TECHNISCHE
UNIVERSITÄT
DRESDEN

The magnetic dipole moment of the muon in different SUSY models

Bachelor-Arbeit
zur Erlangung des Hochschulgrades
Bachelor of Science
im Bachelor-Studiengang Physik

vorgelegt von

Jobst Ziebell
geboren am 05.11.1992 in Herrljunga

Institut für Kern- und Teilchenphysik
Fachrichtung Physik
Fakultät Mathematik und Naturwissenschaften
Technische Universität Dresden
2015

Eingereicht am 1. Juli 2015

1. Gutachter: Prof. Dr. Dominik Stöckinger
2. Gutachter: Prof. Dr. Kai Zuber

Abstract

The magnetic dipole moment of the muon is one of the most precisely measured quantities in modern physics. Theory however predicts values that disagree with measurement by several standard deviations [1, Abstract].

Because this may hint at physics beyond the standard model, it is a great opportunity to examine the magnetic dipole moment in different supersymmetric models. It is then possible to calculate dependences of the dipole moment on various model parameters as well as to find constraints for their particular values.

The purpose of this paper is to describe how the magnetic dipole moment is obtained in supersymmetric extensions of the standard model and to document the implementation of its calculation in FlexibleSUSY, a spectrum generator for supersymmetric models [2, Abstract].

Contents

1 Introduction

The Standard Model (SM) of particle physics provides a physical model of the fundamental particles and their interactions. As of today, it is the most successful model of our universe on the quantum level. It is not, however, complete. e.g. One of the most prominent features in our perceived world is entirely missing from the model: gravity.
Nonetheless the SM gives very deep and accurate insights into the quantum behaviour of things. Among the quantum phenomena that are described by the SM is the interaction of a lepton with the electromagnetic field. This interaction gives rise to a physical observable: the magnetic dipole moment of the corresponding lepton. Its value for the electron has been measured in extremely precise experiments and agrees with the theoretical predictions.
When turning to the muon, another lepton, the theoretical predictions differ from the measurements by several standard deviations [1, Abstract]. This is of course very exciting, since it may hint at physics that lies beyond the realms of the SM.
Acknowledging that the SM gives a very good description of nature at the quantum level, it sounds like a good idea to further extend the model in order to correct its shortcomings. This is where supersymmetry (SUSY) comes into play.
The idea of SUSY is very simple. Each particle of half-integer spin (fermion) is assigned a superpartner of integer spin (boson) and vice versa. The transformation turning fermions into bosons and the other way around is called a SUSY-transformation.
It is important to note that as of now (June 2015) no SUSY particles have been observed yet. Thus it is postulated that SUSY particles have a very large mass compared to the SM particles, requiring very high energy collider experiments to be detected.
Now the presence and interactions of these new SUSY particles with the SM particles causes the values of certain observables to change with respect to the SM. One of those observables that is rather sensitive to SUSY interactions is the magnetic dipole moment of the muon. The reason it is a lot more sensitive than the corresponding value for the electron is its higher mass, enhancing the quantum contributions of virtual SUSY particles.
A very natural question to ask is why not to look at the tau lepton instead of the muon. Given its even higher mass it should be even more sensitive to SUSY contributions. The reason for not looking at the tau is that it is extremely short lived and hence precision measurements are virtually impossible (as of today).

2 The gyromagnetic ratio

2.1 In classical mechanics

Let $\vec{B}$ denote the magnetic field and $\vec{A}$ the magnetic potential as is common in classical electrodynamics. Furthermore $\vec{j}$ shall refer to the electric current density.
From classical magnetostatics it is well known that a current generates a magnetic potential according to the Biot-Savart equation:

$$\vec{A}(\vec{r}) = \frac{\mu_0}{4\pi} \int_{\mathbb{R}^3} \frac{\vec{j}(\vec{r}\,')}{|\vec{r} - \vec{r}\,'|} d^3 r'$$

For a current I circulating in a planar coil C, this simplifies to

$$\vec{A}(\vec{r}) = \frac{\mu_0 I}{4\pi} \int_C \frac{1}{|\vec{r} - \vec{r}\,'|} d\vec{r}\,'$$

Now, assuming that $\vec{r}$ is far away from the source (the region where $\vec{j}$ is nonzero), one can perform a multipole expansion, giving

$$\vec{A}(\vec{r}) = \frac{\mu_0 I}{4\pi} \int_C d\vec{r}\,' \sum_{l=0}^{\infty} \frac{r'^l}{r^{l+1}} P_l(\cos\phi) = \frac{\mu_0 I}{4\pi} \int_C \frac{d\vec{r}\,'}{r} (1 + \frac{r'}{r} \cos\phi + ...)$$

where the P_l are the Legendre polynomials and ϕ is the angle between $\vec{r}$ and $\vec{r}\,'$. Considering only the first two terms (the first one gives no contribution), one obtains

$$\vec{A}(\vec{r}) = \frac{\mu_0 I}{4\pi r^2} \int_C d\vec{r}\,' \, \vec{r} \cdot \vec{r}\,' = \frac{\mu_0 I}{4\pi r^2} \, \vec{S} \times \vec{r} = \frac{\mu_0}{4\pi} \frac{\vec{m} \times \vec{r}}{r^2}$$

Here $\vec{S}$ is the surface vector of the area enclosed by the coil and $\vec{m} = I\vec{S}$ is identified as the magnetic dipole moment of the current loop.
Restricting the view to circular current loops, it is also possible to express the magnetic moment in terms of the angular momentum $\vec{L}$ of a particle with mass m moving along the current. With $\vec{L} = m\vec{r} \times \vec{v}$ and $v = \frac{2\pi r}{q} I$ it follows that

$$\vec{m} = \frac{q}{2m}\,\vec{L}$$

Now the gyromagnetic ratio can be defined! Postulating that $\vec{L} \parallel \vec{m}$, one defines g to be the factor linking $\vec{L}$ to $\vec{m}$ through the relation

$$\vec{m} = \frac{q}{2m}\,g\,\vec{L}$$

For the previous example it is now obvious that g is equal to 1.

