

FeynGraph - A Modern Feynman Diagram Toolkit

Young Scientists Meeting 2026 of the CRC TRR 257

Jens Braun | 22 September 2026

Motivation

- Feynman diagrams are the foundation of most scattering amplitude calculations
 - The number of diagrams grows factorially with the number of loops and external legs
 - Most of the diagrams are equivalent, so a robust elimination technique is required
- Producing all diagrams by hand is very error-prone and becomes intractable quickly

Motivation

- Feynman diagrams are the foundation of most scattering amplitude calculations
 - The number of diagrams grows factorially with the number of loops and external legs
 - Most of the diagrams are equivalent, so a robust elimination technique is required
- Producing all diagrams by hand is very error-prone and becomes intractable quickly

Fully automated tools are widely available

QGRAF

[[Nogueira 1993](#)]

High-performance Fortran program with flexible filters and customizable output format

grc

[[Kaneko 1995](#)]

High-performance C program with algorithms similar to QGRAF, now also part of FORM 5 [[Davies et al. 2026](#)]

FeynArts

[[Denner et al. 1992](#); [Hahn 2001](#)]

Mathematica library with custom filters and support for creating analytic expressions from diagrams

Motivation

- Feynman diagrams are the foundation of most scattering amplitude calculations
 - The number of diagrams grows factorially with the number of loops and external legs
 - Most of the diagrams are equivalent, so a robust elimination technique is required
- Producing all diagrams by hand is very error-prone and becomes intractable quickly

Fully automated tools are widely available

QGRAF

[[Nogueira 1993](#)]

High-performance Fortran program with flexible filters and customizable output format

grc

[[Kaneko 1995](#)]

High-performance C program with algorithms similar to QGRAF, now also part of FORM 5 [[Davies et al. 2026](#)]

FeynArts

[[Denner et al. 1992](#); [Hahn 2001](#)]

Mathematica library with custom filters and support for creating analytic expressions from diagrams

- Can be difficult to integrate into modern workflows due to custom model and output formats
- Automatic evaluation of diagram signs with four-fermion operators is not generally supported
- Custom diagram filters not supported by QGRAF and grc

FeynGraph

Feynman diagram generator for arbitrary loop orders and models

- High-performance Rust library ($\sim \mathcal{O}(10^5)$ diagrams/s, QGRAF : $\sim \mathcal{O}(10^6)$ diagrams/s)
- Readily usable from multiple programming languages or the command line
 - Native Rust interface
 - Fully featured Python interface + CLI program
 - Possibly in the future: C to bridge the gap to C, C++, Fortran, Mathematica, ...
- Python library available from [PyPI](#)
- Automatic calculation of diagram signs for multi-fermion operators based on the analytical expressions
- Built-in import of UFO 2.0 [[Darmé et al. 2023](#)] and QGRAF¹ models
- Arbitrary, user-defined diagram filters
- Automatic diagram drawing in SVG, TikZ and typst formats

Source Code

<https://github.com/Jens-Braun/FeynGraph>

User Guide + Python API documentation

<https://jens-braun.github.io/FeynGraph/>

¹Does not supply the necessary information for all features

Generating Topologies [Kaneko 1995]

- Translate the given number of loops and legs into a set of *node partitions* (set of number of occurrences for each node degree)

Generating Topologies [Kaneko 1995]

- Translate the given number of loops and legs into a set of *node partitions* (set of number of occurrences for each node degree)
- For each node partition, start from the set of nodes with no edges
 - Add the next edge: choose starting node according to the current classification and choose the destination class and node according to ordering conditions
 - Keep adding edges until all open half-edges are occupied and check if final graph is to be kept according to the scalar coding
 - Repeat with all allowed start/destination pairs to exhaust all ways to assign edges between the nodes

Generating Topologies [Kaneko 1995]

- Translate the given number of loops and legs into a set of *node partitions* (set of number of occurrences for each node degree)
- For each node partition, start from the set of nodes with no edges
 - Add the next edge: choose starting node according to the current classification and choose the destination class and node according to ordering conditions
 - Keep adding edges until all open half-edges are occupied and check if final graph is to be kept according to the scalar coding
 - Repeat with all allowed start/destination pairs to exhaust all ways to assign edges between the nodes
- Naively adding edges produces many equivalent topologies, which can be mapped onto each other with edge and node permutations

Generating Topologies [Kaneko 1995]

