat[i,0],
         length(_coefMat[i])*sizeof(_coefMat[i,0]));
  end;

  //Geringste Energiedifferenz zu den letzten suchen
```

```
  energyDiff:=9999999999;
  for i:=0 to high(_lastEnergies) do
    if _lastEnergies[i]<>nil then begin
      energyDiff:=0;
      for j:=0 to elektrons div 2-1 do
        energyDiff+=abs(_energies[j]-_lastEnergies[i][j]);
      if energyDiff<PREC then break;                                    410
    end;

  //lastenergies kopieren und sortieren
  tempVecPointer:=_lastEnergies[0];
  for i:=0 to high(_lastEnergies)-1 do
    _lastEnergies[i]:=_lastEnergies[i+1];
  _lastEnergies[high(_lastEnergies)]:=tempVecPointer;
  move(_energies[0],_lastEnergies[high(_lastenergies)][0],
       sizeof(_energies[0])*length(_energies));
until energyDiff<PREC;                                                  420

//Elektronen-Energie berechnen
result:=0;
for i:=0 to high(orbitals) do
  for j:=i to high(orbitals) do begin
    energyGain:=_oneElectronEnergySymMat[i,j];
    for k:=0 to high(orbitals) do
      for l:=0 to high(orbitals) do begin
        if min(k,l)<i then tempNumber:=_repulsion[min(k,l),max(l,k),i,j]
        else tempNumber:=_repulsion[i,j,min(k,l),max(l,k)];             430
        if min(i,k)<min(j,l) then
          tempNumber-=0.5*_repulsion[min(i,k),max(i,k),min(j,l),max(j,l)]
        else tempNumber-=0.5*_repulsion[min(j,l),max(j,l),min(i,k),max(i,k)];
        energyGain+=_denseSymMat[min(k,l),max(k,l)]*tempNumber;
      end;
    if i<>j then energyGain*=2;
    result+=2*_denseSymMat[i,j]*energyGain;
  end;

//Kern-Kern-Energie berechnen                                           440
for i:=0 to atoms.count-1 do
  for j:=0 to i-1 do
    result+=tatom(atoms[i]).typ*tatom(atoms[j]).typ/vecdist(tatom(atoms[i]).p,
                                                            tatom(atoms[j]).p);

//Wiederherstellen der Sicherung
if oneAtomChange>=0 then
  for i:=0 to oneAtomLastOrbital do
```

```
      for j:=i to high(orbitals) do
       for k:=i to high(orbitals) do
        for l:=k to high(orbitals) do
         if (i>=oneAtomFirstOrbital) or
            ((j>=oneAtomFirstOrbital) and (j<=oneAtomLastOrbital)) or
            ((k>=oneAtomFirstOrbital) and (k<=oneAtomLastOrbital)) or
            ((l>=oneAtomFirstOrbital) and (l<=oneAtomLastOrbital)) then
              _repulsion[i,j,k,l]:=_saveRepulsion[i,j,k,l];
end;

//Molekülgeometrieoptimierung mit sharpest descent und SCF
procedure TMolecule.GeoOptimize(maxsteps:longint=$FFFFFF;
                                progress:TEnergyProgress=nil);
const difstep=0.1; //Schritt beim Ableiten
      movstep=0.1; //Schritt beim Bewegen
var gradient: array of TVector3n; //Berechnete Ableitung
    c:char; i:longint;            //Schleifenvariablen
    energy,oldenergy:number;      //Aktuelle Energie und letzte
    gradDiff,old: number;         //Differenz der Ableitung
    cancel: boolean;              //Abbrechen
begin
  //Initialisieren
  energy:=9999999999;
  SetLength(gradient,atoms.count);
  fillchar(gradient[0],sizeof(gradient[0])*sizeof(gradient),0);
  cancel:=false;
  //Aktuelle Energie berechnen
  energy:=calculateEnergy();
  //Erste Atome auf xy-Ebene bewegen
  normalizeMove;
  //Optimierungsverfahren
  repeat
   gradDiff:=0;
   //Ableitung numerisch für alle Koordinaten berechnen
   for i:=1 to atoms.count-1 do
     for c:='x' to 'z' do begin
       if ord(c)-ord('x') >= i then break;
       old:=gradient[i][c];
       tatom(atoms[i]).p[c]+=difstep;
       gradient[i][c]:=calculateEnergy(i);
       tatom(atoms[i]).p[c]-=2*difstep;
       gradient[i][c]-=calculateEnergy(i);
       tatom(atoms[i]).p[c]+=difstep;
       gradient[i][c]*=0.5*(movstep/difstep);
       //Differenz speichern
```