2.2 In quantum mechanics

From the Dirac equation of relativistic quantum mechanics in the covariant notation

$$(i\hbar\gamma^{\mu}\partial_{\mu} - mc)\psi = 0$$

where ψ denotes a Dirac spinor follows that the orbital angular momentum in the sense of the expectation value of the operator $\hat{\vec{L}}$ is not conserved [3, p. 265-266]. To compensate for this rather undesirable consequence a new quantity is introduced. The spin operator $\hat{\vec{S}}$ of a wave function obeying the Dirac equation is defined to exactly compensate this effect, so that the total angular momentum

$$\hat{\vec{J}} = \hat{\vec{L}} + \hat{\vec{S}}$$

is conserved.
Not too surprisingly both $\hat{\vec{L}}$ and $\hat{\vec{S}}$ affect the magnetic moment. Perhaps very surprisingly though, an electron obeying the Dirac equation can be showed to have a spin gyromagnetic ratio of $g_S = 2$ [3, p. 266], while the orbital part is still $g_L = 1$!
While this is astonishing all by itself, with ever increasingly sophisticated experiments, one has measured the electron *g factor* (g_S) to the mind-boggling accuracy of $1 : 10^{12}$ [4, Abstract]. However the measured value is not exactly 2 but rather

$$\frac{g_S}{2} = 1.00115965218085(76)$$

The reason for this deviation from the expected result has been very successfully modelled by quantum field theory.

2.3 In quantum field theory

One defines the *anomalous magnetic dipole moment* $a = \frac{1}{2}(g-2)$, where g is the g factor of the particle in consideration. As shown by experiments the electron has $a \approx 0.00115965218085$ [4, Abstract]. The deviations from $a = 0$ can be most easily understood by looking at Feynman diagrams.
For the purpose of this paper the interesting interaction is that of a fermion with an electromagnetic field i.e. a photon. Hence the Feynman graphs of interest will look as shown in figure 2.1. The Feynman diagrams in this paper are drawn using the convention that curly lines depict vector bosons, dashed lines mean scalar bosons and solid lines stand for fermions.

The zeroth-order graph (c.f. figure 2.2) involving no virtual particles gives the expected value of g equal to 2. The are however higher order processes that also give contributions to the magnetic dipole moment. Two examples of such one-loop diagrams are shown in figure 2.3.

To be consistent with [5, p. 45], these diagram types will be referred to as *type 3* and *type 4* respectively. Throughout this paper only those diagram types will be taken into account. That is done because in the most common SUSY models the major contributions to a_μ that are not present in the SM, come from these diagrams. Thus the values of the vertices shown in figure 2.4 are needed. It is important to note that the vertex interaction with a photon never changes the particle flavour and is symmetric between left and right chirality.

A corresponding one-loop diagram can be fully classified by specifying the particles b and g, which will be referred to as *photon emitter* and *exchange particle* from here on. The photon emitter is the particle having a vertex interaction with a photon and the exchange particle is the particle being exchanged between the two muon interaction vertices.
Almost adopting the notation in [6, p. 9] define the following quantities:

$$A_{b,g} = z_L z_L^* + z_R z_R^*$$
$$B_{b,g} = z_L z_R^* + z_R z_L^*$$

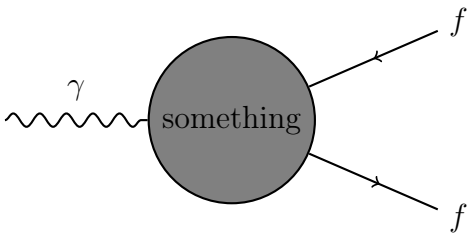

Figure 2.1: Prototypical Feynman diagram depicting the interaction of a fermion with a photon

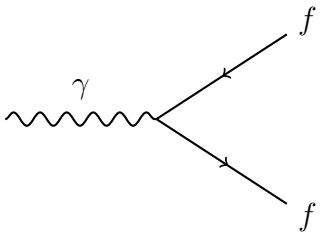

Figure 2.2: Zeroth order Feynman graph showing the interaction of a fermion with a photon

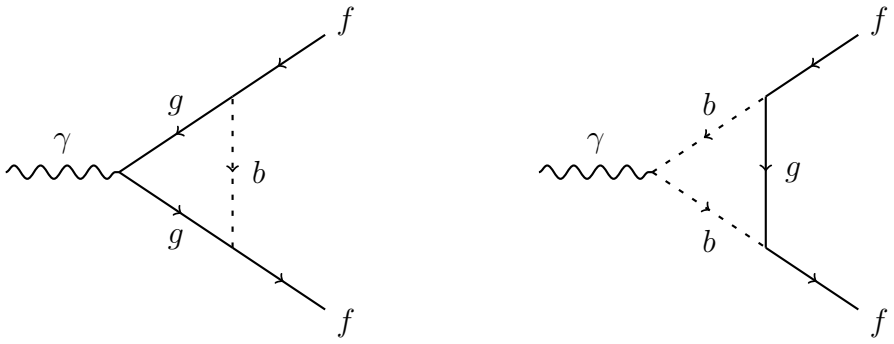

Figure 2.3: Two one loop Feynman graphs showing the interaction of a fermion with a photon. The dotted lines represent scalar bosons, the curly lines vector bosons and the solid lines represent fermions.