- Translate the given number of loops and legs into a set of *node partitions* (set of number of occurrences for each node degree)
- For each node partition, start from the set of nodes with no edges
 - Add the next edge: choose starting node according to the current classification and choose the destination class and node according to ordering conditions
 - Keep adding edges until all open half-edges are occupied and check if final graph is to be kept according to the scalar coding
 - Repeat with all allowed start/destination pairs to exhaust all ways to assign edges between the nodes
- Naively adding edges produces many equivalent topologies, which can be mapped onto each other with edge and node permutations
- In order to greatly reduce the number of equivalent diagrams, a *node classification* is kept to group equivalent vertices according to the current set of edges and choose only *one* of the equivalent choices for the next edge
 - Start without edges, then all vertices of the same degree are in the same class
 - Refine the classification in the course of the procedure

Generating Topologies [Kaneko 1995]

- Translate the given number of loops and legs into a set of *node partitions* (set of number of occurrences for each node degree)
- For each node partition, start from the set of nodes with no edges
 - Add the next edge: choose starting node according to the current classification and choose the destination class and node according to ordering conditions
 - Keep adding edges until all open half-edges are occupied and check if final graph is to be kept according to the scalar coding
 - Repeat with all allowed start/destination pairs to exhaust all ways to assign edges between the nodes
- Naively adding edges produces many equivalent topologies, which can be mapped onto each other with edge and node permutations
- In order to greatly reduce the number of equivalent diagrams, a *node classification* is kept to group equivalent vertices according to the current set of edges and choose only *one* of the equivalent choices for the next edge
 - Start without edges, then all vertices of the same degree are in the same class
 - Refine the classification in the course of the procedure
- This procedure still leaves some equivalent diagrams, which are eliminated with a final comparison of a scalar coding
 - The coding of a topology is compared to the codings of all topologies produced by permuting vertices within a class
 - Only topologies with a maximal coding are kept

Generating Topologies [Kaneko 1995]

- Translate the given number of loops and legs into a set of *node partitions* (set of number of occurrences for each node degree)
- For each node partition, start from the set of nodes with no edges
 - Add the next edge: choose starting node according to the current classification and choose the destination class and node according to ordering conditions
 - Keep adding edges until all open half-edges are occupied and check if final graph is to be kept according to the scalar coding
 - Repeat with all allowed start/destination pairs to exhaust all ways to assign edges between the nodes
- Naively adding edges produces many equivalent topologies, which can be mapped onto each other with edge and node permutations
- In order to greatly reduce the number of equivalent diagrams, a *node classification* is kept to group equivalent vertices according to the current set of edges and choose only *one* of the equivalent choices for the next edge
 - Start without edges, then all vertices of the same degree are in the same class
 - Refine the classification in the course of the procedure
- This procedure still leaves some equivalent diagrams, which are eliminated with a final comparison of a scalar coding
 - The coding of a topology is compared to the codings of all topologies produced by permuting vertices within a class
 - Only topologies with a maximal coding are kept
- For all final diagrams, calculate topological properties like the set of bridges and the edge momenta

Generating Feynman Diagrams

- For each topology, assign all allowed combinations of propagators and vertices to the edges and nodes

Generating Feynman Diagrams

- For each topology, assign all allowed combinations of propagators and vertices to the edges and nodes
- This procedure again produces equivalent diagrams due to propagator and vertex permutations
 - Reuse the information about equivalent nodes from the topology to only a single propagator assignment out of multiple equivalent ones
 - As before, use a scalar coding to eliminate all remaining equivalent diagrams

Generating Feynman Diagrams

- For each topology, assign all allowed combinations of propagators and vertices to the edges and nodes
- This procedure again produces equivalent diagrams due to propagator and vertex permutations
 - Reuse the information about equivalent nodes from the topology to only a single propagator assignment out of multiple equivalent ones
 - As before, use a scalar coding to eliminate all remaining equivalent diagrams
- For all final diagrams, compute the symmetry factor and the diagram sign

Generating Feynman Diagrams

- For each topology, assign all allowed combinations of propagators and vertices to the edges and nodes
- This procedure again produces equivalent diagrams due to propagator and vertex permutations
 - Reuse the information about equivalent nodes from the topology to only a single propagator assignment out of multiple equivalent ones
 - As before, use a scalar coding to eliminate all remaining equivalent diagrams
- For all final diagrams, compute the symmetry factor and the diagram sign

```
[1]: import feyngraph as fg
      diags = fg.generate_diagrams(["u", "u~"], ["g", "g"], 2)
      len(diags)
```

```
[1]: 119312
```

