Erstellung eines Computerprogramms zur automatischen Erzeugung von Feynman-Graphen

Diplomarbeit
von
Ulrich Seul
Johannes–Gutenberg–Universität Mainz
Fachbereich Physik

12. Mai 2003

Inhaltsverzeichnis

Kapitel 1

Einleitung

Bei der theoretischen Untersuchung der Wechselwirkungen von physikalischen Elementarteilchen ist die Betrachtung von Feynman-Graphen von wesentlicher Bedeutung. Um einen physikalischen Prozess zu beschreiben, seien die Arten der einlaufenden und auslaufenden Elementarteilchen sowie deren Impulse gegeben. Die Aufgabe der theoretischen Physik bei einem solchen Prozess ist es, eine mathematische Vorhersage für die Größe zu liefern, die sich im Experiment messen lässt: Den Wirkungsquerschnitt. Wenn man für den Wirkungsquerschnitt mit den Mitteln der Quantenfeldtheorie einen Näherungswert gewinnen will, führt man eine Reihenentwicklung der Streumatrix nach der physikalischen Kopplungskonstante aus. Je nach dem gewünschten Genauigkeitsgrad der Störungsrechnung muss hierzu eine große Zahl von Feynman–Graphen berücksichtigt werden.

Ein Feynman–Graph oder Feynman–Diagramm beschreibt die Wechselwirkung von physikalischen Elementarteilchen in graphischer Form. Es besteht aus äußeren Linien, die zu den einlaufenden und auslaufenden Teilchen gehören, inneren Linien und Vertizes (Knoten). Jedes Diagramm steht für einen mathematischen Ausdruck, den man erhält, wenn man gemäß den Feynman–Regeln für die inneren Linien Teilchen–Propagatoren und für die Vertizes Wechselwirkungs–Faktoren miteinander multipliziert.

Je größer die gewünschte Genauigkeit der Näherung ist, desto mehr Feynman–Diagramme müssen berücksichtigt werden. Vor der eigentlichen Berechnung ist es als oft recht aufwändiger erster Schritt notwendig, alle erforderlichen Feynman–Graphen zu ermitteln.

Gegenstand dieser Arbeit ist die Entwicklung und Beschreibung eines Computerprogramms, das alle erforderlichen Feynman–Graphen automatisch erstellt und weiterhin auch gemäß den Feynman–Regeln in einen mathematischen Ausdruck übersetzt, der für die anschließenden Schritte der Berechnung durch den Computer benutzt werden kann.

Die Programmbibliothek *graphgenerator* ist in der Programmiersprache *C++* verfasst. Die mathematischen Ausdücke werden in der in Mainz entwickelten mathematischen *C++*–Bibliothek GiNaC [2] erstellt. *graphgenerator* wurde entwickelt, um einen Bestandteil des Mainzer *xloops*–Projekts zur automatischen Berechnung von Feynman–Graphen zu bilden.

Als Endergebnis von *graphgenerator* ist es möglich, sämtliche oder ausgewählte erzeugte Graphen auf dem Bildschirm auszugeben, in Gestalt einer Auflistung, welche Vertizes mit welchen Linien verbunden sind und welche Linien zu welchen Elementarteilchen gehören. Außerdem können die aus den Feynman–Regeln ermittelten Faktoren sowohl einzeln für die Linien und Vertizes als auch in korrekter Reihenfolge multipliziert angezeigt werden. Alle erzeugten Informationen können auch direkt als *C++*–Daten übergeben werden und vom Benutzer in eigenen Computerprogrammen weiterverwendet werden.

graphgenerator stellt bislang nur die Feynman–Regeln für das Standardmodell der elektroschwachen Wechselwirkung zur Verfügung.

Die Anzahl der *loops*, mit der Graphen erstellt werden können, ist prinzipiell unbegrenzt und nur den Beschränkungen von Rechenzeit und Speicherplatz des verwendeten Computermodells unterworfen.

Die Anzahl der äußeren Linien ist aus rein pragmatischen Gründen im Hinblick auf Tipparbeit derzeit auf sechs Linien begrenzt; sie kann mit minimalen Ergänzungen im Quelltext beliebig erhöht werden.

Kapitel 2

Einführung und Überblick

2.1 Einführung in den Algorithmus

Wie geht man vor, um Feynman–Graphen zu erzeugen?

Fast noch wichtiger ist die Frage: Wie kann man sich darauf beschränken, nur die gewünschten Feynman–Graphen zu erzeugen und nicht noch unendlich viele andere?

Wenn man vor dieser Frage steht, helfen zwei Informationen, mit denen der Benutzer seine Wünsche mitteilt.

Der Benutzer liefert erstens eine Liste von einlaufenden und auslaufenden Teilchen, die miteinander wechselwirken sollen. Das ist das Grundfundament, das alle Diagramme benutzen werden:

Bei allen Feynman–Graphen, die man erzeugen wird, werden die äußeren Linien genau gleich aussehen. Graphen mit anderen äußeren Linien werden nicht gebraucht und werden auch gar nicht erst erzeugt.

Trotzdem könnten unendlich viele Graphen gefunden werden, bei denen die vom Benutzer geforderten Elementarteilchen einlaufen und auslaufen. Ein Graph könnte schließlich prinzipiell unendlich viele Vertizes haben. Deshalb muss der Benutzer noch eine zweite Eingabe machen, die angibt, wie komplex die Graphen sein sollen.

Diese zweite Eingabe ist die Ordnung der Kopplungskonstante, oder anders ausgedrückt: Derjenige Exponent, mit dem die Kopplungskonstante potenziert wird, wenn alle Faktoren, die zu einem fertigen Graph gehören, bestimmt und aneinandermultipliziert sind und so den Beitrag dieses Diagramms zur Gesamtamplitude liefern.

Für die elektroschwache Wechselwirkung wird als Kopplungskonstante die Elementarladung e verwendet, die schwachen Mischungswinkel treten als zusätzliche Parameter auf.

An dieser Stelle wäre es vielleicht eine anschaulichere Wahl, als Eingabe die Anzahl der Schleifen zu verlangen. Für den Algorithmus erscheint es jedoch günstiger, mit der Ordnung der Kopplungskonstante zu arbeiten.

Nun kennt man die Ordnung der Kopplungskonstante. Bedeutet das, dass man nun weiß, wieviele Vertizes die Graphen haben sollen?

Leider nicht. Unendlich viele Vertizes sind zwar nicht mehr zulässig. Aber die physikalische Theorie, die für die Wechselwirkung der Teilchen zur Anwendung gebracht werden soll, kann Vertizes erlauben, die sich in wichtigen Punkten unterscheiden: Nicht nur darin, welche Teilchen sich an einem Vertex treffen sollen, sondern auch, wieviele Teilchen sich treffen und welche Ordnung der Kopplungskonstante für diesen Vertex gilt. Beispielsweise sind bei der elektroschwachen Wechselwirkung Vertizes mit drei, aber auch mit vier Teilchen möglich. Zu den Dreier–Vertizes gehört ein Faktor mit einfacher Kopplungskonstante, zu den Vierer–Vertizes ein Faktor mit quadrierter Kopplungskonstante.

Der Algorithmus für ein Computerprogramm sieht sich also vor folgende Aufgaben gestellt: Es müssen alle erlaubten Kombinationen von Vertex–Typen (im folgenden als topologische Modell–Vertizes bezeichnet) ermittelt werden, die zusammen zur richtigen Ordnung der Kopplungskonstante für den gesamten Graphen führen.

Beispiel: Der Benutzer hat einen elektroschwachen Prozess der Ordnung $o = 3$ verlangt, bei dem zwei Photonen einlaufen und ein Z–Boson auslaufen soll.

Diese Ordnung kann erreicht werden, indem man drei Dreier–Vertizes kombiniert. Aber genau so gut ist es möglich, einen Dreier–Vertex und einen Vierer–Vertex zu kombinieren.

Jede dieser Kombinationen von Modell–Vertizes muss im Folgenden getrennt betrachtet werden. Das bedeutet, das Programm verzweigt sich und bearbeitet alle verschiedenen Möglichkeiten unabhängig voneinander. Alle folgenden Schritte beschreiben nur noch den weiteren Weg einer einzigen Verzweigung. Dabei ist nun eine Kombination von Modell-Vertizes ausgewählt und wird nicht mehr verändert.

Was ist inzwischen über den Graph bekannt? Mittlerweile hat sich entschieden, wieviele Vertizes der Graph haben wird. Um das vorige Beispiel weiterzuführen, bearbeite das Programm im Folgenden eine Verzweigung, in der jeder Graph einen Dreier–Vertex und einen Vierer–Vertex umfassen soll. Damit ist ab hier entschieden, dass nur noch Graphen mit zwei Vertizes entstehen werden.

Parallel entstehen in der anderen Verzweigung Graphen mit drei Vertizes, von denen jeder drei Teilchen berührt.

Als nächstes müssen die Vertizes Nummern erhalten. Doch nicht nur sie,

sondern auch die Punkte im Unendlichen, die man sich am Beginn einer einlaufenden und am Ende einer auslaufenden Linie vorstellt. Alle Punkte, an denen Linien beginnen oder enden, fasst man mit den Begriffen Knoten oder *nodes* zusammen. Ein Knoten ist also entweder ein Vertex oder ein externer Punkt „im Unendlichen“.

In dem Beispiel erhalten ohne Beschränkung der Allgemeinheit die drei externen Knoten die Nummern $n = 0$, $n = 1$ und $n = 2$. Soll nun der Knoten $n = 3$ der Dreier– oder der Vierer–Vertex werden? Diese Entscheidung ist durchaus von Bedeutung, da später Graphen aussortiert werden müssen, die sich nur durch eine Permutation der Nummerierung unterscheiden, aber die selbe Topologie beschreiben. Es muss also eine neue Verzweigung des Programms stattfinden, so dass jede mögliche Nummerierung der Vertizes getrennt bearbeitet wird.

Nun stellt sich die Frage: Welcher Vertex ist mit welchem anderen Vertex bzw. mit welcher äußeren Linie verbunden? Das Konzept, nach dem an dieser Stelle verfahren wird, ist ganz einfach, alle Möglichkeiten durchzuspielen und in parallelen Verzweigungen jeden Knoten mit jedem anderen Knoten zu verbinden. Dabei können überflüssigen Graphen entstehen, die bei einer identischen Topologie nur die Vertizes anders nummeriert haben. Um sie auszusortieren, wird ein Verfahren verwendet, das von T.Kaneko beschrieben wurde [1].

Wenn diese Prozeduren fertig bearbeitet sind, liegt ein erstes Zwischenergebnis vor:

Alle gesuchten Topologien sind erstellt worden.

Kann es vorkommen, dass unter diesen Topologien einige sind, die nicht benötigt werden? Bisher hat der Algorithmus nur die Anzahl der einlaufenden und auslaufenden Teilchen und nicht ihre Namen berücksichtigt. Das heisst, dass zu allen Eingabedaten mit gleicher Ordnung und gleich vielen externen Teilchen eine exakt gleiche Menge aus Topologien erzeugt wird, unabhängig von der Art der externen Teilchen.

Folglich können durchaus Topologien erzeugt worden sein, zu denen keine Graphen möglich sind, weil zu den gegebenen externen Teilchen keine Vertizes mit passenden Teilchenkombinationen in den Feynman–Regeln existieren, um aus der Topologie einen korrekten Graph zu machen. Die überflüssigen Topologien werden aber nicht gelöscht, weil die Menge der Topologien unabhängig von den erst im nächsten Schritt betrachteten Teilchenarten bleiben soll.

Um aus den Topologien fertige Graphen zu machen, muss entschieden werden, welche Elementarteilchen den bisher noch leeren Linien zugeordnet

werden.

Dazu zieht das Programm die Feynman–Regeln hinzu und betrachtet alle möglichen Teilchenkombinationen, die an einem Vertex zulässig sind.

Wenn ein Vertex noch unvollständig ist, er also Linien berührt, die teilweise bereits mit Teilchen dekoriert wurden, teilweise aber noch nicht, wird geprüft, zu welchen Teilchenkombinationen aus den Feynman–Regeln dieser Vertex ergänzt werden könnte. Daraus ergibt sich eine Liste aller möglichen Teilchen, die hier als nächstes auf eine freie Linie „gelegt“ werden könnten.

An dieser Stelle verzweigt sich das Programm erneut. Jedes erlaubte Teilchen wird in einer eigenen Verzweigung an der freie Stelle hinzugefügt. Dann wird die nächste freie Stelle gesucht und so fort, bis alle Linien des Graphen mit Teilchen dekoriert sind.

Das Endergebnis sind alle Feynman–Graphen, die zu dem Prozess möglich sind, den der Benutzer in seinen Eingabedaten beschrieben hat.