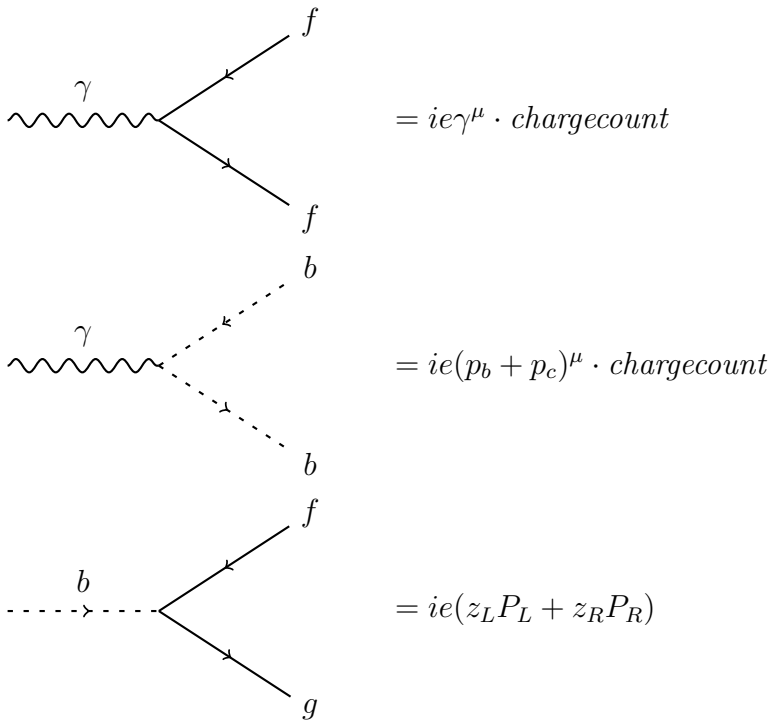

Figure 2.4: The relevant vertices for diagram types 3 and 4

Where z_L and z_R refer to the values in the last vertex of figure 2.4.
Still missing is the charge count of the photon emitter. It is found by calculating the corresponding photon vertex and then normalising the result with respect to the value of a muon-photon-muon vertex. The muon has by definition a charge count of one.
Now all ingredients necessary to calculate the contribution to the anomalous dipole moment of diagrams of type 3 and 4 have been collected. Letting c denote the normalised charge count, generalising the contributions found in [6, p. 10], one obtains:

$$\text{Type 3 diagram:} \quad a_\mu = \frac{c}{16\pi^2}\frac{m_\mu^2}{m_b^2}\left(\frac{1}{12}A_{b,g}F_1^C(x) + \frac{m_g}{3m_\mu}B_{b,g}F_2^C(x)\right) \tag{2.1}$$

$$\text{Type 4 diagram:} \quad a_\mu = \frac{-c}{16\pi^2}\frac{m_\mu^2}{m_b^2}\left(\frac{1}{12}A_{b,g}F_1^N(x) + \frac{m_g}{6m_\mu}B_{b,g}F_2^N(x)\right) \tag{2.2}$$

Here x is a dimensionless mass ratio $x = \frac{m_g^2}{m_b^2}$. The loop functions $F_{1/2}^{N/C}$ are defined in the appendix.
There are some pitfalls in the above formulas though! The first is about the continuity of F_2^C. When x approaches zero (i.e. in a type 3 diagram, when the photon emitter mass goes to zero) F_2^C will diverge. But since in equation 2.1 there is a term $m_g \cdot F_2^C(x)$, a_μ is still continuous.
The second problem lies in the chosen renormalisation scheme. In reality what has to be done, is calculating the *magnetic form factor* $F_M(0)$ [5, p. 42], defined such that

$$a_\mu = -2m_{\mu,physical}F_M(0)$$

where $m_{\mu,physical}$ denotes the *physical* muon mass. Since for the calculations only masses in the $\overline{\text{DR}}$ scheme will be used, one m_μ factor in equations 2.1 and 2.2 has to be replaced with the physical muon mass.
Thus arriving at the final set of equations:

$$\text{Type 3 diagram:} \quad a_\mu = \frac{c}{16\pi^2}\frac{m_{\mu,physical}m_\mu}{m_b^2}\left(\frac{1}{12}A_{b,g}F_1^C(x) + \frac{m_g}{3m_\mu}B_{b,g}F_2^C(x)\right) \tag{2.3}$$

$$\text{Type 4 diagram:} \quad a_\mu = \frac{-c}{16\pi^2}\frac{m_{\mu,physical}m_\mu}{m_b^2}\left(\frac{1}{12}A_{b,g}F_1^N(x) + \frac{m_g}{6m_\mu}B_{b,g}F_2^N(x)\right) \tag{2.4}$$

Now all that has to be done is finding all possible diagrams (of type 3 and 4) and adding up all contributions.