Validation: Topologies

n	l	$D = \{3, 4\}$	$D = \{3, 4, 5\}$	$D = \{3, 4, 5, 6\}$
0	2	3	3	3
0	3	12	14	15
0	4	73	100	108
2	1	3	3	3
2	2	29	34	35
2	3	351	469	496
2	4	5076	7633	8298
2	5	84 749	143 074	159 510
4	1	99	110	111
4	2	2214	2768	2863
4	3	50 051	70 288	74 559
4	4	1 183 503	1 863 728	2 026 330
4	5	29 712 820	52 371 565	58 318 278
6	0	220	235	236
6	1	11 460	13 550	13 813
6	2	436 780	576 945	600 870
6	3	15 112 015	22 331 900	23 796 266
8	0	34 300	38 745	39 179
8	1	2 876 580	3 601 080	3 700 886
8	2	161 048 720	224 680 890	235 609 983

- Comparison of the total number of generated topologies with n legs and l loops
- Choice of different models containing different sets D of allowed vertex degrees
- Vacuum numbers validated against FORM 5, other numbers against QGRAF

Validation: Diagrams

Process	l	QCD	SM	SMEFT
$gg \rightarrow gg$	0	4	4	11
$gg \rightarrow gg$	1	285	533	2026
$gg \rightarrow gg$	2	15 516	88 366	519 234
$gg \rightarrow gg$	3	785 490	18 442 922	182 305 878
$gg \rightarrow HHH$	1	0	132	15 952
$gg \rightarrow HHH$	2	0	175 692	18 257 938
$gg \rightarrow HHH$	3	0	132 033 872	16 023 358 168
$u\bar{u} \rightarrow u\bar{u}gg$	0	72	168	668
$u\bar{u} \rightarrow u\bar{u}gg$	1	6130	68 046	661 670
$u\bar{u} \rightarrow u\bar{u}gg$	2	393 396	21 583 992	484 349 435
$u\bar{u} \rightarrow u\bar{u}u\bar{u}u\bar{u}$	0	2400	48 552	993 624
$u\bar{u} \rightarrow u\bar{u}u\bar{u}u\bar{u}$	1	286 200	32 085 048	1 597 633 008

- Comparison of the total number of generated l -loop diagrams for the given process
- All diagrams are counted, no selection is performed
- The SMEFT model contains 18 additional bosonic operators,

$$\begin{aligned}
 &O_G, \quad O_{\tilde{G}}, \quad O_W, \quad O_{\tilde{W}}, \quad O_\varphi, \quad O_{\varphi\Box} \\
 &O_{\varphi D}, \quad O_{\varphi G}, \quad O_{\varphi\tilde{G}}, \quad O_{\varphi W}, \quad O_{\varphi\tilde{W}}, \quad O_{\varphi B}, \\
 &O_{\varphi\tilde{B}}, \quad O_{\varphi WB}, \quad O_{\varphi\tilde{W}B}, \quad O_{e\varphi}, \quad O_{u\varphi}, \quad O_{d\varphi},
 \end{aligned}$$

amounting to 257 vertices

- In the SMEFT case, at most one new physics vertex insertion is allowed

Diagram Signs with Multi-Fermion Vertices

Diagram signs are determined with the Flipping Rule Algorithm [[Denner et al. 1992](#)]

- 1 Trace all open fermion lines to find pairs of connected external legs (i_k, j_k) (with $i < j$)
- 2 The diagram sign due to spinor permutations is then the sign of the permutation which transforms $(i_1, j_1, \dots, i_n, j_n)$ to an arbitrary reference ordering (usually $(1, \dots, 2n)$)
- 3 Trace the remaining internal fermion lines to find the number L of closed fermion loops
- 4 The diagram sign due to loops is then $(-1)^L$
- 5 Multiply both signs to get the overall diagram sign

²This information is not available in a QGRAF model, therefore signs of diagrams with multi-fermion vertices might not be consistent when using QGRAF models

Diagram Signs with Multi-Fermion Vertices

Diagram signs are determined with the Flipping Rule Algorithm [[Denner et al. 1992](#)]