2.2 Die verwendeten physikalischen Informationen

Wie bereits ersichtlich, verwendet die Programmbibliothek *graphgenerator* noch andere Daten außer den Namen der einlaufenden und auslaufenden Teilchen und der Ordnung der Kopplungskonstante. Das sind vor allem die Feynman–Regeln, die bereits für die Ermittlung der erlaubten Teilchenkombinationen an einem Vertex verwendet wurden und später auch die mathematischen Fakoren liefern werden, die den Linien und Vertizes zugeordnet werden.

graphgenerator stellt die Feynman–Regeln für die elektroschwache Wechselwirkung im Rahmen des Standardmodells der Elementarteilchenphysik zur Verfügung. Es wird die ’t Hooft–Eichung verwendet [5].

Das bedeutet, es treten hier nur Vertizes mit drei oder vier Teilchen auf. Das können entweder skalare Teilchen, Fermionen, Eichbosonen oder Faddeev–Popov–Geister sein. Wenn eine andere Konvention der Feynman–Regeln benutzt werden soll, kann das unschwer, wenn auch vielleicht zeitaufwändig in den im Quelltext dafür gekennzeichneten Routinen geändert werden.

Prinzipiell können die Feynman–Regeln, auf die *graphgenerator* zurückgreift, auch weitergehend verändert oder ersetzt werden. Eine naheliegende Ergänzung wären beispielsweise die Regeln für die Quantenchromodynamik, die die starke Wechselwirkung beschreiben. Wenn dabei keine zusätzlichen Elementarteilchen auftreten als diejenigen, deren Daten *graphgenerator*

für die elektroschwache Wechselwirkung zur Verfügung stellt (einschließlich Gluonen), reicht es zum Hinzufügen einer neuen Feynman-Regel aus, an der entsprechenden Stelle im Quelltext die Namen der beteiligten Teilchen zu identifizieren und den mathematischen Faktor anzugeben (im Code der *C++* –Bibliothek GiNaC).

Für die QCD muss zusätzlich bedacht werden, dass das Auftreten von Farbmatrizen zu veränderten Faktoren führt.

Für andere Theorien (etwa ϕ^6- Kopplung) ist es ohne weiteres möglich, auch Vertizes mit fünf, sechs oder sogar beliebig vielen (aber nicht unendlich vielen) beteiligten Teilchen zu erstellen.

Auf ein leicht durchführbares Verändern oder Ersetzen der vom Programm benutzten Daten über die beteiligten Elementarteilchen ist der Quelltext leider nicht vorbereitet. Mit entsprechend viel Aufwand und Sorgfalt wäre das aber prinzipiell machbar. Nicht angetastet werden sollte die Sonderstellung der Fermionen, deren Faktoren als einzige Dirac–Matrizen enthalten, die bekanntlich nicht kommutieren, weshalb ihre Anordnung im Graph vom Programm gesondert betrachtet wird.

Eingang in die Faktoren findet neben den (noch nicht als Zahlenwert eingesetzten) Massen auch Daten über den schwachen Isospin der Fermionen.

Kapitel 3

Beschreibung des Algorithmus zur Graphengenerierung

Die Erstellung und Bearbeitung der Feynman–Diagramme zu einem physikalischen Prozess, der durch die Eingabedaten festgelegt wird, erfolgt in drei Phasen:

- Es werden alle Topologien für den Prozess erzeugt. (Eine Topologie beschreibt die geometrische Gestalt eines Graphen. Den Linien sind noch keine Elementarteilchen zugeordnet.)
- Zu jeder Topologie werden alle dazu gehörenden Graphen durch Hinzufügen der Elementarteilchen erstellt.
- Ein Graph wird aus der Menge aller erzeugten Graphen ausgewählt. Die ihm zugeordneten mathematischen Faktoren können aus den Feynman–Regeln ermittelt und in korrekter Reihenfolge multipliziert werden.

Es ist möglich, nur die erste Phase allein auszuführen, oder auch nach den ersten beiden auf die dritte Phase zu verzichten.

Im folgenden werden die Algorithmen mehrerer zentraler Routinen des Programms erläutert.

Im Quelltext selbst werden dazwischen noch andere *C++*–Funktionen aufgerufen, auf die hier nicht eingegangen zu werden braucht, weil sie nicht wesentlich für die algorithmische Struktur sind.

Den Anfang macht die Routine *choose_vertex_types*. Sie erzeugt mehrere unterschiedliche Versionen für die Grundlage der Graphenkonstruktion. Jede dieser Versionen wird parallel in der Routine *place_vertex_types* weiterverarbeitet, die die Entwürfe noch weiter aufspaltet und an die Routine

connect_nodes übergibt, die daraus jeweils alle verschiedenen Möglichkeiten für die fertige topologische Gestalt des Graphen erstellt und speichert.

In Phase zwei greift die Routine *decorate_graph* auf eine ausgewählte, zuvor erzeugte Topologie zurück und bereitet die Auswahl vor, welches Teilchen als nächstes auf einer noch „freien“ Linie plaziert werden könnte. In der Routine *find_missing_particles* werden alle möglichen Teilchen alternativ an dieser Stelle hinzugefügt und so unterschiedliche Versionen des Graphen erzeugt. Jede davon wird wieder zurück an die Routine *decorate_graph* übergeben, die Kandidaten für das nächste Teilchen ermittelt, und so fort, bis jeder der unterschiedlichen Graphen zur gleichen Topologie fertiggestellt ist.

3.1 Eingabedaten

Über die verwendete physikalische Theorie werden folgende Informationen zur Verfügung gestellt:

- Informationen über die auftretenden Elementarteilchen. Hier müssen nur in dem Fall die im Programm vorgegebenen Daten ergänzt werden, wenn andere Elementarteilchen auftreten als im Standardmodell der Elementarteilchenphysik.
- Eine Liste aller möglichen Modell–Vertizes. Jeder Modell–Vertex ist gekennzeichnet durch die beteiligten Teilchen sowie Art und Potenz der Kopplungskonstante.
- Die mathematischen Faktoren der Feynman–Regeln für alle Modell–Vertizes und die Teilchen–Propagatoren.

Diese Daten stehen für die elektroschwache Kopplung im Standardmodell der Elementarteilchenphysik im Quelltext bereits zur Verfügung.

Der konkrete physikalische Prozess, für den die gesuchten Feynman–Graphen erstellt werden sollen, wird durch folgende Eingabedaten beschrieben:

- Die einlaufenden und auslaufenden Teilchen.
- Die Ordnung der Kopplungskonstante bzw. der Kopplungskonstanten für den gesamten Graphen.

(Als Schwerpunkt der Anwendung, für welchen der Großteil der Tests durchgeführt wurde, wird erwartet, dass nur die elektroschwache Kopplung allein auftritt. Möglich sind aber durchaus auch beliebige weitere Kopplungsarten, etwa die der Quantenchromodynamik.)

3.2 Erzeugung von Topologien

3.2.1 Überblick

Als erster Schritt der Graphengenerierung werden Topologien erzeugt, d.h. Graphen, die die topologische Anordnung der Linien und Vertizes zeigen, bei denen den Linien aber noch keine Teilchen zugeordnet sind.

Im folgenden werden nach der von T. Kaneko gewählten Bezeichnungsweise [1] die Begriffe *node* und *edge* verwendet. Ein *node* ist entweder ein Vertex oder ein im Unendlichen liegender Ursprungspunkt einer externen Linie. Ein *edge* ist eine innere oder äußere Linie.

Um alle Topologien zu erzeugen, die gemäß den Eingabedaten berücksichtigt werden müssen, werden verallgemeinerte Modell–Vertizes betrachtet, die im folgenden als topologische Modell–Vertizes bezeichnet werden. Ein topologischer Modell–Vertex enthält keine Daten über die Art der beteiligten Elementarteilchen mehr. Gekennzeichnet ist er stattdessen

- durch die Anzahl der Teilchen, die den Vertex bilden, d.h., wieviele Linien an diesem Vertex zusammenlaufen (im Folgenden als Anzahl der *legs* bezeichnet);
- durch die Art und Ordnung der Kopplungskonstante in der Feynman–Regel für diesen Vertex.

Um gößtmögliche Allgemeinheit zu gewährleisten, wird nicht vorausgesetzt, dass sich die Anzahl der *legs* aus der Ordnung der Kopplungskonstante unmittelbar ableiten lässt.

In der ersten Phase der Graphengenerierung wird nun untersucht, durch welche Kombinationen von topologischen Modell–Vertizes die in den Eingabedaten verlangte Ordnung der Kopplungskonstante erreicht werden kann. Zum Beispiel kann für die elektroschwache Kopplung die Ordnung $o = 3$ unter anderem durch die Kombination von einem Vertex mit drei *legs* und einem Vertex mit vier *legs* erreicht werden.

3.2.2 Routine *choose_vertex_types*

Diese Routine bereitet die Konstruktion des Graphen vor. Es wird überprüft, wieviele Linien sich in einem Vertex treffen können. Für jeden dieser Fälle, z.B. Dreier– oder Vierer–Vertizes, wird festgelegt, wieviele Vertizes von dieser Sorte im fertigen Graph vorkommen sollen. Es wird auch berücksichtigt, dass z.B. Dreier–Vertizes, die zu verschiedenen Kopplungskonstanten gehören, unterschieden werden müssen.

Die Funktion *coupling_order* ordnet jeder beteiligten Kopplungskonstante den Wert ihres Exponenten für den gesamten Graph zu.

Aus den Eingabedaten werden die Informationen über alle erlaubten topologischen Modell-Vertizes erstellt. Zusammengefasst werden sie in der Menge *vertex_types*. (Für die elektroschwache Kopplung ist das beispielsweise eine Menge mit nur zwei Elementen: Dem Vertex mit drei und dem mit vier Linien.)

Für jedes Element von *vertex_types* gibt es in der Regel mehrere Möglichkeiten, wie oft der jeweilige topologische Modell–Vertex im fertigen Graph tatsächlich vorkommen soll. Diese Anzahl wird ebenfalls in *vertex_types* gespeichert. Die verschiedenen Möglichkeiten müssen alle in parallelen Verzweigungen des Programms durchgespielt werden und führen zu verschiedenen Graphentwürfen.

Wenn die Zahlenwerte h für die Häufigkeit jedes einzelnen Elements von *vertex_types* ausgewählt sind, kann daraus die Ordnung jeder einzelnen Kopplungskonstante für den kompletten Diagrammentwurf errechnet werden.

Es gilt nun, alle möglichen Versionen von *vertex_types* zu ermitteln, aus denen sich die in *coupling_order* festgelegten Werte für die jeweiligen Kopplungskonstanten ergeben. Jede einzelne Version wird in einer eigenen Verzweigung des Programms zur Grundlage eines darauf aufbauenden Diagramm–Entwurfs gemacht.

Konkret umfasst der Algorithmus folgende Schritte:

- Schritt 1:

  In dem *C++*–Vektor *coupling_order* wird jeder vorkommenden Kopplungskonstante (in diesem Vektor nicht benannt, sondern nur nummeriert) der Wert ihrer Ordnung zugeordnet. Die Elemente des *C++*–Vektors *vertex_types* sind Datenstrukturen, die die Informationen über die topologischen Modell-Vertizes zusammenfassen.

  Zusätzlich ist jedem Element *vertex_types*$[v]$ auch ein Zahlenwert $h[v]$ zugeordnet, der angibt, wie viele konkrete Vertizes von der Art dieses verallgemeinerten Typs später im fertigen Graph enthalten sein sollen. *Vertex_types* ist nach Kopplungsarten geordnet, und zwar in der selben Reihenfolge wie *coupling_order*.

- Schritt 2:

  Aus den Elementen von *vertex_types*, die noch keine Häufigkeit $h[v]$ zugeordnet bekommen haben, wird dasjenige Element *vertex_types*$[v]$ mit der höchsten Nummer v ausgewählt.

Wenn festgestellt wird, dass jedes Element *vertex_types*$[v]$ eine Häufigkeit $h[v]$ zugeordnet bekommen hat, die besagt, dass von diesem topologischen Modell–Vertex genau $h[v]$ Vertreter dem Graphen hinzugefügt wurden (wobei jeweils auch die Häufigkeit $h[v] = 0$ erlaubt ist) und sich daraus in der Tat die verlangten Ordnungen der Kopplungskonstanten ergeben, wird mit diesen Versionen von *vertex_types* und h die anschließende Routine *place_vertex_types* begonnen, d.h. es wird Schritt 6 ausgeführt.

- Schritt 3:

  Von den Kopplungskonstanten, welche durch die bisher für den Graphen ausgewählten Vertizes noch nicht die Ordnung erreicht haben, die in *coupling_order* verlangt wird, wird diejenige mit der höchsten internen Nummerierung ausgewählt.

- Schritt 4:

  Für diese Kopplungskonstante wird die Differenz der insgesamt verlangten und durch die bisher ausgewählten topologischen Modell–Vertizes bereits erreichten Ordnung errechnet. Durch Quotientenbildung wird die maximale Häufigkeit n ermittelt, mit der *vertex_types*$[v]$ dem Diagramm hinzugefügt werden kann, ohne dass die Kopplung zu häufig auftritt.