3 The implementation in FlexibleSUSY

FlexibleSUSY is a program that creates spectrum generators for different SUSY models written in Mathematica and C++ [2, Abstract]. It first executes model-dependent meta code, that dynamically generates C++ code, which is then compiled into a binary file that can be executed by the user.
Following this model, the implementation of the magnetic dipole moment of the muon (From here on referred to GMM2 for *g factor of muon minus two*) can be split into three parts.

1. Writing generic C++ code that does not need to access the mathematica part
2. Writing a Mathematica module that generates the model-dependent (C++) code
3. Making FlexibleSUSY run the GMM2 code

The first two parts are not completely disentangled, but enough to view them separately.

3.1 The C++ part

FlexibleSUSY makes use of an intermediate step before it compiles the generic code. Instead of pure C++ files, there are *template* C++ files, containing placeholders @PlaceholderName@. These placeholders are the injection points for the mathematica-generated c++ code.
The first task is to write such generic templated C++ files.
Since not much functionality is added from a user's point of view, one simple header file, declaring only one calculation routine is enough.
That is done in *templates/g_muon_minus_2.hpp.in*. This file defines the function *double calculate_amuon(@ModelName@<Two_scale> &model)* which will be the interface to the calculation routine. The parameter of type *@ModelName@<Two_scale> &* is a reference to the currently run model, which contains e.g. mass definitions, vertex functions etc.
The implementation part resides in *templates/g_muon_minus_2.cpp.in*. It consist of some helper and wrapper classes, the loop functions and the calculation of the a_μ contribution of generic type 3 and 4 diagrams.
The implementation may sound complicated at first, but it offers a very clean interface, is very easily extensible and is mostly DRY (a programming idiom - *Do not Repeat Yourself*). As a

motivation the calculation routine generated by the Mathematica part will look as simple as something like

```
double calculate_amuon(const CMSSM<Two_scale> &model)
{
EvaluationContext context{ model };
double val = 0.0;

val += DiagramEvaluator<OneLoopDiagram<3>, Cha, Sv>::value( context );
val += DiagramEvaluator<OneLoopDiagram<3>, Fe, Ah>::value( context );
val += DiagramEvaluator<OneLoopDiagram<3>, Fe, hh>::value( context );
val += DiagramEvaluator<OneLoopDiagram<4>, Hpm, Fv>::value( context );
val += DiagramEvaluator<OneLoopDiagram<4>, Se, Chi>::value( context );

return val;
}
```

The Diagram classes

These are classes that represent diagram types. For the implementation of GMM2 there is a template *OneLoopDiagram* with explicitly instantiated specialisations *OneLoopDiagram<3>* and *OneLoopDiagram<4>*, since only the diagrams in figure 2.3 are implemented.

The loop functions

The loop functions $F_{1/2}^{C/N}$ referred to in the last chapter are implemented as *OneLoopFunctionFix* where $i \in \{1, 2\}$ and $x \in \{C, N\}$. The implementations are rather straightforward. In general they simply return the mathematical expression of the corresponding function evaluated with *double* precision. Because however parts of the expressions diverge at $x = 0$ and $x = 1$, some special cases have to be taken care of.
All loop functions except F_2^C are continuous at 0. Therefore they first test whether the argument is very small and if this is true the value of the mathematical limit as $x \to 0$ is returned.
If the argument is close to 1, for all loop functions a Taylor expansion is used instead of the direct expression. On an Intel Core i7 processor with 64-bit doubles the absolute error is in all tested cases (which contain at least the taylored range as well as a region around zero) less than 10^{-12}. This can be improved, but it was not deemed a priority.

The particle classes

These are completely generated by the Mathematica part. There is one empty base class *Particle* from which every particle (family) is derived. A typical particle family may be *Fe* for the leptons and a typical particle is *VP* for the photon. Which particles or particle families exist depends on the chosen SUSY model and is ultimately governed by SARAH, a Mathematica package used by FlexibleSUSY. In fact there is a one-to-one mapping between the C++ particles and the SARAH particles.
Two special particle (family) typedefs are always made, so that there will always be the classes *Photon* and *MuonFamily*. These are needed for obvious reasons.
Every particle class contains a static member *numberOfGenerations* specifying the number of SARAH generations that a particle family has. A particle may not be fully specified by its generation index though! Some particles have e.g. a colour index as well.
It will always be assumed that a photon has exactly **one** generation.
There are also definitions for the antiparticles. Their types can be extracted by using the *anti* template. For any particle (or antiparticle) *P*, *typename anti<P>::type* will represent the corresponding antiparticle. For particles that are their own antiparticles these will be the same types.

The evaluation context

This is a simple wrapper around the current model that is run. It contains a *const reference* to the model object, so that e.g. vertex functions can be accessed and it also contains a simple interface for retrieving particle masses. It is used like *mass<Particle>()* or *mass<Particle>(index)* depending on whether the particle class has one or more generations. It also works for antiparticles.
The concrete definitions are supplied by the mathematica part. But the definitions simply forward the calls to the corresponding model class member functions e.g. *get_MFe()*. In all cases $\overline{\text{DR}}$ masses are returned.