- ➊ Trace all open fermion lines to find pairs of connected external legs (i_k, j_k) (with $i < j$)
 - ➋ The diagram sign due to spinor permutations is then the sign of the permutation which transforms $(i_1, j_1, \dots, i_n, j_n)$ to an arbitrary reference ordering (usually $(1, \dots, 2n)$)
 - ➌ Trace the remaining internal fermion lines to find the number L of closed fermion loops
 - ➍ The diagram sign due to loops is then $(-1)^L$
 - ➎ Multiply both signs to get the overall diagram sign
-
- The central component of the algorithm is the correct tracing of the fermion lines
 - For renormalizable vertices, the fermion flow is unambiguous since there are exactly zero or two fermionic legs per vertex
 - For four (or more) fermion operators, the fermion flow is ambiguous without additional information on which legs of a vertex are connected
 - The connection information is encoded in the analytical structure of the vertices, available from the UFO model²

²This information is not available in a QGRAF model, therefore signs of diagrams with multi-fermion vertices might not be consistent when using QGRAF models

Diagram Signs with Multi-Fermion Vertices

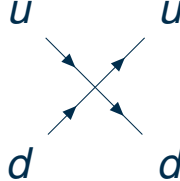
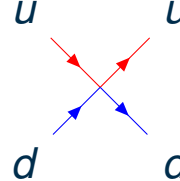
- For vertices with ambiguous fermion flow, the connection map has to be constructed from the analytical expression of the vertex

e.g.

$$\begin{array}{c} u & & u \\ & \searrow \quad \nearrow \\ & \nearrow \quad \searrow \\ d & & d \end{array} \sim (\gamma^\mu)_{s_2 s_1} (\gamma_\mu)_{s_4 s_3} \sim \begin{array}{c} u & & u \\ & \searrow \quad \nearrow \\ & \nearrow \quad \searrow \\ d & & d \end{array}$$

Diagram Signs with Multi-Fermion Vertices

- For vertices with ambiguous fermion flow, the connection map has to be constructed from the analytical expression of the vertex

e.g.  $\sim (\gamma^\mu)_{s_2 s_1} (\gamma_\mu)_{s_4 s_3} \sim$ 

- If there are multiple fermion flows in the analytical structure, the vertex is split to make the flow unambiguous

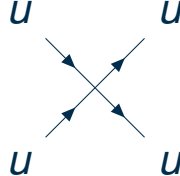
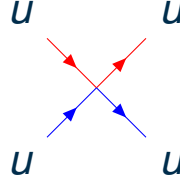
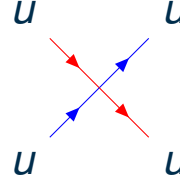
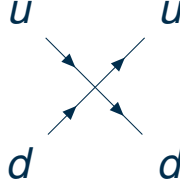
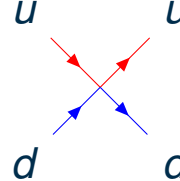
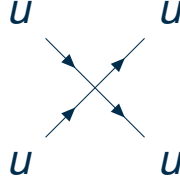
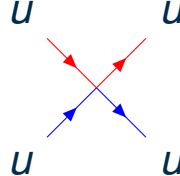
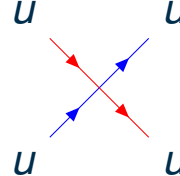
e.g.  $\sim (\gamma^\mu)_{s_2 s_1} (\gamma_\mu)_{s_4 s_3} + (\gamma^\mu)_{s_2 s_3} (\gamma_\mu)_{s_4 s_1} \sim$  $+$ 

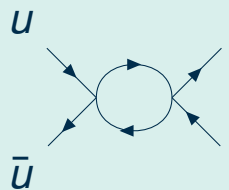
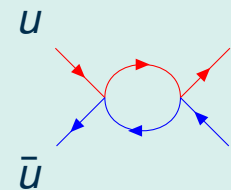
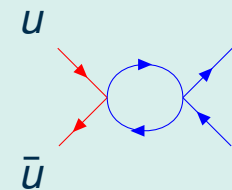
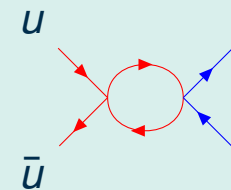
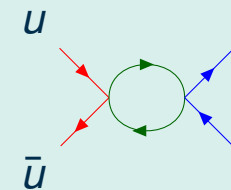
Diagram Signs with Multi-Fermion Vertices