- Schritt 5:

  Das Programm verzweigt sich: *vertex_types*$[v]$ wird alternativ in allen Häufigkeiten von 0 bis n dem Graphen hinzugefügt, d.h. der jeweilige Häufigkeitswert wird in $h[v]$ gespeichert und mit diesen verschiedenen Versionen von h wird jeweils als nächstes wieder Schritt 2 ausgeführt. Falls in einem Fall dadurch schon die erforderliche Ordnung der Kopplungskonstante erreicht wurde, obwohl andere Elemente *vertex_types*$[v]$ mit der gleichen Kopplungskonstante noch gar nicht betrachtet wurden, wird für diese $h[v] = 0$ gesetzt.

3.2.3 Routine *place_vertex_types*

Diese Routine stellt fest, wieviele Vertizes der Feynman–Graph umfassen soll, und trifft für jeden Vertex die Entscheidungen, wieviele Linien sich an ihm treffen sollen.

Die Version von h, die an *place_vertex_types* übergeben wird, legt zusammen mit *vertex_types* die Zahl m der im Diagramm vorkommenden konkreten Vertizes fest. Es werden alle möglichen Permutationen erstellt, mit der die

topologischen Modell–Vertizes v in der durch $h[v]$ bezeichneten Häufigkeit auf die konkreten Vertex–Positionen 1 bis m verteilt werden können.

- Schritt 6:

  Der *C++*–Vektor *vertex_gets_type*, der die Daten der konkreten Vertizes enthält, wird initialisiert, bleibt aber an dieser Stelle noch leer.

  Es wird festgelegt, dass zuerst der konkrete Vertex mit der Nummer $p = 1$ betrachtet werden soll. Bei der Auswahl, welches Element *vertex_types*$[q]$ für den konkreten Vertex p ausgewählt werden soll, (womit unter anderem festgelegt wird, wieviele *legs* und welche Kopplungskonstante der Vertex p erhalten soll), wird mit dem Wert $q = 1$ begonnen.

- Schritt 7:

  Es wird geprüft, ob alle topologischen Modell–Vertizes so oft plaziert wurden, wie in h gefordert; wenn ja, wird mit der erstellten Version von *vertex_gets_type* die nächste Routine *connect_nodes* begonnen, d.h. Schritt 10 ausgeführt.

- Schritt 8:

  Es wird geprüft, ob *vertex_types*$[q]$ für alle Positionen von 1 bis m im Diagramm in Erwägung gezogen wurde, d.h. ob $p > m$ ist; wenn ja, wird p auf den Wert 1 zurückgesetzt und das nächste Element von *vertex_types* ausgewählt, d.h. q um eins erhöht.

- Schritt 9:

  Es wird geprüft, ob der Vertex p in dieser Version von *vertex_gets_type* noch als frei gekennzeichnet ist.

  Wenn nicht, wird p um 1 erhöht und Schritt 7 ausgeführt.

  Wenn ja, verzweigt das Programm:

  vertex_types$[q]$ wird an der Stelle p sowohl plaziert als auch nicht plaziert, d.h. es wird jeweils als nächster Schritt 7 ausgeführt, aber mit unterschiedlichen Versionen von *vertex_gets_type*, wo nur in einer die Daten des topologischen Modell–Vertizes an der freien Stelle p hinzugefügt wurden.

  In beiden Fällen wird p um eins erhöht.

  Wenn *vertex_types*$[q]$ damit hinreichend oft plaziert wurde, wird q erhöht und p auf 1 gesetzt, d.h. der nächste topologische Modell–Vertex wird für alle konkreten Positionen, die noch frei bleiben, in Erwägung gezogen.

3.2.4 Routine *connect_nodes*

Diese Routine vollendet die Konstruktion der topologischen Gestalt des Graphen, in der für die Linien noch keine Informationen über die Elementarteilchen vorhanden sind. Es wird festgelegt, welcher Knoten mit welchen anderen Knoten verbunden werden soll.

Die in *place_vertex_types* benannten konkreten Vertizes bilden zusammen mit den im Unendlichen liegenden Ursprüngen der externen Partikel–Linien die Menge der *nodes*, die neu durchnummeriert wird, so dass die externen *nodes* die niedrigsten Nummern, beginnend mit Null, haben. Diese Bezeichnungsweise beruht auf der von T. Kaneko verwendeten [1]; bei den meisten anderen nummerierten Mengen oder Vektoren ist die niedrigste Nummer die Eins.

Es müssen alle möglichen Versionen erstellt werden, wie die *nodes* verbunden werden können. Das bedeutet, dass jeder *node* alternativ mit allen anderen verbunden wird, die noch ein freies *leg* haben, d.h. noch nicht die beabsichtigte Anzahl von Linien berühren, die zuvor in *vertex_gets_type* für diesen *node* festgelegt wurde.

Die Daten über die topologische Gestalt des Graphen werden in zwei *C++*–Vektoren erstellt:

node_touches[i] benennt die anderen *nodes*, die der *node* mit der Nummer i berührt. Würde *node_touches*[i] zum Beispiel die Zahlen a,b und c enthalten, bedeutet das, dass von *node* i Linien beginnen, die an den *nodes* mit den Nummern a,b und c enden.

Während des Erstellungsprozesses muss unterschieden werden, von welchem *node* als Ausgangspunkt eine Verbindung hergestellt wurde. In *node_touches* stehen vorläufig nur die „aktiven“ Verbindungen, die von diesem *node* aus angelegt wurden. Für die „passiven“ Verbindungen wird ein zweiter *C++*–Vektor *node_is_touched* angelegt, in dem die *nodes* genannt werden, die den *node* i mit einer von dort begonnen Linie berühren.

Diese Unterscheidung, von welchem *node* aus eine Linie „begonnen“ wurde, ist später nicht mehr erforderlich: Nach topologischer Fertigstellung eines Graphen werden die Daten aus *node_is_touched* zu *node_touches* hinzugefügt. *node_touches* umfasst dann die kompletten Daten der Topologie.

Graphen, die bislang nur eine topologische Gestalt ohne Teilchen haben, sind isomorph, wenn sie sich nur durch eine Permutation ihrer Nummerierung unterscheiden. Eine Permutation bedeutet, dass durch eine Funktion P jedem *node* i eine neue Nummer $P(i)$ zugeordnet wird, so dass anstelle jedes *nodes* k, mit dem i vorher verbunden war, er jetzt mit $P(k)$ verbunden ist. Solche isomorphe Graphen werden nach der von T. Kaneko beschriebenen Weise [1]

bis auf einen einzigen Repräsentanten aussortiert.

Zu Anfang wird einer der externen *nodes* als „*root-node* “ ausgewählt, von dem die Konstruktion der Topologie, also die Verbindung der *nodes* durch Linien, ihren Ausgang nehmen soll. (In der Implementierung wird der externe *node* mit der höchsten Nummer als *root-node* ausgewählt.)

Jedem *node* i wird, sobald er erstmals mit einem anderen verbunden wird, ein *level* zugeordnet: Dieser Zahlenwert misst die minimale Anzahl von Linien, die zwischen diesem *node* i und dem *root-node* liegen.

Von allen *nodes*, die bereits mit dem *root–node* verbunden sind, aber noch freie *legs* haben, werden nur diejenigen auf dem niedrigeren *level* (das als das „aktuelle“ *level* bezeichnet wird) zum weiteren Verbinden ausgewählt (und zwar hierunter immer der *node* mit der niedrigsten Nummer i, bis alle seine *legs* Anschluß gefunden haben). Dadurch ist es nicht möglich, dass ein *node* eine um mehr als Eins höhere *level*–Nummer als die des aktuellen *level*s erhält. (*Nodes*, von denen noch kein einziges *leg* mit einem anderen *node* verbunden ist, haben noch keine *level*–Nummer erhalten.)

Sobald alle *nodes* eines *levels* vollständig sind, also keine freien *legs* mehr aufweisen, wird mit dem in [1] beschriebenen *refinement*–Verfahren eine Voruntersuchung vorgenommen, mit der überflüssige isomorphe Graphen bereits frühzeitig aussortiert werden können.

Hierzu werden alle *nodes* des Diagramms in nummerierte Klassen eingeteilt. Es wird überprüft, ob jeder *node* eine höhere *node*–Nummer als alle *nodes* in niedrigeren Klassen hat. Wenn nicht, wird dieses Diagramm bereits im unfertigen Stadium verworfen; eine Permutation der *node*–Nummerierung für die gleiche topologische Gestalt des Diagramms wird stattdessen in einer parallelen Verzweigung diese Bedingung erfüllen.

- Schritt 10:

  Ohne Beschränkung der Allgemeinheit wird als erste Linie der externe *node* mit der höchsten Nummer (der *root–node*) mit dem internen *node* mit der höchsten Nummer verbunden. (Alle internen *nodes* bzw. Vertizes haben höhere Nummern als die externen *nodes* .)

  Sie erhalten die *level*–Nummern 0 und 1.

- Schritt 11:

  Auf dem aktuellen und dem nächsthöheren *level* werden die nächsten unfertigen *nodes* ermittelt.

  Wenn auf dem aktuellen *level* keiner gefunden wird, ist das *level* abgeschlossen, das nächsthöhere wird zum aktuellen *level*, und das *refinement*–Verfahren wird aufgerufen (Schritt 15).

- Schritt 12:

  Es wird geprüft, ob die topologische Gestalt des Graphen bereits vollständig abgeschlossen ist, d.h. ob überhaupt kein *node* mehr freie *legs* enthält.

  Wenn das der Fall ist, wird der Graph mit isomorphen Graphen verglichen, die sich durch eine Permutation der Nummerierung der *nodes* ergeben. Dabei ist es lediglich erforderlich, Vertizes, die durch das *Refinement*–Verfahren in die gleiche Klasse eingeteilt wurden, zu permutieren.

  Jeder Permutation wird ein Zahlenwert zugeordnet. Dies geschieht anhand der *Adjacency–Matrix*. Die *Adjacency–Matrix* beschreibt die Topologie, indem sie in Spalte x, Zeile y den Wert 1 oder 0 enthält, je nachdem, ob die *nodes* x und y direkt verbunden sind oder nicht.

  Die *Adjacency–Matrix* kann direkt als binärer Zahlenwert aufgefasst werden.

  Nur, wenn der aktuelle Graphentwurf einen niedrigeren solchen *Adjacency*–Wert aufweist als alle seine Permutationen, wird er nicht verworfen, sondern mit seinen topologischen Daten in dem *C++*–Vektor *stored_topologies* als ein vorläufiges Endergebnis gespeichert.

- Schritt 13:

  Für den nächsten unfertigen *node* i auf dem aktuellen *level* wird die Menge aller *nodes* erstellt, die noch für eine Verbindung mit ihm in Frage kommen.

  Dazu wird der *C++*–Vektor $node\_touches[i]$ betrachtet, der nur berücksichtigt, welche Verbindungslinien bisher aktiv von diesem *node* i aus hergestellt wurden.

  Wenn *node* i bereits von Linien berührt wird, die von anderen *nodes* aus „begonnen“ wurden, werden die Nummern dieser *nodes* nicht in $node\_touches[i]$ gespeichert, sondern in dem ergänzenden *C++*–Vektor $node\_is\_touched[i]$.

  Dadurch steht fest, wenn man nun $node\_touches[i]$ betrachtet und darin den „aktiv berührten“ *node* mit der niedrigsten Nummer k auswählt, dass alle *nodes* mit höheren Nummern als k bereits in früheren Rekursionen dieser Routine *connect_nodes* für eine Verbindung mit *node* i in Erwägung gezogen wurden und jetzt nicht mehr betrachtet werden müssen.

Neue Verbindungen dürfen nur noch zu *nodes* mit Nummern kleiner oder gleich k hergestellt werden.

Anmerkung: Bei der Prüfung, ob ein *node* noch unfertig ist, also noch nicht die geforderte Anzahl von Linien berührt, die in *vertex_gets_type* angegeben ist, werden selbstredend sowohl die „aktiv“ als auch die „passiv“ mit anderen *nodes* verbundenen *legs* mitgezählt.

Wenn alle *legs* eines *nodes* „eine Verbindung eingegangen“ sind, werden auch die in *node_is_touched*[i] gespeicherten Verbindungen zu *node_touches*[i] hinzugefügt; im folgenden enthält *node_touches*[i] dann die kompletten Daten über die Linien, die diesen *node* treffen.

- Schritt 14:

  Es findet eine Verzweigung statt: Alternativ werden die Nummern aller noch in Frage kommenden *nodes* zu *node_touches*[i] hinzugefügt. Außerdem wird jeweils i in dem *C++*–Vektor *node_is_touched*, der zu dem *node* am anderen Ende der neuen Linie gehört, gespeichert.

  Mit den jeweils ergänzten Daten über die Gestalt des Diagramms wird wieder Schritt 11 ausgeführt.

3.2.5 *Refinement*

- Schritt 15:

  Die *nodes* werden in nummerierte Klassen eingeteilt und die Einteilung immer weiter verfeinert. Man spricht hierbei von *refinement* [1].

  Als erste Klassifizierung werden die externen *nodes* je in eine eigene Klasse gelegt. Die Vertizes werden hingegen in gleichen Klassen zusammengefasst, wenn sie die gleiche Anzahl *legs* aufweisen.