The vertex classes

There are several different vertex classes. Some represent specific vertices with fully specified particles, some define more general vertices where *particle indices* (e.g. lepton generation indices) are not yet specified and some are helper classes, that should not be used directly.
Starting with the first ones, the fully specified vertex types, there are two classes *SingleComponentedVertex* and *LeftAndRightComponentedVertex*. These are simple wrappers around one respectively two complex numbers and they can represent (but are not limited to!) the vertices depicted in 2.4. They all have a member function *bool isZero()* that checks whether all absolute

values are below a tiny numerical limit.
The more general vertices are a little more complicated in their definitions. The idea behind their implementation is that the interfaces should be as simple as possible. Therefore the way SARAH handles vertices is mimiced. To this end a template *VertexFunction* is introduced, that is used as follows:

1. The **type** *VertexFunction<Particle1, Particle2, Particle3, ...>* represents a Feynman vertex

2. All particles should of course be derived from the *Particle* class

3. There is no limit on the number of particles through use of variadic templates

4. There is a member typedef *vertex_type* that defines the underlying fully specified vertex type

5. There is a member typedef *indices_type* that is an *std::array* of *unsigned ints* and a specific length

6. There is a member typedef *index_bounds* that contains the upper and lower limits of the indices the particles have

7. There is a static member function *vertex_type vertex(const indices_type &, const EvaluationContext &)* that returns a fully specified vertex type for a given set of indices and an *EvaluationContext*

8. There is a static member function template *std::vector<unsigned int> particleIndices(const indices_type &)* that given a set of indices returns an array containing the indices referring to the particle at the specified template index.
   Furthermore the first index in the returned *std::vector* will always be the generation index if the particle has one.

Given this very general interface it is possible to write **one** general algorithm for every diagram type. Also because there are no restrictions on the represented vertices, this template can be used even outside of the scope of the GMM2 module.
The implementation of the *VertexFunction* template is a little more intricate and can be found in the appendix. However the concrete implementation is not relevant for its use.

The Diagram evaluator template

This is a general template taking any number of template arguments. Its purpose is to calculate the contribution of all diagrams of a specific type with given *Particles*. Its intended use is

DiagramEvaluator<DiagramType, Particles...> For this implementation there exist two partial specializations:

1. *DiagramEvaluator<OneLoopDiagram<3>, PhotonEmitter, ExchangeParticle>*
2. *DiagramEvaluator<OneLoopDiagram<4>, PhotonEmitter, ExchangeParticle>*

Both have a static member function *double value(const EvaluationContext &)* that calculates the contributions for the respective diagrams. It is very important to note that both *PhotonEmitter* and *ExchangeParticle* are derived from *Particle* and are in general **not fully specified**. There may be open indices (generation, colour, ...) and the *value()* routine sums up all contributions for all index combinations.
The implementation of the *value()* routines is very straightforward and can be found in the appendix.

The IndexBounds helper class

This is a simple helper class that represents upper and lower limits for a set of *unsigned ints*. As is common for C++ ranges, the upper limits are non-inclusive. There are also C++ style *begin()* and *end()* functions returning iterators. So the use of this class is very simple.

3.2 The Mathematica part

The Mathematica module *meta/GMuonMinus2.m* contains the code that dynamically generates the model-dependent C++ code. Most of what happens here has already been mentioned in the C++ section. The injection points for the dynamically generated C++ code are as follows:

1. @GMuonMinus2_Particles@, replaced by CreateParticles[]:
   Creates the particle classes
2. @GMuonMinus2_MuonFunctionPrototypes@, replaced by CreateMuonFunctions[]:
   Creates muon helper function prototypes e.g. muonCharge()
3. @GMuonMinus2_Diagrams@, replaced by CreateDiagrams[] :
   Creates the diagram classes
4. @GMuonMinus2_VertexFunctionData@, replaced by CreateVertexFunctionData[]:
   Creates all *VertexFunctionData* specialisations
5. @GMuonMinus2_Calculation@, replaced by CreateCalculation[]:
   Creates the calculation code

6. @GMuonMinus2_ThreadedCalculation@, replaced by CreateThreadedCalculation[]:
   For now, just forwards to CreateCalculation[]

7. @GMuonMinus2_Definitions@, replaced by CreateDefinitions[]:
   Creates all function definitions: mass functions, muon functions and vertex function specialisations

The main job of the Mathematica part is to find out which particles can be inserted into the diagrams and to generate the corresponding vertex functions. The first part is done by iterating over all relevant particles (scalar bosons, fermions and vector bosons) depending on the diagram type and simply checking if the corresponding vertices are non-zero.
The second part is much more intricate. The goal here is to generate small *VertexFunctionData* structures which are explained in the appendix. They contain just enough information so that the C++ template *VertexFunction* can work out everything it needs by itself. This is again in the spirit of DRY and also allows for extremely easy extensibility. Also the *VertexFunction* member function *vertex()* is defined by the mathematica part. It simply forwards the call to the current model object and stores the result in a fully specified vertex object.

3.3 The FlexibleSUSY part

Most of the work here is done in *meta/FlexibleSUSY.m*. The main part is the *WriteGMuonMinus2Class[]* function, that invokes the *GMuonMinus2* module and copies the generated code into the C++ file. Also *MakeFlexibleSUSY[]* is tweaked to store further *nPointFunctions* generated by *GMuonMinus2* in the *NPointFunctions[]* function. These are then used to generate the necessary vertex functions in the model class. In order for this to work a slight change to *SelfEnergies`CreateNPointFunctions[]* was made, so that it ignores *nPointFunctions* with a *Null* head but still generates the correct vertices.