- For vertices with ambiguous fermion flow, the connection map has to be constructed from the analytical expression of the vertex

e.g.  $\sim (\gamma^\mu)_{s_2 s_1} (\gamma_\mu)_{s_4 s_3} \sim$ 

- If there are multiple fermion flows in the analytical structure, the vertex is split to make the flow unambiguous

e.g.  $\sim (\gamma^\mu)_{s_2 s_1} (\gamma_\mu)_{s_4 s_3} + (\gamma^\mu)_{s_2 s_3} (\gamma_\mu)_{s_4 s_1} \sim$  $+$ 

 $=$  $+$  $+$  $+$ 

Model Definition

- Fully customizable model system, based on a Rust `trait` abstraction
 - FeynGraph supports using all Rust object implementing a `Model` trait as the model

Model Definition

- Fully customizable model system, based on a Rust trait abstraction
 - FeynGraph supports using all Rust object implementing a Model trait as the model

```
pub trait ModelBase: Debug + Send + Sync {
    type Particle: ParticleBase;
    type Vertex: VertexBase;
    fn particle(&self, index: usize) -> &Self::Particle;
    fn vertex(&self, index: usize) -> &Self::Vertex;
    fn particles(&self)
        -> impl ExactSizeIterator<Item = &Self::Particle>;
    /* Some more methods similar to the above */
}

pub trait ParticleBase {
    fn name(&self) -> &str;
    fn anti_name(&self) -> &str;
    fn id(&self) -> i64;
    fn is_self_anti(&self) -> bool;
    fn is_fermi(&self) -> bool;
}

pub trait VertexBase {
    fn name(&self) -> &str;
    fn particles(&self) -> &[impl AsRef<str>];
    fn coupling_orders(&self) -> &HashMap<String, usize>;
    fn fermi_map(&self, in_ray: usize) -> i64;
}
```

- Implemented out of the box for QGRAF and UFO models
- Can be implemented from Rust or Python to allow FeynGraph to use any user-defined model format

Model Definition

- Fully customizable model system, based on a Rust trait abstraction
 - FeynGraph supports using all Rust object implementing a Model trait as the model

```
pub trait ModelBase: Debug + Send + Sync {
    type Particle: ParticleBase;
    type Vertex: VertexBase;
    fn particle(&self, index: usize) -> &Self::Particle;
    fn vertex(&self, index: usize) -> &Self::Vertex;
    fn particles(&self)
        -> impl ExactSizeIterator<Item = &Self::Particle>;
    /* Some more methods similar to the above */
}

pub trait ParticleBase {
    fn name(&self) -> &str;
    fn anti_name(&self) -> &str;
    fn id(&self) -> i64;
    fn is_self_anti(&self) -> bool;
    fn is_fermi(&self) -> bool;
}

pub trait VertexBase {
    fn name(&self) -> &str;
    fn particles(&self) -> &[impl AsRef<str>];
    fn coupling_orders(&self) -> &HashMap<String, usize>;
    fn fermi_map(&self, in_ray: usize) -> i64;
}
```

- Implemented out of the box for QGRAF and UFO models
- Can be implemented from Rust or Python to allow FeynGraph to use any user-defined model format

```
[1]: mod1 = fg.Model.new_ufo("UFO_QL_all_operators")
      print(len(mod1.particles()), len(mod1.vertices()))
```

```
[1]: 43 6618
```

```
[2]: mod2 = fg.Model.new_qgraf("sm.qgraf")
      print(len(mod2.particles()), len(mod2.vertices()))
```

```
[1]: 43 145
```

Model Definition

```
[ quark, antiquark, -1 ; TSP='1', COLOR='3']  
[ gluon, gluon, + ; TSP='2', COLOR='8']  
[ ghost, antighost, - ; TSP='-2', COLOR='8']
```

```
[ antiquark, quark, gluon ; QCD='1']  
[ gluon, gluon, gluon ; QCD='1']  
[ gluon, gluon, gluon, gluon ; QCD='2']  
[ antighost, ghost, gluon ; QCD='1']
```