  Für die darauf aufbauenden verfeinerten Klassifizierungen wird für jeden Vertex ein „Klassifizierungs–Vektor“ angelegt:

  Dessen Element Nr. 0 enthält die zurzeitige Klassennummer des Vertex', das Element Nr. i enthält die Anzahl der Verbindungen des Vertizes mit *nodes* der Klasse i.

  Aus diesen Vektoren lässt sich eine verfeinerte Klassifizierung gewinnen:
  Vertizes, deren Vektoren identisch sind, kommen in die selbe Klasse. Auf diese Weise werden die bisherigen Klassen nur weiter zerlegt, d.h, die Elemente einer alten Klasse können auf mehrere neue Klassen verteilt werden, aber in einer neuen Klasse können nie Vertizes zusammengefasst werden, die vorher in verschiedenen waren.

Die neuen Klassen werden per *lexicographical ordering* der zugehörigen Vektoren nummeriert.

Das *refinement* wird so lange fortgesetzt, bis die Klassen nicht mehr weiter zerlegbar sind.

Sodann wird eine Bedingung an die Nummerierung der *nodes* in diesem topologischen Graphentwurf überprüft:

Wenn die Nummer von *node* i kleiner ist als die von *node* j, muss die Nummer der Klasse, zu der i gehört, kleiner oder gleich der Nummer der Klasse von j sein. Erfüllt das aktuelle Diagramm diese Bedingung nicht, wird die Arbeit daran abgebrochen.

3.3 Belegung der Topologien mit Teilchen

3.3.1 Routine *decorate_graph*

In dieser Routine wird die „Dekorierung“ der Linien des Graphen mit Elementarteilchen begonnen. Dazu wird ein unfertiger Vertex ausgewählt, für den festgestellt wird, zu welchen Teilchenkombinationen aus den Feynman–Regeln er vollendet werden könnte.

In den bisherigen Routinen wurden alle Topologien für den Prozess in der Menge *stored_topologies* gespeichert.

Nacheinander wird nun jede dieser Topologien ausgewählt und weiter bearbeitet.

Eine Topologie wird zu einem fertigen Graphen ergänzt, indem die inneren Linien mit Teilchen „dekoriert“ werden. Da hierfür in der Regel mehrere verschiedene Möglichkeiten zur Auswahl stehen, entstehen zu einer Topologie t oft eine große Zahl unterschiedlicher fertig dekorierter Graphen.

Mit den vorgegebenen Daten über die Feynman-Regeln wird dem Programm auch eine Menge M aller möglichen Modell–Vertizes zur Verfügung gestellt, von denen hier von Bedeutung ist, welche Teilchen sich an dem Vertex treffen. (Die Daten für einen Vertex betrachten alle zugehörigen Teilchen als in den Vertex einlaufend.)

Aus dieser Menge kann eine Auswahl getroffen werden, zu welchem „Endergebnis“ ein Vertex, dessen Linien erst teilweise mit Teilchen „dekoriert“ wurden, durch Hinzufügen von Teilchen ergänzt werden könnte.

- Schritt 16:

  Eine Topologie t wird ausgewählt, um als Grundlage zur Dekorierung mit Teilchen zu dienen.

Die Datenstruktur *vertex_particles* ergänzt diese Topologie um die Informationen, welche Teilchen zu welchen Linien gehören.

Beim ersten Aufruf von *decorate_graph* sind das nur die Daten, welche Vertizes die externen Teilchen berühren.

- Schritt 17:

  Es wird geprüft, welche Vertizes noch unfertig sind, aber mindestens bereits ein Teilchen berühren. Unter diesen wird der Vertex mit der niedrigsten Nummer l ausgewählt.

  Wenn kein unfertiger Vertex gefunden wird, ist der Graph fertig. Er wird auf Korrektheit geprüft und eventuell in dem *C++*–Vektor *stored_graphs* als Endergebnis gespeichert; anschließend wird diese Verzweigung des Programms beendet.

- Schritt 18:

  Aus M wird die Teilmenge M' derjenigen Modell–Vertizes gebildet, in denen das erste Teilchen aus Vertex l anzutreffen ist. (Wenn Vertex l bereits mehr als ein Teilchen enthält, wird dennoch an dieser Stelle nur das erste berücksichtigt. Das erscheint vorteilhafter, weil das Programm bereits im Voraus für jedes Teilchen aus dem physikalischen Modell eine Liste aller Modell–Vertizes, in denen das Teilchen vorkommt, angelegt hat.)

- Schritt 19:

  Aus M' werden alle Modell–Vertizes entfernt, die nach Art der Kopplung und Anzahl der *legs* nicht zur Plazierung am konkreten Vertex l in Frage kommen.

- Schritt 20:

  Nacheinander wird jedes Element von M' ausgewählt; eine Kopie der Daten über seine Teilchenkombination wird unter dem Namen *still_can_find* angelegt.

- Schritt 21:

  Für jedes einzelne Teilchen im unfertigen Vertex l wird die aktuelle Version von *still_can_find* durchgegangen, ob es auch darin enthalten ist; beim jeweils ersten Treffer, und nur beim jeweils ersten Treffer, wird dieses Teilchen in *still_can_find* gelöscht. Auf diese Weise können auch doppelt vorkommende Teilchen korrekt berücksichtigt werden.

- Schritt 22:

  Wenn alle Teilchen aus Vertex l in *still_can_find* gefunden wurden, wird der jeweilige Modell-Vertex zur weiteren Verwendung in der Menge *model_vertices* gespeichert, die an die nächste Routine übergeben wird.

3.3.2 Routine *find_missing_particles*

Diese Routine stellt fest, welche Teilchen als Nächstes zu dem zuvor ausgewählten Vertex hinzugefügt werden könnten, und fügt das ausgewählte Teilchen dann tatsächlich hinzu.

Es wird ermittelt, welche Teilchen an die freie Stelle in Vertex l plaziert werden dürfen. Hierzu wird die Menge *particles_missing* angelegt, die alle Teilchen sammelt, die aus den in *model_vertices* enthaltenen Modell-Vertizes „herausgelöst“ und an den unfertigen Vertex l „angehängt“ werden könnten.

Es wird nicht etwa ein einziger Modell–Vertex ausgewählt, so dass im Folgenden nur noch seine Teilchen zum Vertex l hinzugefügt würden; stattdessen werden alle noch möglichen Teilchen zu *particles_missing* hinzugefügt, ohne zu dokumentieren, aus welchem Modell–Vertex sie stammen.

Sodann wird in parallelen Verzweigungen je eines dieser Teilchen tatsächlich hinzugefügt.

Anschließend wird in der vorigen Routine *decorate_graph* von Neuem nach Modell–Vertizes gesucht, die auch nach den neuen Gegebenheiten noch immer als Endform für Vertex l in Frage kommen.

- Schritt 23:

  Nacheinander wird jeder in *model_vertices* enthaltene Modell–Vertex v ausgewählt.

  Eine Kopie der Daten über die Teilchen, die bereits am bearbeiteten Vertex l plaziert wurden, wird unter dem Namen *placed_already* angelegt.

- Schritt 24:

  Um zu ermitteln, welche Teilchen aus dem Modell–Vertex v in *placed_already* noch fehlen, wird nacheinander jedes Teilchen aus v in der aktuellen Version von *placed_already* gesucht. Wird es gefunden, wird es aus *placed_already* gelöscht (so wird die Gefahr vermieden, dass Teilchen, die in v ein zweites Mal erscheinen, nicht mehr als fehlend erkannt werden, wenn sie bereits einfach in l enthalten sind).

Wird es nicht gefunden, wird das Teilchen der Datenstruktur *particles_missing* hinzugefügt (sofern es noch nicht darin enthalten ist, ein doppeltes Vorkommen muss vermieden werden).

- Schritt 25:

  Es findet eine Verzweigung statt: Alle Teilchen aus *particles_missing* werden alternativ an die erste freie Stelle im Vertex l geschrieben, sowie deren Antiteilchen an der Position in *vertex_particles*, die durch *node_touches* als anderes Ende der Linie bezeichnet wird.

  Mit diesen veränderten Daten wird wieder Schritt 17 aufgerufen.

Kapitel 4

Untersuchung der Graphen und Ermittlung der Faktoren

Nachdem die Topologien und die zugehörigen Graphen fertig erstellt sind, wird die Ermittlung der mathematischen Faktoren gemäß den Feynman–Regeln vorbereitet. Das bedeutet, dass die Graphen auf Schleifen untersucht werden müssen, damit zusätzlich benötigte Impulsvariablen eingeführt werden können, und der Verlauf der zusammenhängenden Fermion–Linien für die Reihenfolge der Faktoren betrachtet werden muss.

Außerdem wird berücksichtigt, dass Graphen sich durch einen Symmetriefaktor von -1 unterscheiden können, je nachdem, wie die äußeren Linien verbunden sind, und ob geschlossene Fermion–Schleifen auftreten.

Eine weitere Untersuchung wird vorgenommen, die an dieser Stelle noch nicht direkt relevant ist, aber später für die Integration von beträchtlichem Nutzen sein kann: Es wird ermittelt, ob verschiedene Graphen zu einem identischen verallgemeinerten Graphen gehören.

4.1 Routine *generalize_graph*

Wenn eine größere Anzahl Feynman–Graphen zu einem physikalischen Prozess erzeugt werden, sind meist mehrere darunter, die sich nur durch konstante Parameter unterscheiden, etwa die Teilchenmassen. Das bedeutet, dass für diese Graphen die aufwändige Integration nur für einen verallgemeinerten Fall durchgeführt zu werden braucht und die individuellen Parameter der einzelnen Graphen erst hinterher eingesetzt werden.

Die Routine *generalize_graph* reduziert die Menge der erstellten Graphen auf eine kleinere Menge verallgemeinerter Graphen. Dazu werden die Namen der Elementarteilchen verallgemeinert.

Bereits in den vom Programm zur Verfügung gestellten physikalischen Daten wird jedem Teilchen ein Buchstabe zugeordnet, der es in eine verallgemeinernde Klasse einordnet:

„f“ für Fermionen, „v“ für Eichbosonen, „s“ für skalare Teilchen und „g“ für Faddeev–Popov–Geister.

Die drei letzteren Klassen werden nur für diese Routine benötigt; die Einteilung in Fermionen und Nicht–Fermionen hat noch an mehreren anderen Stellen Bedeutung, insbesondere bei der Anordnung der Faktoren mit Dirac-Matrizen.

Für jeden Graph zu einer Topologie wird ein verallgemeinerter Graph erstellt, indem jeder Teilchenname durch den Buchstaben für seine verallgemeinerte Teilchenklasse ersetzt wird. Sodann wird geprüft, ob der so entstandene verallgemeinerte Graph bereits von einem anderen Graphen mit der gleichen Topologie erzeugt wurde; wenn nicht, wird er zu der Liste der verallgemeinerten Graphen für diese Topologie hinzugefügt. Nur die verallgemeinerten Graphen brauchen später zum Ausgangspunkt eines Integrationsverfahrens gemacht zu werden.

4.2 Routine *find_loops*

Diese Routine untersucht die Anzahl der Loops in einem Graphen. Genauer gesagt wird ermittelt, wie viele neue Impulsvariablen durch Schleifen im Diagramm zusätzlich zu den äußeren Impulsvariablen auftreten müssen. Außerdem werden die Nummern der Linien festgelegt, zu denen die neuen Impulsvariablen gehören werden, so dass für alle anderen Linien ihr jeweiliger Impuls eindeutig als Summe der äußeren und der Loop–Impulse berechnet werden kann.

Jeder Linie i wird eine Variable $loop\_num[i]$ zugeordnet.

Für externe Linien wird $loop\_num = 0$ gesetzt, alle innere Linien erhalten den Wert $loop\_num = -1$.

Sodann wird für jeden Vertex überprüft, wieviele Linien er berührt, für die $loop\_num = -1$ ist. Die erste solche Linie pro Vertex erhält den neuen Wert $loop\_num = 0$.

Alle weiteren Linien an diesem Vertex mit $loop\_num = -1$ werden dafür markiert, dass sie später eine neue Impulsvariable erhalten sollen:

Dies geschieht, indem an der jeweiligen Linie die Variable $loop\_num$ die Nummer der neuen Impulsvariable erhält, beginnend mit $loop\_num = 1$, dann $loop\_num = 2$ usw.

4.3 Routine *fermion_lines*

Es wird untersucht, welche Fermion–Linien in einem Graphen direkt verbunden sind, d.h. einen „Block“ bilden.

- Schritt 1:

  Jeder Linie l wird eine Variable *been_here*[l] zugeordnet.

  Für alle Linien, die nicht mit einem Fermion dekoriert sind, wird *been_here*[l] =*true* gesetzt, d.h. sie brauchen für die folgenden Betrachtungen nicht mehr berücksichtigt zu werden.

- Schritt 2:

  Es wird die Variable *next_edge* $= -1$ gesetzt, um anzuzeigen, dass noch keine Linie ausgewählt wurde.

- Schritt 3:

  Wenn *next_edge* > 0, d.h. ein aktueller Block verbundener Fermion–Linien noch nicht bis zum Ende verfolgt ist, wird die von *next_edge* identifizierte Linie als nächste betrachtet: *this_edge* = *next_edge*.