```
        gradDiff+=sqr(old-gradient[i][c]);
        //Große Sprünge vermeiden
        if gradient[i][c]>2 then gradient[i][c]:=2
        else if gradient[i][c]<-2 then gradient[i][c]:=-2;
      end;
    //Abbrechen bei kleiner Differenz
    if gradDiff<sqr(0.0001) then break;
    for i:=1 to atoms.count-1 do
      for c:='x' to 'z' do
        tatom(atoms[i]).p[c]-=gradient[i][c];
    //Energie berechnen
    oldenergy:=energy;
    energy:=calculateEnergy();
    //Fortschrittsfunktion aufrufen
    if assigned(progress) then begin
      progress(self,energy,maxsteps,cancel);
      if cancel then break;
    end;
    dec(maxsteps)
  until (abs(oldenergy-energy)<1e-10)or(maxsteps<=0);
  geometryChanged();
end;

//Tetraedermodelloptimierung
procedure TMolecule.GeoOptimizeTrivial;
type TTetraeder=array[1..4] of TVector3n;
  //Berechnung einer Bindungslänge
  //nach http://www-lehre.inf.uos.de/~okrone/DIP/node13.html
  function bondLen(const t1,t2:longint;const n: longint):number;
  const BOND_SUB:array[1..3] of number=(0,0.397,0.642);
  var SchomakerStevenson: number;
  begin
    SchomakerStevenson:=0.1511781;
    if (t1>10) or (t2>10) then SchomakerStevenson:=0.038;
    result:=CovalentRadius[t1]+CovalentRadius[t2]+
            SchomakerStevenson*abs(EN[t1]-EN[t2])-
            BOND_SUB[n];
  end;
  //Tetraeder spiegeln um Eben mit Normale n
  function mirrorTetraeder(const tetra: TTetraeder; const n: TVector3n):
                                                          TTetraeder;
  var i:longint;
  begin
    for i:=1 to 4 do
      result[i]:=mirrorVec(tetra[i],n);
```

```
  end;
  //Tetraeder um Achse axe um a Grad spiegeln
  function rotateTetraeder(const tetra: TTetraeder; const axe: TVector3n;
                           const a: number): TTetraeder;
  var i:longint;
      rm: TMatrix;
  begin
    rm:=createRotationMat3(a,axe);
    for i:=1 to 4 do
      result[i]:=matVecTrans(rm,tetra[i]);

  end;
  //Sortiert die Ecken so um, dass die n-Ecken nach (inklusive) e
  //hinten sind
  function hideEdge(const tetra: TTetraeder; const e,n:longint): TTetraeder;
  var i:longint;
  begin
    if e+n>4 then exit(tetra); //sind bereits hinten
    for i:=e to e+n-1 do
      result[i-e+5-n]:=tetra[i];
    for i:=1 to e-1 do
      result[i]:=tetra[i];
    for i:=e+n to 4 do
      result[i-n]:=tetra[i];
  end;

var arranged: array of boolean; //Hash bereits positionierter Atome
  //Atom a anordnen
  procedure arrange(a: tatom; tetra: TTetraeder);
  var i,j,e:longint;
      temp: TVector3n; //Richtung zum nächsten Kern
      b:^TBonds;
  begin
    e:=1;
    b:=@a.bond;
    for i:=1 to a.james do //Für alle Bindungen
      if not arranged[b^[i].a.index] then begin //wenn Atom nicht positioniert
        arranged[b^[i].a.index]:=true; //ist es das gleich
        case b^[i].count of
          1:  begin //Ecke an Ecke
            //Atom zur nächsten Ecke
            b^[i].a.p:=vecadd(a.p,
                              vecscale(bondLen(a.typ,b^[i].a.typ,1),tetra[e]));
            //Rekursiv fortsetzen mit gespiegeltem und gedrehten Tetraeder
            arrange(b^[i].a,hideEdge(mirrorTetraeder(
```

```
                    rotateTetraeder(tetra,tetra[e],pi),tetra[e]),e,1));;
          end;
          2:  begin //Kante an Kante
            //Atom zwischen die nächsten beiden Ecken
            temp:=vecadd(tetra[e],tetra[e+1]);
            vecnormalize(temp);
            b^[i].a.p:=vecadd(a.p,vecscale(bondLen(a.typ,b^[i].a.typ,2),temp));
            //Tetraeder an temp spiegel
            arrange(b^[i].a,hideEdge(mirrorTetraeder(tetra,temp),e,2));;;
          end;
          3: begin //Seite an Seite
            //Atom zwischen drei Ecken
            temp:=vecadd(vecadd(tetra[e],tetra[e+1]),tetra[e+2]);
            vecnormalize(temp);
            //Tetraeder an temp spiegel
            b^[i].a.p:=vecadd(a.p,vecscale(bondLen(a.typ,b^[i].a.typ,3),temp));
            arrange(b^[i].a,hideEdge(mirrorTetraeder(tetra,temp),e,3));;;
          end;
        end;
        e+=b^[i].count;
      end;
  end;
var tetra: TTetraeder;
begin
  if atoms.count=0 then exit;
 // if not closedShell then exit;
  setlength(arranged,atoms.count);
  FillChar(arranged[0],length(arranged)*sizeof(arranged[0]),0);
  arranged[0]:=true;
  tatom(atoms[0]).p:=nullvector3n;
  //normalisierter Ausgangstetraeder
  tetra[1]['x']:=1/sqrt(3);
  tetra[1]['y']:=1/sqrt(3);
  tetra[1]['z']:=-1/sqrt(3);