Model Definition

```
[ quark, antiquark, -1 ; TSP='1', COLOR='3']  
[ gluon, gluon, + ; TSP='2', COLOR='8']  
[ ghost, antighost, - ; TSP='-2', COLOR='8']
```

```
[ antiquark, quark, gluon ; QCD='1']  
[ gluon, gluon, gluon ; QCD='1']  
[ gluon, gluon, gluon, gluon ; QCD='2']  
[ antighost, ghost, gluon ; QCD='1']
```

```
class MyModel:  
    _inner: fg.Model  
    _particles: dict[str, MyParticle]  
    def __init__(self, path: str, quiet = True):  
        self._inner = fg.Model.new_qgraf(path, quiet = quiet)  
        self._particles = {  
            p.name(): MyParticle(p) for p in self._inner.particles()  
        }  
    def particle(self, index: int) -> MyParticle:  
        return list(self._particles.values())[index]  
    # Some more methods passing other method calls through to `inner`  
  
class MyParticle:  
    _inner: fg.Particle  
    def __init__(self, p: fg.Particle):  
        self._inner = p  
    def two_spin(self) -> str:  
        return self._inner.constants()["TSP"]  
    # Same as above
```



Model Definition

```
[ quark, antiquark, -1 ; TSP='1', COLOR='3']  
[ gluon, gluon, + ; TSP='2', COLOR='8']  
[ ghost, antighost, - ; TSP='-2', COLOR='8']  
  
[ antiquark, quark, gluon ; QCD='1']  
[ gluon, gluon, gluon ; QCD='1']  
[ gluon, gluon, gluon, gluon ; QCD='2']  
[ antighost, ghost, gluon ; QCD='1']
```

```
class MyModel:  
    _inner: fg.Model  
    _particles: dict[str, MyParticle]  
    def __init__(self, path: str, quiet = True):  
        self._inner = fg.Model.new_qgraf(path, quiet = quiet)  
        self._particles = {  
            p.name(): MyParticle(p) for p in self._inner.particles()  
        }  
    def particle(self, index: int) -> MyParticle:  
        return list(self._particles.values())[index]  
    # Some more methods passing other method calls through to `inner`  
  
class MyParticle:  
    _inner: fg.Particle  
    def __init__(self, p: fg.Particle):  
        self._inner = p  
    def two_spin(self) -> str:  
        return self._inner.constants()["TSP"]  
    # Same as above
```

```
diags = fg.generate_diagrams(["quark", "antiquark"], ["quark", "antiquark"], 1, model = m_custom)  
assert(  
    [(p.particle().name(), p.particle().custom().two_spin()) for p in diags[2].propagators()]  
    ==  
    [('gluon', 2), ('antiquark', 1), ('gluon', 2), ('antiquark', 1)]  
)
```

Diagram Selection

Diagram selection criteria can be applied on two levels

- Purely topology dependent criteria can be applied already during topology generation
 - Momentum-based criteria
 - Node partition criteria
- Model dependent criteria can be applied during diagram generation

Diagram Selection

Diagram selection criteria can be applied on two levels

- Purely topology dependent criteria can be applied already during topology generation
 - Momentum-based criteria
 - Node partition criteria
- Model dependent criteria can be applied during diagram generation

Selection criteria can be set and combined with a `DiagramSelector`

- Several useful predefined filter options
 - Coupling power
 - On-shell external legs
 - Self-loops
- Arbitrary user-defined filters

Diagram Selection: Example

```
[1]: def gluon_in_second_loop(d: fg.Diagram) -> bool:
      return any(
          p.particle().name() == "g" for p in d.chord(1)
      )

s = fg.DiagramSelector()
s.select_on_shell()
s.select_self_loops(0)
s.add_custom_function(gluon_in_second_loop)
len(fg.generate_diagrams(["u", "u~"], ["g", "g"], 2, selector = s))
```

```
[1]: 1230
```

```

[1]: def topo_filter(topo: Topology) -> bool:
    three_loop = any(sum(c != 0 for c in e.momentum()[3:]) == 3 for e in topo.edges())
    two_tadpoles = sum(all(c == 0 for c in e.momentum()) for e in topo.edges()) == 2
    return three_loop and two_tadpoles
def diagram_filter(d: fg.Diagram) -> bool:
    gluon_tadpole = any(
        p.vertex(0).id() == p.vertex(1).id() and p.particle().name() == "g" for p in d.propagators()
    )
    Z_tadpole = any(
        p.vertex(0).id() == p.vertex(1).id() and p.particle().name() == "Z" for p in d.propagators()
    )
    fermion_loop = any(p.particle().is_fermi() for i in range(5) for p in d.chord(i))
    return gluon_tadpole and Z_tadpole and fermion_loop
s = fg.DiagramSelector()
s.select_on_shell()
s.select_self_loops(2)
s.select_vertex_degree(4, 0)
s.select_coupling_power("QED", 2)
s.add_topology_function(topo_filter)
s.add_custom_function(diagram_filter)
fg.DiagramGenerator(["g"], ["t", "t~"], 5, fg.Model(), selector = s).count()