  Im anderen Fall wird die Linie mit der niedrigsten Nummer l gesucht, für die *been_here*[l] = *false* ist, und *this_edge* $= l$ gesetzt.

  Wenn für alle l *been_here*[l] = *true*, wird zu Schritt 6 gesprungen.

- Schritt 4:

  Es wird die Linie betrachtet, die durch *this_edge* identifiziert wird.

  Es wird *been_here*[*this_edge*] = *true* gesetzt.

  Die Liniennummer *this_edge* wird in der Datenstruktur *the_fermion_lines* gespeichert.

  Die Menge aller Linien, die *this_edge* berühren, wird erstellt, indem aus den Daten dieser Linie die beiden *nodes* ausgelesen werden, die die Linie verbindet, und aus den Daten über jene *nodes* die Nummern aller dort jeweils anhängenden Linien gewonnen werden.

  Aus allen berührten Linien l wird eine gesucht, für die *been_here*[l] =*false* ist; dann wird *next_edge* $= l$ gesetzt.

  Ist für alle von *this_edge* berührten Linien *been_here* = *true*, ist der aktuelle Block aus Fermionlinien zum Ende gekommen; es wird *this_edge* $= -1$ gesetzt.

- Schritt 5:

  Als Nächstes wird Schritt 3 ausgeführt.

- Schritt 6:

  Die Routine wird beendet.

4.4 Routine *set_fermion_symmetry*

In dem *C++*–Vektor *fermion_order* werden die Nummern aller Linien, die mit Fermionen dekoriert sind, in der Reihenfolge angeordnet, in der später ihre Faktoren multipliziert werden müssen.

Aus dem Vergleich dieser Reihenfolge mit der natürlichen Sortierung nach aufsteigenden Liniennummern ergibt sich ein Symmetriefaktor von *symmetry* $= -1$ oder *symmetry* $= +1$.

Als Anfangswert wird *symmetry* $= 1$ gesetzt.

Für jeden geschlossenen Fermion–Loop wird *symmetry* mit -1 multipliziert.

Dann werden diejenigen Reihen aus zusammenhängenden Fermion–Linien betrachtet, zu denen externe Fermion–Linien gehören.

Je zwei benachbarte Elemente, *fermion_order*$[l]$ und *fermion_order*$[k]$, werden miteinander verglichen. Wenn $l < k$, aber für die zugehörigen Liniennummern *fermion_order*$[l]$ >*fermion_order*$[k]$ gilt, werden die Plätze dieser beiden Elemente von *fermion_order* vertauscht.

Für jede solche Vertauschung wird die Variable *symmetry* mit -1 multipliziert.

Wenn die in *fermion_order* gespeicherten Liniennummern schließlich in der Reihenfolge ihrer Größe angeordnet sind, enthält *symmetry* das Endergebnis für den Symmetriefaktor.

4.5 Routine *generate_factors*

Nach diesen vorbereitenden Untersuchungen kann die Ermittlung der mathematischen Faktoren gemäß den Feynman–Regeln beginnen.

Die Programmbibliothek *graphgenerator* benutzt eine Dokumentation der Feynman–Regeln, die einer Linie gemäß dem zugehörigen Elementarteilchen und einem Vertex gemäß seiner Teilchenkombination einen mathematischen Faktor zuordnet, für den die Kodierung der mathematischen Programmbibliothek GiNaC benutzt wird.

Kapitel 5

Bedienungsanleitung

Eine Anleitung für Download und Installation der Programmbibliothek *graphgenerator* ist auf der Internetseite der Thep-Arbeitsgruppe der Universität Mainz in Vorbereitung.

Um *graphgenerator* zu verwenden, muss einem *C++*–Programm folgender *include*–Befehl vorausgehen:

```
#include "graph_generator.cpp"
```

Zu Beginn muss der Benutzer eine Variable der Klasse *cgenerator* erzeugen:

```
cgenerator *process_name = new cgenerator;
```

Der erste Schritt ist die Erzeugung von Topologien und mit Teilchen dekorierten Graphen. Für einen elektroschwachen Prozess müssen folgende Informationen eingegeben werden:

Die Ordnung der elektroschwachen Kopplungskonstante sowie die Namen der einlaufenden und auslaufenden Teilchen.

Für jedes einlaufende Teilchen muss der Benutzer einen *C++–string* mit dem Teilchennamen der Methode *add_incoming_particle* als Argument übergeben. Auf gleiche Weise werden die Namen der auslaufenden Teilchen nacheinander der Methode *add_outgoing_particle* übergeben.

Die von *graphgenerator* verwendeten Teilchennamen können mit dem Aufruf der Methode *show_particle_list* angezeigt werden.

```
process_name->show_particle_list();
```

Für das Beispiel einer Elektron–Positron–Streuung wäre folgende Eingabe erforderlich:

```
process_name->add_incoming_particle("electron");
process_name->add_incoming_particle("positron");

process_name->add_outgoing_particle("electron");
process_name->add_outgoing_particle("positron");
```

Die Erzeugung aller möglichen Graphen erfolgt durch den Aufruf der Methode *generate_graphs*. Der Übergabeparameter ist die Ordnung o der Kopplungskonstante für den gesamten Graphen, beispielsweise $o = 2$:

```
process_name->generate_graphs( int i = 2 );
```

Es ist auch möglich, mit der Routine *generate_topologies* (gleicher Übergabeparameter) nur Topologien zu erzeugen. Dies sollte aber nur getan werden, wenn für den Prozess keine darauf aufbauenden Betrachtungen benötigt werden. Die im folgenden genannten Routinen können nur korrekt arbeiten, wenn *generate_graphs*, nicht *generate_topologies* aufgerufen wurde!

Die Bildschirmausgabe der erzeugten Daten erfolgt mit den Methoden *show_all_topologies* und *show_all_graphs*:

```
process_name->show_all_topologies();

process_name->show_all_graphs();
```

Die Methode *how_many_graphs* ermittelt die Gesamtzahl der erzeugten Graphen:

```
int i = process_name->how_many_graphs();
```

Für das Folgende ist es notwendig, einen konkreten Graph aus der erzeugten Menge auszuwählen.

Identifiziert wird ein Graph erstens durch die Nummer der ihm zugrunde liegenden Topologie, und zweitens durch seine Nummer in der Menge der zu dieser konkreten Topologie gehörenden Graphen.

Bei der Auswahl dieser beiden Nummern helfen die folgenden Methoden:

Die Funktion *how_many_topologies* gibt die Anzahl der erzeugten Topologien zurück.

```
int k = process_name->how_many_topologies();
```

Es ist wichtig zu wissen, wie viele Graphen zu einer Topologie gehören. Das ermittelt die Methode *graphs_for_topology*:

```
int l = process_name->graphs_for_topology();
```

Mit diesen Informationen ist es möglich, einen Graphen anhand der Nummer der Topologie und seiner Nummer unter den Graphen zu dieser Topologie auszusuchen. Diese beiden Zahlen werden als Übergabeparameter für die Methode *select_graph* benötigt, die den erwünschten Graph auswählt und weiter bearbeitet.

Um beispielsweise die dritte Topologie mit ihrem ersten Graph auszuwählen, wäre folgender Befehl erforderlich:

```
process_name->select_graph( 3,1 );
```

Hinweis: Alle im folgenden genannten Methoden dürfen nur aufgerufen werde, wenn zuvor *select_graph* ausgeführt wurde!

Für den ausgewählten Graph sind durch *select_graph* nun erstmals auch die Linien nummeriert worden. Diese Liniennummerierung ermöglicht zwei neue Betrachtungsweisen des Graphen:

```
process_name->show_touched_edges();
```

show_touched_edges gibt eine Darstellung der Topologie auf dem Bildschirm aus, in der für jeden *node* die Nummern der Linien angegeben sind, die er berührt.

```
process_name->show_edge_list();
```

Durch *show_edge_list* wird der Graph auf dem Bildschirm dargestellt, indem für jede Linie angegeben wird, welche *nodes* sie berührt und welches Teilchen bzw. welches Antiteilchen sich auf dieser Linie befindet.

```
process_name->show_fermion_lines();
```

show_fermion_lines zeigt an, welche Fermion–Linien des Graphen zusammenhängen.

Die Richtung des *fermion_number_flows* kann man der folgenden Methode entnehmen:

```
process_name->show_factor_order();
```

show_factor_order zeigt für alle Linien, die Fermionen tragen, und für alle Vertizes, die Fermionen berühren, die korrekte Reihenfolge an, in der ihre Faktoren nach den Feynman–Regeln multipliziert werden müssen.

Die mathematischen Faktoren selbst werden durch die folgende Methode mit Hilfe der *C++*-Programmbibliothek GiNaC erzeugt:

```
process_name->generate_factors();
```

Das Resultat kann auf zweierlei Weise angezeigt werden.

```
process_name->show_multiplied_factors();
```

show_multiplied_factors gibt den gesamten Beitrag des Graphen zur Wahrscheinlichkeitsamplitude auf dem Bildschirm aus.

```
process_name->show_factors();
```

show_factors hingegen zeigt eine Auflistung, welcher Teilfaktor zu welcher Linie und welcher Teilfaktor zu welchem Vertex gehört.

Kapitel 6

Anwendungsbeispiele

6.1 Die Elektron–Positron–Streuung

Mit folgendem Beispielprogramm erzeugt man die Feynman–Diagramme zu einem Prozess, in dem ein Elektron an einem Positron gestreut wird:

$$e^+e^- \rightarrow e^+e^-$$

Es werden die Graphen zur Ordnung $o = 2$ erzeugt.

Unter den erzeugten Feynman–Diagrammen werden jene beiden genauer betrachtet, bei denen die einlaufenden und auslaufenden Teilchen durch Austausch eines Photons wechselwirken. Es wird insbesondere überprüft, welche Symmetriefaktoren sich bei diesen Graphen ergeben.

```
#include "graph_generator.cpp"

void main()
{
        cgenerator *process1  = new cgenerator;

        process1->add_incoming_particle("positron");
        process1->add_incoming_particle("electron");

        process1->add_outgoing_particle("electron");
        process1->add_outgoing_particle("positron");
```

```
process1->generate_graphs(2);

process1->how_many_topologies();

 // how_many_topologies zeigt die Gesamtzahl der Topologien an.
 // Wenn dies bereits bekannt ist, kann fuer jede dieser
 // Topologien die Anzahl ihrer Graphen abgefragt werden:

process1->graphs_for_topology( 1 );

process1->graphs_for_topology( 2 );

process1->graphs_for_topology( 3 );

process1->graphs_for_topology( 4 );



// Nun wird zur ersten Topologie der erste Graph ausgewaehlt:


process1->select_graph( 1,1 );


// Zur Erinnerung am Bildschirm werden die
// soeben eingegebenen Kennziffern des Graphen angezeigt:

process1->which_graph();


// Erstellen der mathematischen Daten zu diesem Graphen:

process1->generate_factors();


// Ergebnisse anzeigen:

process1->show_line_particles();

process1->show_fermion_lines();
```

```
        process1->show_factor_order();

        process1->show_fermion_symmetry();

        process1->show_multiplied_factors();



        // Nun wird ein anderer Graph ausgewaehlt,
        // naemlich zur dritten Topologie der erste Graph:


        process1->select_graph( 3,1 );

        process1->which_graph();

        process1->generate_factors();

        process1->show_line_particles();

        process1->show_fermion_lines();

        process1->show_factor_order();

        process1->show_fermion_symmetry();

        process1->show_multiplied_factors();
}
```

Dieses Programmbeispiel erzeugt folgende Ausgaben auf dem Bildschirm:

```
There were 4 topologies created.
```

```
Topology #1 has 4 graphs.

Topology #2 has 0 graphs.

Topology #3 has 4 graphs.

Topology #4 has 0 graphs.
```

```
The Graph which gets selected now is:   Graph 1    of topology 1
```
(Ausgabe 1)

```
 node 0 : electron (line 1),
 node 1 : positron (line 2),
 node 2 : electron (line 3),
 node 3 : positron (line 4),
 node 4 : electron (line 2),  positron (line 1),  photon (line 5),
 node 5 : electron (line 4),  positron (line 3),  photon (line 5),
```
(Ausgabe 2)

```
Fermion-Lines:

connected Lines: 1 : 2 :
connected Lines: 3 : 4 :
```
(Ausgabe 3)

```
The Order of Factors:
```

```
 Line: 1    Vertex: 4
 Line: 2    Vertex: 1
 Line: 3    Vertex: 5
 Line: 4    Vertex: 3


Fermion-symmetry-factor:  1
```

(Ausgabe 4)

```
Result:

   1
 *(  v_bar(p1)   )
 *(  -I*e*gamma~mu10  )
 *(  u(p2)   )
 *(  v(p3)   )
 *(  -I*e*gamma~mu11  )
 *(  u_bar(p4)   )
 *(  -I*(p2+p1)^(-2)*eta~mu11~mu10+(p2+p1)~mu11
  *(p2+p1)~mu10*(p2+p1)^(-4)*(I-I*Xsi_A)  )
```

(Ausgabe 5)

```
Topology #3 has 4 graphs.


Selected is:   Graph 1    of topology 3


 node 0 : electron (line 1),
 node 1 : positron (line 2),
 node 2 : electron (line 3),
 node 3 : positron (line 4),
 node 4 : electron (line 2),  positron (line 3),  photon (line 5),
 node 5 : electron (line 4),  positron (line 1),  photon (line 5),
```

```
Fermion-Lines:

connected Lines: 1 : 4 :
connected Lines: 2 : 3 :



The Order of Factors:

 Line: 1 Vertex: 5
 Line: 4 Vertex: 3
 Line: 3 Vertex: 4
 Line: 2 Vertex: 1
```

(Ausgabe 6)

```
Fermion-symmetry-factor:  -1
```

(Ausgabe 7)

```
Result:


   -1
 *(  v_bar(p1)   )
 *(  -I*gamma~mu11*e  )
 *(  u_bar(p4)   )
 *(  v(p3)   )
 *(  -I*gamma~mu10*e  )
 *(  u(p2)   )
 *(  -I*(p3+p2)^(-2)*eta~mu11~mu10+(p3+p2)^(-4)
  *(p3+p2)~mu10*(p3+p2)~mu11*(I-I*Xsi_A)  )
```

(Ausgabe 8)

Ausgabe 1 zeigt an, dass vier Topologien erzeugt wurden, aber nur zwei davon sinnvoll sind, weil zu den anderen beiden keine erlaubten Graphen ermittelt werden konnten. Die vier Topologien ergeben sich dadurch, dass es drei Möglichkeiten gibt, die externen Linien zu zwei Dreier–Vertizes zu verbinden, und eine Möglichkeit, sie mit einem Vierer–Vertex zu verbinden.