4 Results

Having implemented the calculation of a_μ, FlexibleSUSY has now been used to analyse the theoretical predictions for the muon magnetic moment in the SUSY models CMSSM, E6SSM and MRSSM.
The CMSSM is the *constrained minimal supersymmetric model*. It is a version of the MSSM, the *minimal supersymmetric model*, with additional parameter constraints, such as GUT unification and experimental results [7]. The E6SSM is based on the E_6 group in mathematics and contains additional particles corresponding to a multiplet of the E_6 gauge group [8]. The MRSSM is the *minimal R-symmetric SUSY model*. It contains the additional R-symmetry that might solve different problems in SUSY models [9].
First the E6SSM was examined. The chosen input parameters were:

$$\begin{aligned} m_0 &= 500\,\text{GeV} \\ \tan\beta &= 10 \end{aligned}$$

Here m_0 corresponds to the running masses of the scalar particles in the $\overline{\text{DR}}$ scheme at the unification (GUT) scale and $\tan\beta$ is the ratio of the vacuum expectation values.
Then the running masses of the fermionic particles at the GUT scale ($m_{1/2}$) were varied, to find the point where the contributions to a_μ are maximal. The resulting graph is shown in figure 4.1 and has its peak at $m_{1/2} \approx 850\text{GeV}$. It is very enlightening to look at which particles contribute which amount to a_μ. For the E6SSM that data is compiled in table 4.1.
As predicted by [10, p. 13] (though only for the MSSM) the leading contributions are those of chargino-sneutrino and neutralino-smuon loops, while contributions of loops containing Higgs Bosons are suppressed.

Now the $\tan\beta$ dependence was analysed. For the E6SSM and the CMSSM the input parameters

$$\begin{aligned} m_0 &= 500\,\text{GeV} \\ m_{1/2} &= 850\,\text{GeV} \end{aligned}$$

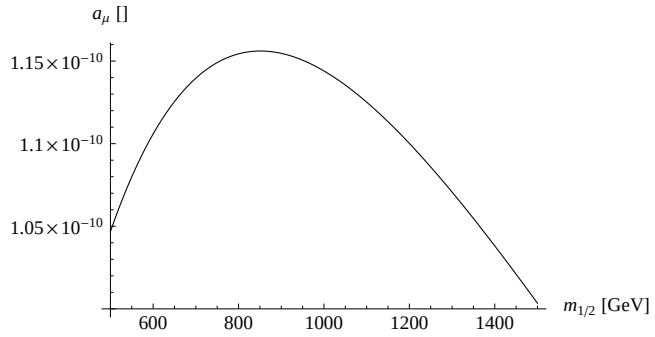

Figure 4.1: a_μ in the E6SSM as a function of $m_{1/2}$

Photon Emitter	Exchange Particle	Contribution $[10^{-10}]$
Chargino(1)	Muon Sneutrino	2.68704
Chargino(2)	Muon Sneutrino	-1.49412
Muon	CP odd Higgs(1)	$-3.58096 \cdot 10^{-4}$
Muon	CP odd Higgs(2)	$-1.68216 \cdot 10^{-10}$
Muon	CP odd Higgs(3)	$-1.36371 \cdot 10^{-4}$
Muon	CP even Higgs(1)	$3.79381 \cdot 10^{-4}$
Muon	CP even Higgs(2)	$1.41508 \cdot 10^{-4}$
Muon	CP even Higgs(3)	$5.79069 \cdot 10^{-10}$
Charged Higgs(1)	Muon Neutrino	$-6.74896 \cdot 10^{-6}$
Charged Higgs(2)	Muon Neutrino	$-1.28891 \cdot 10^{-6}$
Smuon(1)	Neutralino(1)	$-1.19125 \cdot 10^{-1}$
Smuon(1)	Neutralino(2)	$2.34885 \cdot 10^{-2}$
Smuon(1)	Neutralino(3)	$-1.22858 \cdot 10^{-1}$
Smuon(1)	Neutralino(4)	$2.33048 \cdot 10^{-1}$
Smuon(1)	Neutralino(5)	$8.48621 \cdot 10^{-3}$
Smuon(1)	Neutralino(6)	$-7.37169 \cdot 10^{-3}$
Smuon(2)	Neutralino(1)	$1.37325 \cdot 10^{-2}$
Smuon(2)	Neutralino(2)	$-2.42069 \cdot 10^{-1}$
Smuon(2)	Neutralino(3)	$-6.94937 \cdot 10^{-2}$
Smuon(2)	Neutralino(4)	$2.43027 \cdot 10^{-1}$
Smuon(2)	Neutralino(5)	$-9.67685 \cdot 10^{-3}$
Smuon(2)	Neutralino(6)	$1.19754 \cdot 10^{-2}$
Total		1.15610

Table 4.1: Contributions to a_μ in the E6SSM at $m_0 = 500$, $m_{1/2} = 850$ and $\tan\beta = 10$

were used. For the MRSSM the values at the benchmark point with $\tan\beta = 10$ from [9, p. 10-11] were used. Now $\tan\beta$ was varied in the range $[1, 50]$ to find the impact on a_μ. The results are plotted in figure 4.2.
Both the CMSSM and the E6SSM show the expected linear dependence of a_μ on $\tan\beta$ as predicted in [10, p. 12]. The MRSSM does apparently not display the same behaviour. There a_μ seems to be independent of $\tan\beta$. This result is not unexpected. It meets the expectations provided through private communication with Prof. Dr. Dominik Stöckinger, but has previously not been investigated in literature.
Upon magnification though, more structure becomes visible as shown in figure 4.3, but the variations are very small ($\approx 1\%$), so that it is questionable whether they represent physical phenomena or numerical artifacts. To determine this, further analysis that is not within the scope of this paper would be required.