```

[1]: 12809

Automatic Diagram Drawing

Fully automatic diagram drawing using stress majorization [[Gansner, Koren, and North 2005](#)]

- Handles layout generation for arbitrary loop orders
- All drawing handled internally, output based on a unified backend abstraction which only requires the backend to be able to draw text and (filled) Bézier curves

Automatic Diagram Drawing

Fully automatic diagram drawing using stress majorization [[Gansner, Koren, and North 2005](#)]

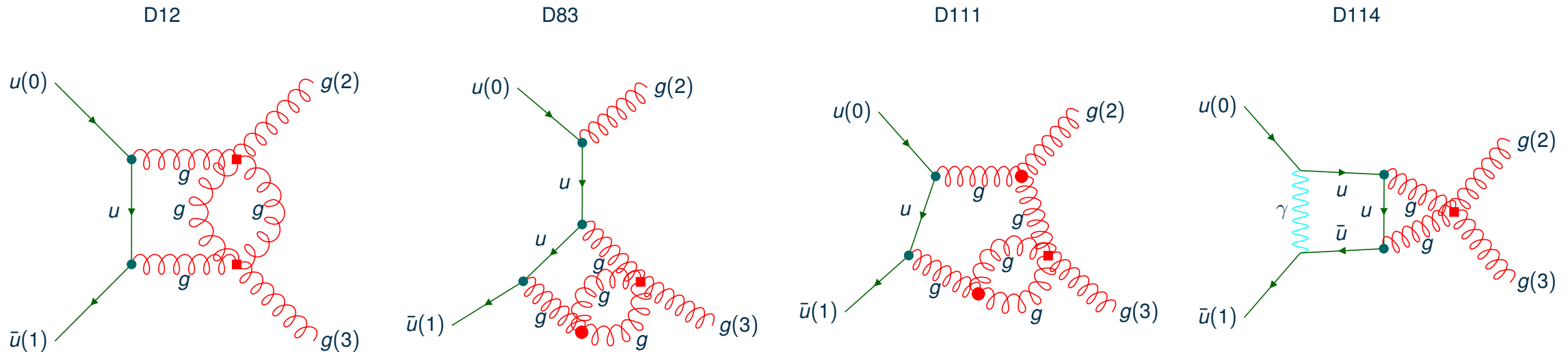
- Handles layout generation for arbitrary loop orders
- All drawing handled internally, output based on a unified backend abstraction which only requires the backend to be able to draw text and (filled) Bézier curves
 - SVG (e.g. displays inline in Jupyter notebooks)
 - $\text{\LaTeX}$ (TikZ) and typst (CetZ) output for direct inclusion into documents
 - More target formats can be implemented trivially

Automatic Diagram Drawing

Fully automatic diagram drawing using stress majorization [Gansner, Koren, and North 2005]

- Handles layout generation for arbitrary loop orders
- All drawing handled internally, output based on a unified backend abstraction which only requires the backend to be able to draw text and (filled) Bézier curves
 - SVG (e.g. displays inline in Jupyter notebooks)
 - $\text{\LaTeX}$ (TikZ) and typst (CetZ) output for direct inclusion into documents
 - More target formats can be implemented trivially

```
[1]: diags.draw_tikz_string([11, 97, 161, 190])
```



Automatic Construction of Amplitudes (Work in Progress)

- Diagrams will usually be translated to a mathematical amplitude expression according to the model as next step
 - Redundant model handling (the model has to be fully understood by FeynGraph *and* the user code)
 - Requires careful synchronization between FeynGraph and the consuming code to ensure splitting and merging of vertices is consistent

Automatic Construction of Amplitudes (Work in Progress)

- Diagrams will usually be translated to a mathematical amplitude expression according to the model as next step
 - Redundant model handling (the model has to be fully understood by FeynGraph *and* the user code)
 - Requires careful synchronization between FeynGraph and the consuming code to ensure splitting and merging of vertices is consistent
- Full model information allows automatic construction of (largely) unoptimized amplitude expressions from diagrams through FeynGraph