Die Art der einlaufenden Teilchen gestattet aber nur bei der ersten und der dritten Topologie, Vertizes in den Feynman–Regeln zu finden, mit denen hierzu Graphen erzeugt werden können.

Zu jeder der beiden Topologien wurden vier Graphen erzeugt:

Die Kopplung kann entweder durch ein Photon, ein Z–Boson oder durch das skalare Higgs oder das unphysikalische skalare χ–Teilchen erfolgen.

Als nächster Schritt wird ein einzelner Graph ausgewählt, um genauer betrachtet zu werden. Wie angezeigt wird, wurde im Quelltext der erste Graph zur ersten Topologie verlangt.

Ausgabe 2 zeigt an, welcher Vertex in diesem Graph mit welcher Linie verbunden ist und welche Teilchen zu den Linien gehören. Mit diesen Informationen kann man die graphische Geatalt des Diagramms rekonstruieren (*Abb.6.1a*).

Ausgabe 3 zeigt an, zu welchen Linien Fermionen gehören, wobei zusammenhängende Fermion–Linien in der selben Zeile ausgedruckt werden.

In Ausgabe 4 werden die zusammenhängenden Fermion–Linien nach dem *fermion–number–flow* sortiert. Bei verbundenen äußeren Linien bedeutet das, dass mit einem einlaufenden Antiteilchen oder einem auslaufenden Teilchen begonnen wird. Dazwischen werden die Vertizes angezeigt, an denen die Linien zusammentreffen, in der korrekten Reihenfolge, in der ihre Faktoren später multipliziert werden.

Ausgabe 5 liefert den kompletten Beitrag des Feynman–Graphen zur Wahrscheinlichkeitsamplitude im Code der Programmbibliothek GiNaC. Die externen Impulse p_i erhalten kein Vorzeichen, unabhängig davon, ob sie zu einlaufenden oder auslaufenden Teilchen gehören. Als Index erhalten sie die

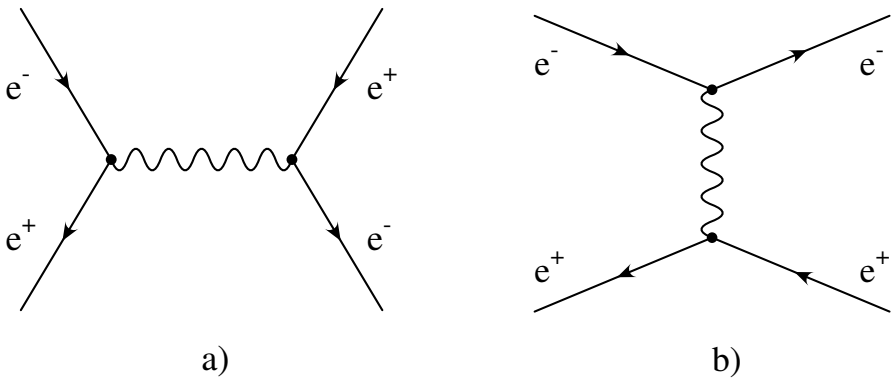

Abbildung 6.1: Elektron–Positron–Streuung

Nummer ihrer Linie. Die verschiedenen Indizes μ werden durch eine Nummerierung unterschieden, abhängig davon, zu welchem Ende welcher Linie sie gehören.

Mit Hilfe von GiNaC könnte das Ergebnis noch weiter vereinfacht werden. Die abgedruckte Bildschirmausgabe lautet übersetzt in die Darstellung von LaTeX wie folgt, wobei $\eta_{\mu_{11}\mu_{10}}$ für den metrischen Tensor, γ für Dirac–Matrizen, p_1 und p_2 für die einlaufenden Impulse, $u(p), v(p), \bar{u}(p), \bar{v}(p)$ für die Spinoren der externen Fermionen, e für die Ladung des Positrons und ξ_A für den Eichparameter steht:

$$-e^2\bar{v}(p_1)\gamma_{\mu_{10}}u(p_2)v(p_3)\gamma_{\mu_{11}}\bar{u}(p_4)$$
$$\left(\left(p_1+p_2\right)_{\mu_{10}}\frac{\left(p_1+p_2\right)_{\mu_{11}}}{\left(p_1+p_2\right)^4}\left(i-i\xi_A\right)-i\frac{\eta_{\mu_{11}\mu_{10}}}{\left(p_1+p_2\right)^2}\right)$$

In Ausgabe 6 werden die analogen Daten für den nächsten Graphen angezeigt, der ausgewählt wird, nämlich den ersten Graphen der dritten Topologie.

Ausgabe 7 zeigt an, dass die Symmetriefaktoren der beiden betrachteten Graphen sich um den Faktor -1 unterscheiden.

Ausgabe 8 liefert den Beitrag zur Amplitude für den neu ausgewählten Graphen, was folgendem Ergebnis entspricht:

$$e^2\bar{v}(p_1)\gamma_{\mu_{11}}\bar{u}(p_4)v(p_3)\gamma_{\mu_{10}}u(p_2)$$
$$\left(\left(p_3+p_2\right)_{\mu_{11}}\frac{\left(p_3+p_2\right)_{\mu_{10}}}{\left(p_3+p_2\right)^4}\left(i-i\xi_A\right)-i\frac{\eta_{\mu_{11}\mu_{10}}}{\left(p_3+p_2\right)^2}\right)$$

6.2 Die Photon–Selbstenergie

Das folgende Beispielprogramm untersucht Korrekturen zum Photonpropagator, d.h. die Photon–Selbstenergie, was durch ein einlaufendes Photon und ein auslaufendes Photon beschrieben wird. Mit der Ordnung der Kopplungskonstante $o = 2$ erhält man Einschleifenkorrekturen zu:

$$\gamma \to \gamma$$

```
#include "graph_generator.cpp"
```

```
void main()
{
        cgenerator *process2  = new cgenerator;

        process2->add_incoming_particle("photon");

        process2->add_outgoing_particle("photon");

        process2->generate_graphs(2);

        process2->how_many_topologies();

        process2->graphs_for_topology( 1 );

        process2->graphs_for_topology( 2 );

        process2->graphs_for_topology( 3 );

        process2->show_all_topologies();

// Wenn die obigen Funktionen in einem frueheren Durchlauf
// des Programms bereits Ergebnisse geliefert haben, kann
// man damit entscheiden, welcher einzelne Graph nun
// ausgewaehlt werden soll:

        process2->select_graph( 1, 9 );

        process2->which_graph();

        process2->generate_factors();

        process2->show_line_particles();

        process2->show_fermion_lines();

        process2->show_factor_order();

        process2->show_fermion_symmetry();

        process2->show_multiplied_factors();
}
```

Dieses Beispielprogramm erzeugt folgende Ausgabe auf dem Bildschirm:

```
 There were 3 topologies created.

Topology #1 has 30 graphs.


Topology #2 has 0 graphs.


Topology #3 has 4 graphs.


****************************** Topology: 1

node: 0  touches:   - 2 -
node: 1  touches:   - 3 -
node: 2  touches:   - 0 -  - 3 -  - 3 -
node: 3  touches:   - 1 -  - 2 -  - 2 -

****************************** Topology: 2

node: 0  touches:   - 3 -
node: 1  touches:   - 3 -
node: 2  touches:   - 2 -  - 2 -  - 3 -
node: 3  touches:   - 0 -  - 1 -  - 2 -

****************************** Topology: 3

node: 0  touches:   - 2 -
node: 1  touches:   - 2 -
node: 2  touches:   - 0 -  - 1 -  - 2 -  - 2 -
```

(Ausgabe 9)

```
The Graph which gets selected now is:   Graph 9    of topology 1


 node 0 : photon (line 1),
 node 1 : photon (line 2),
 node 2 : electron (line 3),  positron (line 4),  photon (line 1),
 node 3 : electron (line 4),  positron (line 3),  photon (line 2),

Fermion-Lines:

Connected Lines: 3 : 4 :



The Order of Factors:

 Line: 3     Vertex: 2
 Line: 4     Vertex: 3


Fermion-symmetry-factor:  1


Result:

   -1
 * dirac_trace(
(-M_electron^2+(l1+p1)^2)^(-1)*(-M_electron^2+l1^2)^(-1)*e^2
*(((l1+p1)\+ONE*M_electron)*gamma~mu5*(l1\+ONE*M_electron)*gamma~mu3)
   )
 *(  1  )
```

(Ausgabe 10)

Ausgabe 9 zeigt alle drei Topologien an, die erzeugt wurden. Dies ist das Resultat der Methode *show_all_topologies*. Für jeden Node wird angegeben, mit welchen anderen *nodes* er verbunden ist. Die analogen graphischen Darstellungen sind in *Abb.6.2* zu sehen.

Man sieht, dass Topologie 1 die äußeren Linien durch eine Schleife aus zwei Linien verbindet. Bei dem im Folgenden betrachteten Graph 9 zu dieser Topologie bedeutet das, dass ein Elektron–Positron–Paar erzeugt und wieder vernichtet wird.

Zu Topologie 2 konnten keine erlaubten Graphen ermittelt werden.

In Topologie 3 sind die äußeren Linien direkt verknüpft, und der entstehende Vertex ist außerdem in einer Schleife aus einer einzigen Linie mit sich selbst verbunden.

Ausgabe 10 wählt zur genaueren Betrachtung den neunten Graph zur ersten Topologie aus. Es wird angezeigt, welche Linien und Teilchen jeder Node berührt. Man kann daraus erkennen, dass ein Elektron–Positron–Paar erzeugt und wieder vernichtet wird (*Abb.6.3*).

Es folgen die Daten über die Anordnung der beiden Fermion–Linien und der dazwischen liegenden beiden Vertizes. Daraus ergibt sich die Reihenfolge der Faktoren, die aneinandermultipliziert als Beitrag des Graphen zur Amplitude ausgegeben werden.

Die Ausgabe *dirac_trace()* weist darauf hin, dass die Spur des geklammerten Terms genommen werden muss. Hierfür stellt GiNaC die Methode *dirac_trace()* zur Verfügung, auf deren Ausführung hier verzichtet wurde. Die Notation $p\backslash$ steht für die Funktion *dirac_slash*(p), d.h. eine Kontraktion von p mit den Dirac–Matrizen ($\not{p} = p_\mu \gamma^\mu$).

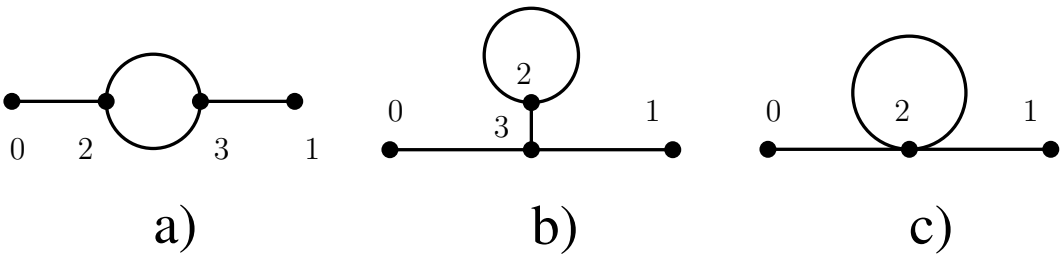

Abbildung 6.2: Die erzeugten Topologien

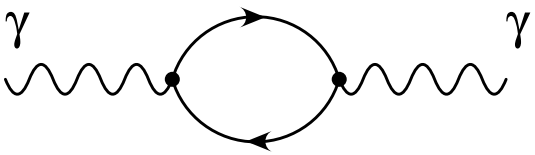

Abbildung 6.3: Photon–Selbstenergie

Die Ausgabe der Routine *show_multiplied_factors* entspricht hier folgendem Term:

$$-\mathrm{Tr}\left(\frac{1}{-m_e^2+(l_1+p_1)^2}\frac{1}{-m_e^2+l_1^2}e^2\left((\not{l}_1+\not{p}_1+m_e)\gamma_{mu_5}(\not{l}_1+m_e)\gamma_{\mu_3}\right)\right)$$

Auf die explizite Angabe von Imaginärteilen $+i\epsilon$ in den Nennern der Propagatoren kann verzichtet werden: Diese können später bei Bedarf ergänzt werden, wenn man vereinbart, dass jede Masse einen negativen Imaginärteil enthalten soll.