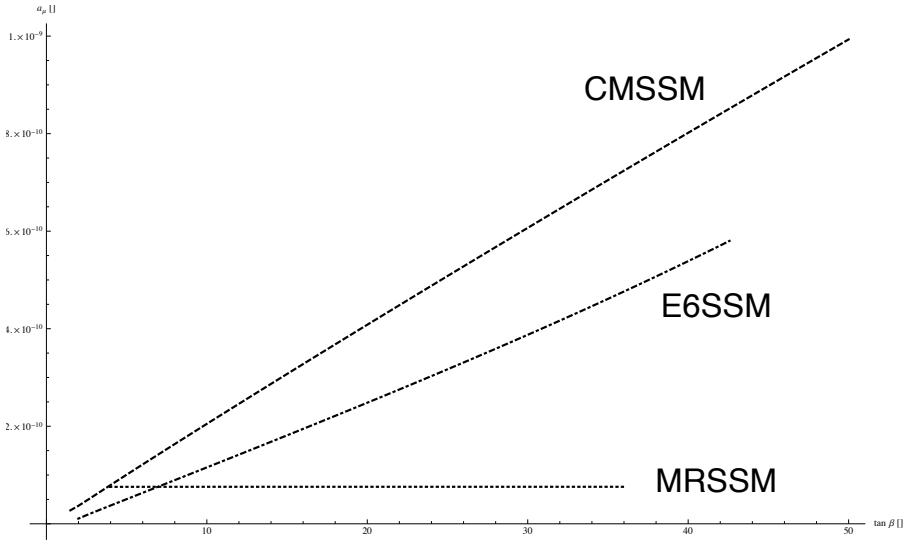

Figure 4.2: The dependence of a_μ on $\tan\beta$ in different SUSY models

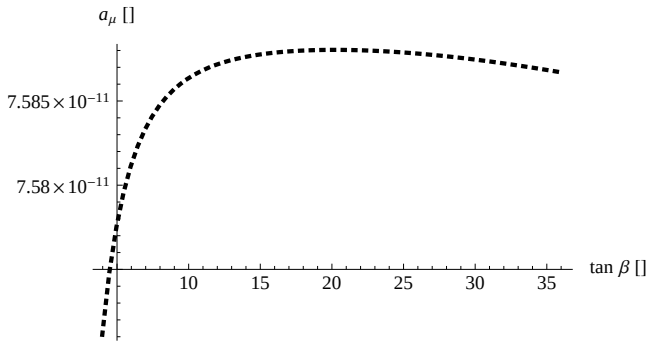

Figure 4.3: The dependence of a_μ on $\tan\beta$ in the MRSSM. It may only be a numerical artifact

Appendices

A The Loop functions

The relevant loop functions are defined in [6, p. 49] as:

$$F_1^C(x) = \frac{2}{(1-x)^4}(2 + 3x - 6x^2 + x^3 + 6x\log x) \tag{A.1}$$

$$F_2^C(x) = \frac{3}{2(1-x)^3}(-3 + 4x - x^2 - 2\log x) \tag{A.2}$$

$$F_1^N(x) = \frac{2}{(1-x)^4}(1 - 6x + 3x^2 + 2x^3 - 6x^2\log x) \tag{A.3}$$

$$F_2^N(x) = \frac{3}{(1-x)^3}(1 - x^2 + 2x\log x) \tag{A.4}$$

B The *VertexFunction* template

The *VertexFunction* template makes use of two helper templates. One is *VertexFunctionHelper* and the other is *VertexFunctionData*. As suggested by the names the *VertexFunctionData* specialisations contain data regarding specific vertices and the *VertexFunctionHelper* template acts as glue between *VertexFunction* and *VertexFunctionData*.
The main complication with the desired interface to *VertexFunction* is the fact that permutations of the particles should not change the overall layout and definition. There are several solutions to this:

1. Using very complex metaprogramming idioms called *generators*, essentially letting C++ fix this by itself

2. Generate complete code for every needed permutation

3. Generate complete code for a canonical order and just the bare minimum for permutations

The main drawback of the first approach is obviously complexity. It is indeed very difficult to get this right and the code is hardly readable at all. Also it tends to really challenge the compiler, resulting in very long compile times. I decided against this. The second option is

arguably the worst one, hence I went for the third solution.
It should be noted that parts of the implementation could be simplified significantly if full C++11 conformance were guaranteed by the compiler. As of now backwards compatibility to g++4.4 has to be maintained and thus complicates things.
The implementation can be described as follows: There are *VertexFunctionData* specialisations for every necessary particle permutation. One of them is chosen as the canonical one and contains a *static const bool is_permutation* flag that is set to false. The non-canonical ones obviously have this flag set to true. A non-canonical *VertexFunctionData* specialisation then only further contains a typedef *orig_type* which is the canonical *VertexFunction* type and a *boost::mpl::vector_c* describing the permutation. A better implementation would use *std::array* and *constexpr* for the permutation, but the latter is missing from g++4.4.
A canonical *VertexFunctionData* specialisation needs to contain more information. It contains the typedefs *index_bounds* and *vertex_type*, with the obvious definitions. Another *boost::mpl::vector_c* is used to specify how many indices the different particles have and stores that in an array of offsets. Moreover a *const index_bounds* is also present for obvious reasons.
Now the *VertexFunctionHelper* template is specialised in two versions, one for canonical vertices and one for permutations. The only new thing they define is the *particleIndices()* template function, one canonical and one permuted variant.
The *VertexFunction* template is derived from the corresponding *VertexFunctionHelper* template. It defines the member template function *vertex()*, that is expected to be specialised for canonical vertices. The specialisation is then provided by the Mathematica part.