  tetra[2]['x']:=-1/sqrt(3);
  tetra[2]['y']:=-1/sqrt(3);
  tetra[2]['z']:=-1/sqrt(3);

  tetra[3]['x']:=-1/sqrt(3);
  tetra[3]['y']:=1/sqrt(3);
  tetra[3]['z']:=1/sqrt(3);

  tetra[4]['x']:=1/sqrt(3);
  tetra[4]['y']:=-1/sqrt(3);
```

```
  tetra[4]['z']:=1/sqrt(3);
  //Erstes Atom positionieren
  arrange(tatom(atoms[0]),tetra);
  geometryChanged;
end;

//Überprüft ob alle Schalen geschlossen sind
function TMolecule.closedShell(): boolean;
var i,j,t:longint;
begin
  result:=true;
  for i:=0 to atoms.count-1 do begin
    t:=ATOM_CLOSED_BONDS[tatom(atoms[i]).typ];
    for j:=1 to tatom(atoms[i]).james do
      t-=tatom(atoms[i]).bond[j].count;
    if t<>0 then exit(false);
  end;
end;

//Fügt ein Atom hinzu
function TMolecule.addAtom(const typ: longint): tatom;
begin
  //Zufallsposition suchen und Zwilling rufen
  Randomize;
  result:=addAtom(typ,random(10)-5,random(10)-5,random(10)-5);
end;

//Fügt ein Atom hinzu
function TMolecule.addAtom(const typ: longint; const x, y, z: number): tatom;
begin
  //Atom erzeugen
  result:=tatom.create;
  //Werte einfüllen
  result.index:=atoms.count;
  Result.typ:=typ;
  result.p['x']:=x;
  result.p['y']:=y;
  result.p['z']:=z;
  //speichern
  atoms.add(result);
  geometryChanged;
end;

//Atom löschen
procedure TMolecule.deleteAtom(a: Tatom);
```

```
var i:longint;
begin
  //Alle Bindungen löschen
  for i:=a.james downto 1 do
    deleteBonds(a,a.bond[i].a);
  //Atom entfernen
  atoms.delete(a.index);
  a.free;
  //Alle Indizes aktualisieren
  for i:=0 to atoms.count-1 do
    tatom(atoms[i]).index:=i;
  geometryChanged;
end;

//Bindungen löschen
procedure TMolecule.deleteBonds(a: Tatom);
var i:longint;
begin
  for i:=a.james downto 1 do
    deleteBonds(a,a.bond[i].a); //Alle Bindungen einzeln löschen
end;

//Bindung löschen
procedure TMolecule.deleteBonds(a, b: Tatom);
var i:longint;
begin
  //Bindung bei a suchen und löschen
  for i:=a.james downto 1 do
    if a.bond[i].a=b then begin
      if i<>high(b.bond) then
        move(a.bond[i+1],a.bond[i],sizeof(a.bond[1])*(a.james-i));
      a.james-=1;
    end;
  //Bindung bei b suchen und löschen
  for i:=b.james downto 1 do
    if b.bond[i].a=a then begin
      if i<>high(b.bond) then
        move(b.bond[i+1],b.bond[i],sizeof(b.bond[1])*(b.james-i));
      b.james-=1;
    end;
end;

//Bindung hinzufügen
procedure TMolecule.addBond(a, b: tatom;const count:longint=1);
var i,j:longint;
```

```
begin
  //Suche nach vorhandener Bindungen
  for i:=1 to a.james do
    if a.bond[i].a=b then begin
      a.bond[i].count+=count;
      for j:=1 to b.james do
        if b.bond[j].a=a then
          b.bond[j].count+=count;
      exit;
    end;
  //Sonst neue hinzufügen
  a.james+=1;
  b.james+=1;
  a.bond[a.james].a:=b;
  a.bond[a.james].count:=count;
  b.bond[b.james].a:=a;
  b.bond[b.james].count:=count;
end;