6.3 Die Photon–Photon–Z–Kopplung

Das folgende Beispielprogramm beschreibt einen Beitrag zur Berechnung von Einschleifen–Korrekturen, bei dem zwei Photonen in ein Z–Boson übergehen. Diese Vertexkorrektur wird als Baustein für die Berechnung von Strahlungskorrekturen benötigt.

$$\gamma\gamma \to Z$$

Ausgewählt werden die zwei Graphen, bei denen ein Elektron bzw. Positron eine innere Schleife in jeweils gegensätzlichem Umlaufsinn durchläuft (*Abb.6.4*).

```
#include "graph_generator.cpp"


void main()
{

        cgenerator *process3  = new cgenerator;


        process3->add_incoming_particle("photon");

        process3->add_incoming_particle("photon");
```

```
process3->add_outgoing_particle("z");


process3->generate_graphs(3);

process3->how_many_topologies();
process3->graphs_for_topology( 1 );

//Ersten Graph auswaehlen:

process3->select_graph( 1, 17 );

process3->which_graph();
process3->generate_factors();
process3->show_line_particles();
process3->show_fermion_lines();
process3->show_factor_order();
process3->show_fermion_symmetry();
process3->show_multiplied_factors();


//Naechsten Graph auswaehlen:

process3->select_graph( 1, 18 );

process3->which_graph();
```

```
        process3->generate_factors();

        process3->show_line_particles();

        process3->show_fermion_lines();

        process3->show_factor_order();

        process3->show_fermion_symmetry();

        process3->show_multiplied_factors();
}
```

Dieses Beispielprogramm führt zu folgender Ausgabe auf dem Bildschirm:

```
 There were 11 topologies created.


Topology #1 has 38 graphs.


The Graph which gets selected now is:   Graph 17    of topology 1


 node 0 : photon (line 1),
 node 1 : photon (line 2),
 node 2 : z (line 3),
 node 3 : electron (line 4),  positron (line 5),  photon (line 1),
 node 4 : electron (line 6),  positron (line 4),  photon (line 2),
 node 5 : electron (line 5),  positron (line 6),  z (line 3),
```

```
Fermion-Lines:

Connected Lines: 4 : 5 : 6 :
```
(Ausgabe 11)

```
The Order of Factors:

 Line: 6     Vertex: 5
 Line: 5     Vertex: 3
 Line: 4     Vertex: 4
```
(Ausgabe 12)

```
Fermion-symmetry-factor:  -1


Result:

   -1
 * dirac_trace(
-(-M_electron^2+(l1+p1)^2)^(-1)*(-M_electron^2+l1^2)^(-1)
*e^3*(((l1+p2+p1)\+ONE*M_electron)*gamma~mu7*(1/2*sin_theta_w
*cos_theta_w^(-1)*(gamma5+ONE)+1/2*sin_theta_w^(-1)
*(-1/2+sin_theta_w^2)*cos_theta_w^(-1)*(-gamma5+ONE))
*(l1\+ONE*M_electron)*gamma~mu3*((l1+p1)\+ONE*M_electron)
*gamma~mu5)*(-M_electron^2+(l1+p2+p1)^2)^(-1)
  )
 *(  1  )


The Graph which gets selected now is:   Graph 18    of topology 1
```

```
 node 0 : photon (line 1),
 node 1 : photon (line 2),
 node 2 : z (line 3),
 node 3 : electron (line 5),  positron (line 4),  photon (line 1),
 node 4 : electron (line 4),  positron (line 6),  photon (line 2),
 node 5 : electron (line 6),  positron (line 5),  z (line 3),


Fermion-Lines:

Connected Lines: 4 : 5 : 6 :
```

(Ausgabe 13)

```
The Order of Factors:

 Line: 4     Vertex: 3
 Line: 5     Vertex: 5
 Line: 6     Vertex: 4
```

(Ausgabe 14)

```
Fermion-symmetry-factor:  -1

Result:

   -1
 * dirac_trace(
-((l1+p1)^2-M_electron^2)^(-1)*(-M_electron^2+l1^2)^(-1)
*(((l1+p1)\+ONE*M_electron)*gamma~mu3*(l1\+ONE*M_electron)
*gamma~mu7*(1/2*cos_theta_w^(-1)*sin_theta_w
*(gamma5+ONE)+1/2*cos_theta_w^(-1)*sin_theta_w^(-1)
*(-1/2+sin_theta_w^2)*(-gamma5+ONE))*((p2+l1+p1)\+ONE*M_electron)
*gamma~mu5)*(-M_electron^2+(p2+l1+p1)^2)^(-1)*e^3
  )
 *(  1  )
```

(Ausgabe 15)

Ausgabe 11 zeigt an, dass der Quelltext zu einem Ergebnis von 11 Topologien geführt hat. Die Graphen aus *Abb.6.4* gehören zur ersten Topologie. Die übrigen Topologien enthalten entweder *Tadpole*–Schleifen, zu denen hier keine Graphen gefunden werden können, die zu den externen Teilchen passen, oder sie enthalten einen Vierer–Vertex, was zu zahlreichen physikalisch korrekten Graphen führt.

Es werden nacheinander der 17. und 18. Graph zur ersten Topologie ausgewählt, weil in diesen Graphen Elektronen bzw. Positronen ausgetauscht werden.

In den übrigen Graphen zur ersten Topologie durchlaufen entweder ein Fermion, ein Eichboson, ein skalares Teilchen oder eine Kombination davon ($W^{\pm}\phi^{\pm}$) die innere Schleife.

Ausgabe 12 und Ausgabe 14 beschreiben, dass der unterschiedliche Umlaufsinn des Elektrons in der inneren Schleife in beiden Graphen zu einer anderen Anordnung der Faktoren führt. Der 17. Graph zur ersten Topologie liefert folgenden Beitrag zur Amplitude:

$$
\begin{aligned}
&-\mathrm{Tr}\left(-\frac{1}{(l_1+p_1)^2-m_e^2}\frac{1}{l_1^2-m_e^2}e^3\left((\not{l}_1+\not{p}_2+\not{p}_1+m_e)\gamma_{\mu_7}\right.\right.\\
&\left(\frac{1}{2}\frac{\sin\theta_W}{\cos\theta_W}(\gamma_5+1)+\frac{1}{2}\frac{1}{\sin\theta_W\cos\theta_W}(\sin^2\theta_W-\frac{1}{2})(1-\gamma_5)\right)\\
&\left.\left.(\not{l}_1+m_e)\gamma_{\mu_3}(\not{l}_1+\not{p}_1+m_e)\gamma_{\mu_5}\right)\frac{1}{(l_1+p_2+p_1)^2-m_e^2}\right)
\end{aligned}
$$

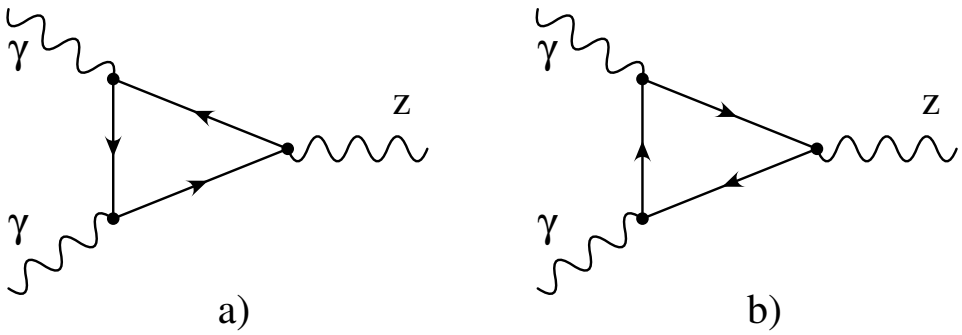

Abbildung 6.4: Photon–Photon–Z–Kopplung

Der 18. Graph zur ersten Topologie entspricht folgendem Term:

$$
\begin{aligned}
&-\mathrm{Tr}\left(-\frac{1}{(l_1+p_1)^2-m_e^2}\frac{1}{l_1^2-m_e^2}\left((\not{l}_1+\not{p}_1+m_e)\gamma_{\mu_3}\right.\right.\\
&(\not{l}_1+m_e)\gamma_{\mu_7}\left(\frac{1}{2}\frac{\sin\theta_W}{\cos\theta_W}(\gamma_5+1)+\frac{1}{2}\frac{1}{\sin\theta_W\cos\theta_W}(\sin^2\theta_W-\frac{1}{2})(1-\gamma_5)\right)\\
&\left.\left.(\not{p}_2+\not{l}_1+\not{p}_1+m_e)\gamma_{\mu_5}\right)e^3\frac{1}{(p_2+l_1+p_1)^2-m_e^2}\right)
\end{aligned}
$$

Zur weiteren Verarbeitung dieser Ergebnisse wäre es erforderlich, ein separates Programm zu erstellen, das Routinen der Programmbibliothek GiNaC verwendet.

6.4 Zweischleifen–Selbstenergien

Es folgt eine Statistik der Graphen, die *graphgenerator* bei Ordnung $o = 4$ zu einlaufenden und auslaufenden Eichbosonen erzeugt.

Die Ordnung $o = 4$ führt bei zwei externen Teilchen unabhängig von der Art dieser Teilchen zu 24 erzeugten Topologien. Dies umfasst auch Topologien mit einem oder zwei Vierer–Vertizes.

Ein interessantes Resultat ist, dass die Anzahl der Graphen sich deutlich reduziert, wenn man die Graphen verallgemeinert und diejenigen zusammenfasst, die zu einem identischen Integrationsprozess führen, bei dem man individuelle Parameter wie die jeweiligen Teilchenmassen erst hinterher einzusetzen braucht.

Es muss allerdings darauf hingewiesen werden, dass *graphgenerator* bei Graphen, in denen Teilchen–Antiteilchen Schleifen auftreten, zwei separate Graphen zählt, wenn die Teilchen– und die Antiteilchen–Linie vertauscht werden. Das bedeutet, dass *graphgenerator* zu viele Graphen erzeugt, von denen einige identisch sind.

Externe Teilchen	*Graphen*	*Verallgemeinerte Graphen*
$Z \to Z$	23296	402
$\gamma \to \gamma$	6698	251
$W^+ \to W^+$	0	0
$W^+ \to W^-$	55019	546
$\gamma \to Z$	7118	254

Anhang A

Die Feynman–Regeln

Es folgt eine Auflistung der in *graphgenerator* verwendeten Feynman–Regeln in der 't Hooft Eichung [5].

Es werden folgende Abkürzungen verwendet:

$c = \cos\theta_W$,
$s = \sin\theta_W$,
wobei θ den schwachen Mischungswinkel bezeichnet.

Für die Teilchenarten werden folgende Symbole verwendet:

V für Eichbosonen ($A,\ Z,\ W^{\pm}$);
G für Faddev–Popov–Geister ($u^A,\ u^Z,\ u^{\pm}$);
S für skalare Felder ($\eta,\ \chi,\ \phi^{\pm}$);
F für Fermionen (f_i),

wobei der Index i für die Generation der Fermionen steht.

Wenn es erforderlich ist, werden die Familien der Leptonen mit l,ν und die der Quarks mit u, d bezeichnet. A bezeichnet das Photon, η das physikalische Higgs mit Masse M_η, χ das unphysikalische Higgs–Teilchen.

Für das Photon wird $M_A = 0$ eingesetzt. Für die Massen der unphysikalischen Teilchen gelten folgende Beziehungen:

$$\begin{aligned} M_{u^A} &= 0, \\ M_{u^Z} &= \sqrt{\xi_Z} M_Z, \\ M_{u^\pm} &= \sqrt{\xi_W} M_W, \\ M_\chi &= \sqrt{\xi_Z} M_Z, \end{aligned}$$

$$M_{\phi^\pm} \quad = \quad \sqrt{\xi_W} M_W .$$

$\eta_{\mu\nu}$ steht für den metrischen Tensor, k_μ und p_i für die Impulse der Linien, Q_f für die Teilchenladung.

Jeder Vertex enthält eine δ–Funktion zur Erhaltung der 4–Impulse. Dies wird nicht explizit aufgeführt, sondern von *graphgenerator* während der Konstruktion der Feynman–Graphen bei der Zuordnung der Impulse direkt berücksichtigt.