C The *DiagramEvaluator<...>::value()* functions

Here the implementation of the *OneLoopDiagram<3>* partial specialisation shall be discussed. One main goal was to only need to write this code once for all type 3 diagrams. Therefore the code has to be very generic and is the reason for the aforementioned abstractions. But because the interfaces were chosen carefully, the code becomes rather straightforward.
The first observation is that only two *VertexFunction* types are needed. One for the vertex involving the muon and one for the vertex with the photon. These are typedef'd as *muonVertexFunction* and *photonVertexFunction*.
Now one has to iterate over all index combinations. Since there is a standard *begin()/end()* interface, this is done in the usual C++ way. The current indices are stored in *photonVertexIndices* and the value of the fully specified vertex is obtained and stored in *photonVertex*. One then checks for numerical significance with *isZero()* and continues with the next index combination if there is none, moving further into the calculation if the vertex is non-zero.
The charge count of the photon emitter is then calculated and all indices concerning the photon

emitter are stored in *photonEmitterIndices*.
Now the same iteration is started for all index combinations in the muon vertex. It is also ensured that only muon contributions are accounted for, by limiting the generation index of the *MuonFamily* to the muon generation index if there is any. There are SARAH models where the muon has its own particle family containing only itself.
Because the second loop is independent of the first, the *photonEmitterIndices* in the outer loop have to match the ones in the inner loops. Only when they are the same, proceeding makes sense.
The rest of the code is very self-explanatory. It is just the expression obtained in the Quantum Field Theory part in the introduction implemented in C++.
One note of caution though! The *DiagramEvaluator* template cannot distinguish between particles and particle families, but it needs to be able to retrieve the masses of particles in both cases. This is done seemingly simple with an *if* switch, specifying a generation index only when necessary. If this was done naively though, it would always cause a compiler error, since one of the function calls is undefined at compile time. This is circumvented by having the *mass* template absorb calls that would be otherwise undefined into infinite recursions causing a runtime error. The *if* switch however ensures that never happens and because the compiler can easily optimise away the *if* statement no performance is lost here.

D How to add more diagram types

This part describes how to improve the value of a_μ by adding more diagram types. Suppose for example support for TwoLoopDiagrams of type 42 should be added (whatever the 42 might stand for). Then the following has to be done:

1. Implement the necessary loop functions in the C++ input file
2. Add *TwoLoopDiagram[42]* to *contributingFeynmanDiagramTypes* in *GMuonMinus2.m*
3. Write new overloads of *CreateDiagramEvaluatorClass[]*, *ContributingDiagramsOfType[]* and *VerticesForDiagram[]* in *GMuonMinus2.m*
4. Implement *DiagramEvaluator<TwoLoopDiagram<42>, ...>::value()* in the C++ input file

Everything else has already been abstracted. Therefore adding more diagrams is a rather simple task, requiring almost only to write the corresponding evaluation routine.

Bibliography

[1] H. N. B. et al, "Precise measurement of the positive muon anomalous magnetic moment," *Phys. Rev. Lett.*, vol. 86, pp. 2227–2231, Mar 2001.

[2] P. Athron, J. hyeon Park, D. Stöckinger, and A. Voigt, "Flexiblesusy — a spectrum generator generator for supersymmetric models," *Computer Physics Communications*, vol. 190, no. 0, pp. 139 – 172, 2015.

[3] Dirac, *Quantum Mechanics*. Oxford University Press, 3 ed., 1947.

[4] D. Hanneke, S. F. Hoogerheide, and G. Gabrielse, "Cavity control of a single-electron quantum cyclotron: Measuring the electron magnetic moment," *Phys. Rev. A*, vol. 83, p. 052122, May 2011.

[5] D. Stöckinger, "Einschleifenbeiträge zu schwachen dipolmomenten und quark-/squarkzerfällen im mssm," diplomarbeit, Universität Karlsruhe, 1998.

[6] H. Fargnoli, C. Gnendiger, S. Paßehr, D. Stöckinger, and H. Stöckinger-Kim, "Two-loop corrections to the muon magnetic moment from fermion/sfermion loops in the mssm: detailed results," *Journal of High Energy Physics*, vol. 2014, no. 2, 2014.

[7] G. L. Kane, C. Kolda, L. Roszkowski, and J. D. Wells, "Study of constrained minimal supersymmetry," *Phys. Rev. D*, vol. 49, pp. 6173–6210, Jun 1994.

[8] P. Athron, D. Stöckinger, and A. Voigt, "Threshold corrections in the exceptional supersymmetric standard model," *Phys. Rev. D*, vol. 86, p. 095012, Nov 2012.

[9] P. Diessner, J. Kalinowski, W. Kotlarski, and D. Stöckinger, "Higgs boson mass and electroweak observables in the mrssm," *Journal of High Energy Physics*, vol. 2014, no. 12, 2014.

[10] D. Stöckinger, "The muon magnetic moment and supersymmetry," *Journal of Physics G: Nuclear and Particle Physics*, vol. 34, no. 2, p. R45, 2007.