//Alle Daten binär in eine Datei schreiben
procedure TMolecule.saveToFile(fn: string);
var f: TFileStream;
    i,j:longint;
begin
  f:=TFileStream.Create(fn,fmCreate);
  f.WriteDWord(atoms.count);
  for i:=0 to atoms.count-1 do begin
    f.WriteDWord(tatom(atoms[i]).index);
    f.WriteDWord(tatom(atoms[i]).typ);
    f.WriteDWord(tatom(atoms[i]).james);
    f.WriteBuffer(tatom(atoms[i]).p['x'],sizeof(tatom(atoms[i]).p));
    f.WriteDWord(tatom(atoms[i]).flatX);
    f.WriteDWord(tatom(atoms[i]).flatY);
    for j:=1 to tatom(atoms[i]).james do begin
      f.WriteDWord(tatom(atoms[i]).bond[j].a.index);
      f.WriteDWord(tatom(atoms[i]).bond[j].count);
    end;
  end;
  f.free;
end;

//Alle Daten binär aus einer Datei lesen
procedure TMolecule.loadFromFile(fn: string);
var f: TFileStream;
    c,i,j:longint;
```

```
begin
  for i:=0 to atoms.count-1 do
    tatom(atoms[i]).free;
  atoms.clear;
  f:=TFileStream.Create(fn,fmOpenRead);
  atoms.count:=f.ReadDWord;
  for i:=0 to atoms.count-1 do                                          770
    atoms[i]:=tatom.create;
  for i:=0 to atoms.count-1 do begin
    tatom(atoms[i]).index:=f.ReadDWord;
    tatom(atoms[i]).typ:=f.ReadDWord;
    tatom(atoms[i]).james:=f.ReadDWord;
    f.ReadBuffer(tatom(atoms[i]).p['x'],sizeof(tatom(atoms[i]).p));
    tatom(atoms[i]).flatX:=f.ReadDWord;
    tatom(atoms[i]).flatY:=f.ReadDWord;
    for j:=1 to tatom(atoms[i]).james do begin
      tatom(atoms[i]).bond[j].a:=tatom(atoms[f.ReadDWord]);             780
      tatom(atoms[i]).bond[j].count:=f.ReadDWord;
    end;
  end;
  f.free;
  geometryChanged();
end;


end.
                                                                        790
```

Literatur

[BV04] Boyd, Stephen ; Vandenberghe, Lieven: *Convex Optimization*. Cambrige University Press, 2004

[Coo74] Cook, D. B.: *Ab Initio Valence Calculation in Chemistry*. Butterworths, 1974

[ED63] E.Clementi ; D.L.Raimondi: Atomic Screening Constants from SCF Functions. In: *Journal of chemical physics* 38 (1963), S. 2686

[EFK+03] Erbrecht, R. ; Felsch, M. ; König, H. ; Kricke, W. ; Martin, K. ; W.Pfeil ; Winter, R. ; Wörstenfeld, W.: *Das große Tafelwerk interaktiv*. Cornelsen, 2003

[FIZ07] FIZ CHEMIE Berlin. *ChemgaPedia, eine Enzyklopädie zur Chemie*. `http://www.chemgapedia.de`. 20. März 2007

[GY82] Ghélis, Charis ; Yon, Jeannine: *Protein Folding*. Academic Press, 1982

[HW94] Haken, Hermann ; Wolf, Hans: *Molekülphysik und Quantenchemie*. 2. Springer Verlag, 1994

[Kro03] Krone, Oliver. *Diplomarbeit: Webfähige interaktive 3D-Visualisierung von Proteinstrukturen: Die Bindungslänge*. `http://www-lehre.inf.uos.de/~okrone/DIP/node13.html`. 28. April 2003

[LKK82] Latscha, H.P. ; Kazmeier, U. ; Klein, H. A.: *Organische Chemie*. 5. Springer Verlag, 1982

[LNC94] Lehninger, A. L. ; Nelson, D.L. ; Cox, M.M.: *Prinzipien der Biochemie*. 2. Spektrum Akademischer Verlag, 1994

[Low78] Lowe, John P.: *Quantum Chemistry*. Academic Press, 1978

[Mic07] Michigan Tech Chemistry. *Cheminformatics B.S.* `http://www.chemistry.mtu.edu/pages/undergrad/cheminf/whatis.pdf`. 22. März 2007

[Pre95] Preuss, Heinzwerner: *Atomekerne und Elektronen*. Verlag Vieweg, 1995

[Rei94] Reinhold, Joachim: *Quantentheorie der Moleküle*. B.G.Teubner, 1994

[Ton07] Tonau, Christoph. *Computer-Animation*. 22. März 2007

[vU80] Čársky, Petr ; Urban, Miroslav: *Ab Initio Calculations*. Springer Verlag, 1980

[Win07] Winter, Mark. *WebElements*. 22. März 2007

[Zsc93] Zschunke, Adolf: *Molekülstruktur*. Spektrum Akademischer Verlag, 1993

Selbständigkeitserklärung

Hiermit bestätige ich das ich die Arbeit selbständig alleine durchgeführt habe und außer den angegebenen Quellen keine weiteren benutzt habe.