A.1 Propagatoren

Den Linien der verschiedenen Teilchen werden folgende Propagatoren zugeordnet:

V	$\dfrac{-i\eta_{\mu\nu}}{k^2 - M_V^2} + \dfrac{i(1-\xi_V)k_\mu k_\nu}{(k^2 - M_V^2)(k^2 - \xi_V M_V^2)}$
G	$\dfrac{i}{k^2 - M_G^2}$
S	$\dfrac{i}{k^2 - M_S^2}$
F	$\dfrac{i(\not{p} + m_F)}{p^2 - m_F^2}$

A.2 Vertizes

A.2.1

Ein $VVVV$ –Vertex führt zu folgendem Ausdruck:

$ie^2C\,[2\eta_{\mu\nu}\eta_{\sigma\rho} - \eta_{\nu\rho}\eta_{\mu\sigma} - \eta_{\rho\mu}\eta_{\nu\sigma}]$

Für C werden folgende Ausdrücke eingesetzt:

$V_\mu V_\nu V_\sigma V_\rho$	C
W^+W^-ZZ	$-\frac{c^2}{s^2}$
W^+W^-AZ	$\frac{c}{s}$
W^+W^-AA	-1
$W^+W^-W^+W^-$	$\frac{1}{s^2}$

A.2.2

Ein VVV –Vertex führt zu folgendem Ausdruck:

$$ieC\left[\eta_{\mu\nu}(k_1-k_2)_\rho+\eta_{\nu\rho}(k_2-k_3)_\mu+\eta_{\rho\mu}(k_3-k_1)_\nu\right]$$

Zum Linienindex μ gehört der Impuls k_1, zum Linienindex ν gehört der Impuls k_2, zum Linienindex ρ gehört der Impuls k_3.

Für C werden folgende Ausdrücke eingesetzt:

$V_\mu V_\nu V_\rho$	C
AW^+W^-	1
ZW^+W^-	$-\frac{c}{s}$

A.2.3

Ein SVV –Vertex führt zu folgendem Ausdruck:

$ie\eta_{\mu\nu}C$

Für C werden folgende Ausdrücke eingesetzt:

$SV_\mu V_\nu$	C
ηZZ	$\frac{1}{c^2 s} M_W$
$\eta W^+ W^-$	$\frac{1}{s} M_W$
$\phi^\pm W^\mp A$	$-M_W$
$\phi^\pm W^\mp Z$	$-\frac{s}{c} M_W$

A.2.4

Ein $V\bar{F}F$ –Vertex führt zu folgendem Ausdruck:

$$ie\gamma_\mu \left(C_L \frac{1-\gamma_5}{2} + C_R \frac{1+\gamma_5}{2} \right)$$

Für C_L und C_R werden folgende Ausdrücke eingesetzt:

$V_\mu \bar{F} F$	C_L	C_R
$A\bar{f}_i f_j$	$-Q_f \delta_{ij}$	$-Q_f \delta_{ij}$
$Z\bar{f}_i f_j$	$\frac{I^3_{W,f} - s^2 Q_f}{sc} \delta_{ij}$	$-\frac{s}{c} Q_f \delta_{ij}$
$W^+ \bar{u}_i d_j$	$\frac{1}{\sqrt{2}s} V_{ij}$	0
$W^- \bar{d}_j u_i$	$\frac{1}{\sqrt{2}s} V^\dagger_{ji}$	0
$W^+ \bar{\nu}_i l_j$	$\frac{1}{\sqrt{2}s} \delta_{ij}$	0
$W^- \bar{l}_j \nu_i$	$\frac{1}{\sqrt{2}s} \delta_{ij}$	0

A.2.5

Ein $S\bar{F}F$ –Vertex führt zu folgendem Ausdruck:

$$ie\left(C_L\frac{1-\gamma_5}{2}+C_R\frac{1+\gamma_5}{2}\right)$$

Für C_L und C_R werden folgende Ausdrücke eingesetzt:

$S\bar{F}F$	C_L	C_R
$\eta\bar{f}_i f_j$	$-\frac{m_{f,i}}{2sM_W}\delta_{ij}$	$-\frac{m_{f,i}}{2sM_W}\delta_{ij}$
$\chi\bar{f}_i f_j$	$-2iI^3_{W,f}\frac{m_{f,i}}{2sM_W}\delta_{ij}$	$2iI^3_{W,f}\frac{m_{f,i}}{2sM_W}\delta_{ij}$
$\phi^+\bar{u}_i d_j$	$+\frac{m_{u,i}}{\sqrt{2s}M_W}V_{ij}$	$-\frac{m_{d,j}}{\sqrt{2s}M_W}V_{ij}$
$\phi^-\bar{d}_j u_i$	$-\frac{m_{d,j}}{\sqrt{2s}M_W}V^\dagger_{ji}$	$+\frac{m_{u,i}}{\sqrt{2s}M_W}V^\dagger_{ji}$
$\phi^+\bar{\nu}_i l_j$	0	$-\frac{m_{l,j}}{\sqrt{2s}M_W}\delta_{ij}$
$\phi^-\bar{l}_j \nu_i$	$-\frac{m_{l,j}}{\sqrt{2s}M_W}\delta_{ij}$	0

A.2.6

Ein $V\bar{G}G$ –Vertex führt zu folgendem Ausdruck:

$iek_{1,\mu}C$

Der Impuls k_1 gehört zum Teilchen $\bar{G}$.
Für C werden folgende Ausdrücke eingesetzt:

$V_\mu \bar{G} G$	C
$A\bar{u}^{\pm}u^{\pm}$	± 1
$W^{\pm}\bar{u}^{A}u^{\mp}$	± 1
$W^{\mp}\bar{u}^{\mp}u^{A}$	± 1
$Z\bar{u}^{\pm}u^{\pm}$	$\mp\frac{c}{s}$
$W^{\pm}\bar{u}^{Z}u^{\mp}$	$\mp\frac{c}{s}$
$W^{\mp}\bar{u}^{\mp}u^{Z}$	$\mp\frac{c}{s}$

A.2.7

Ein $S\bar{G}G$ –Vertex führt zu folgendem Ausdruck:

$ieC\xi_1$

Für den Eichparameter ξ_1 wird ξ_W bei $u^{\pm}$–Teilchen und ξ_Z bei u^Z–Teilchen eingesetzt.

Für C werden folgende Ausdrücke eingesetzt:

$S\bar{G}G$	C
$\eta\bar{u}^Z u^Z$	$-\frac{1}{2c^2 s}M_W$
$\eta\bar{u}^{\pm}u^{\pm}$	$-\frac{1}{2s}M_W$
$\chi\bar{u}^{\pm}u^{\pm}$	$\mp\frac{i}{2s}M_W$
$\phi^{\pm}\bar{u}^{\pm}u^A$	M_W
$\phi^{\pm}\bar{u}^{\pm}u^Z$	$-\frac{c^2-s^2}{2cs}M_W$
$\phi^{\pm}\bar{u}^Z u^{\mp}$	$\frac{1}{2cs}M_W$

A.2.8

Ein $SSSS$ –Vertex führt zu folgendem Ausdruck:

ie^2C

Für C werden folgende Ausdrücke eingesetzt:

SSSS	C
$\eta\eta\eta\eta$	$-\frac{3}{4s^2}\frac{M_H^2}{M_W^2}$
$\chi\chi\chi\chi$	$-\frac{3}{4s^2}\frac{M_H^2}{M_W^2}$
$\eta\eta\chi\chi$	$-\frac{1}{4s^2}\frac{M_H^2}{M_W^2}$
$\eta\eta\phi^+\phi^-$	$-\frac{1}{4s^2}\frac{M_H^2}{M_W^2}$
$\chi\chi\phi^+\phi^-$	$-\frac{1}{4s^2}\frac{M_H^2}{M_W^2}$
$\phi^+\phi^-\phi^+\phi^-$	$-\frac{1}{2s^2}\frac{M_H^2}{M_W^2}$

A.2.9

Ein SSS –Vertex führt zu folgendem Ausdruck:

$$ieC$$

Für C werden folgende Ausdrücke eingesetzt:

SSS	C
$\eta\eta\eta$	$-\frac{3}{2s}\frac{M_H^2}{M_W}$
$\eta\chi\chi$	$-\frac{1}{2s}\frac{M_H^2}{M_W}$
$\eta\phi^+\phi^-$	$-\frac{1}{2s}\frac{M_H^2}{M_W}$

A.2.10

Ein VSS –Vertex führt zu folgendem Ausdruck:

$ieC(k_1 - k_2)_\mu$

Der Impuls k_1 gehört zum Teilchen S_1, der Impuls k_2 gehört zum Teilchen S_2.

Für C werden folgende Ausdrücke eingesetzt:

$V_\mu S_1 S_2$	C
$Z\chi\eta$	$-\frac{i}{2cs}$
$A\phi^+\phi^-$	-1
$Z\phi^+\phi^-$	$\frac{c^2-s^2}{2cs}$
$W^\pm\phi^\mp\eta$	$\mp\frac{1}{2s}$
$W^\pm\phi^\mp\chi$	$-\frac{i}{2s}$

A.2.11

Ein $VVSS$ –Vertex führt zu folgendem Ausdruck:

$ie^2\eta_{\mu\nu}C$

Für C werden folgende Ausdrücke eingesetzt:

$V_\mu V_\nu SS$	C
$ZZ\eta\eta$	$\frac{1}{2c^2s^2}$
$ZZ\chi\chi$	$\frac{1}{2c^2s^2}$
$W^+W^-\eta\eta$	$\frac{1}{2s^2}$
$W^+W^-\chi\chi$	$\frac{1}{2s^2}$
$W^+W^-\phi^+\phi^-$	$\frac{1}{2s^2}$
$AA\phi^+\phi^-$	2
$ZA\phi^+\phi^-$	$-\frac{c^2-s^2}{cs}$

$V_\mu V_\nu SS$	C
$ZZ\phi^+\phi^-$	$-\frac{(c^2-s^2)^2}{2c^2s^2}$
$W^\pm A\phi^\mp\eta$	$-\frac{1}{2s}$
$W^\pm A\phi^\mp\chi$	$\mp\frac{i}{2s}$
$W^\pm Z\phi^\mp\eta$	$-\frac{1}{2c}$
$W^\pm Z\phi^\mp\chi$	$\mp\frac{i}{2c}$

Danksagung

Ich möchte mich bei allen bedanken, die mich bei der Anfertigung dieser Arbeit unterstützt haben, insbesondere bei

- Prof. Dr. K. Schilcher für die Ermöglichung dieser Arbeit und viele wertvolle Anregungen
- Hubert Spiesberger für die geduldige Betreuung, wesentliche Beiträge und freundliche Kritik
- Richard Kreckel und Christian Bauer für schnelle Hilfe und ausführliche Erklärungen bei Problemen mit C++ und Linux
- Jens Vollinga, Do Hoang Son und Markus Knodel für viele anregende Diskussionen und die gute Zusammenarbeit in der xloops–Gruppe
- Mustapha Azzouz für gutgelaunte Stimmung in unserem Zimmer und dafür, dass er immer bereit war, sein Radio leise zu stellen
- und ganz besonders bei meinen Eltern für außerordentliche Unterstützung und Geduld.

Literaturverzeichnis

[1] Toshiaki Kaneko, *A Feynman–graph generator for any order of coupling constants*, Computer Physics Communications **92** (1995) 127.

[2] Christian Bauer, Alexander Frink, Richard Kreckel, *GiNaC 1.0.1*, Mainz, 2001, `http://www.ginac.de`

[3] Otto Nachtmann, *Phänomene und Konzepte der Elementarteilchenphysik*, Friedr. Vieweg und Sohn, Braunschweig, 1986

[4] Bjarne Stroustrup, *Die C++ Programmiersprache*, Addison Wesley, München, 2000

[5] Manfred Böhm, Ansgar Denner, Hans Joos, *Gauge Theories of Strong and Elektroweak Interactions*, Teubner, Stuttgart, 2001

Zusammenfassung

Es wurde ein Computerprogramm in der Sprache $C++$ erstellt, das zu einem physikalischen Prozess, welcher durch die Eingabe der Ordnung der Kopplungskonstante und der Namen der einlaufenden und auslaufenden Teilchen beschrieben wird, alle sich ergebenden Feynman–Graphen erstellt.

Jeder einzelne Feynman–Graph kann auf dem Bildschirm ausgegeben werden, in Gestalt einer Tabelle, die angibt, welche Linien zu welchen Elementarteilchen gehören und welche Linien bzw. Knoten mit welchen anderen Knoten verbunden sind.

Anhand der Feynman–Regeln kann das Programm jeden Graph in seinen mathematischen Beitrag zur Wahrscheinlichkeitsamplitude übersetzen, wobei die mathematische $C++$–Programmbibliothek GiNaC verwendet wird.

Für die elektroschwache Wechselwirkung stellt das Programm alle Feynman–Regeln zur Verfügung; Erweiterungen sind möglich.

Es wurden die wesentlichen Aspekte des Algorithmus erläutert und die Bedienung des Programms sowie mehrere Beispiele beschrieben.