Automatic Construction of Amplitudes (Work in Progress)

- Diagrams will usually be translated to a mathematical amplitude expression according to the model as next step
 - Redundant model handling (the model has to be fully understood by FeynGraph *and* the user code)
 - Requires careful synchronization between FeynGraph and the consuming code to ensure splitting and merging of vertices is consistent
- Full model information allows automatic construction of (largely) unoptimized amplitude expressions from diagrams through FeynGraph

```
[8]: amps = fg.amplitude.construct_amplitude(["u", "u~"], ["u", "u~"], "models/QCD_UF0", 0)
      amps
```

```
[8]: [<Amplitude at 0x7fe15b75f0b0>, <Amplitude at 0x7fe15b75dab0>]
```

```
[9]: amps[0]
```

$$[9]: \frac{(-1) i i i g_s g_s \delta_{c_{-1}, c_{-2}} T_{i_3, i_2}^{a_{-2}} T_{i_0, i_1}^{a_{-1}} (\gamma^{\mu_{-2}})_{s_3 s_2} (\gamma^{\mu_{-1}})_{s_0 s_1} \eta^{\mu_{-1} \mu_{-2}} u_{s_0}(p_0, m_u) \bar{u}_{s_2}(p_2, m_u) v_{s_3}(p_3, m_u) \bar{v}_{s_1}(p_1, m_u)}{1((p_0 + p_1)^2 + (-1)0^2)}$$

```
[10]: amps[1]
```

$$[10]: \frac{i i i g_s g_s \delta_{c_{-1}, c_{-2}} T_{i_3, i_1}^{a_{-2}} T_{i_0, i_2}^{a_{-1}} (\gamma^{\mu_{-2}})_{s_3 s_1} (\gamma^{\mu_{-1}})_{s_0 s_2} \eta^{\mu_{-1} \mu_{-2}} u_{s_0}(p_0, m_u) \bar{u}_{s_2}(p_2, m_u) v_{s_3}(p_3, m_u) \bar{v}_{s_1}(p_1, m_u)}{1((p_0 + (-1)p_2)^2 + (-1)0^2)}$$

Automatic Construction of Amplitudes (Work in Progress)

- Diagrams will usually be translated to a mathematical amplitude expression according to the model as next step
 - Redundant model handling (the model has to be fully understood by FeynGraph *and* the user code)
 - Requires careful synchronization between FeynGraph and the consuming code to ensure splitting and merging of vertices is consistent
- Full model information allows automatic construction of (largely) unoptimized amplitude expressions from diagrams through FeynGraph
- All analytic operations handled internally, without any computer algebra system dependency
- Final result is an abstract syntax tree (AST) of the amplitude, which can be easily translated to any desired format (Wolfram, FORM, ...) with little additional work

Automatic Construction of Amplitudes (Work in Progress)

```
struct FormDictionary;
impl Dictionary<UFOAtom, String, FormTranslator> for FormDictionary {
    fn atom(buf: &mut String, atom: &UFOAtom, _ctx: &mut FormTranslator) -> Result<(), std::fmt::Error> {
        match atom {
            UFOAtom::Lorentz(a) => match a {
                LorentzAtom::Epsilon(mu, nu, rho, sigma) => write!(buf, "e_{mu}{nu}, mu{rho}, mu{sigma}"),
                /* ... */
            },
            /* ... */
        }
    }
}

struct FormTranslator;
impl Translator<UFOAtom, FormDictionary> for FormTranslator {
    type Output = String;
    fn translate(&mut self, expr: &expression::Expression<UFOAtom>) -> Result<Self::Output, std::fmt::Error> {
        let mut buf = String::new();
        self.expr(&mut buf, expr)?;
        Ok(buf)
    }
}
```



Command Line Interface

- Python based CLI application to run FeynGraph as a standalone program
- Configuration through a single [TOML](#) file
- Output diagrams are rendered from a customizable [Jinja](#) template

Command Line Interface

- Python based CLI application to run FeynGraph as a standalone program
- Configuration through a single [TOML](#) file
- Output diagrams are rendered from a customizable [Jinja](#) template

Example configuration [example.toml]

```
template = "json.jinja"
```

[process]

```
in = ["u", "u~"]
```

```
out = ["g", "g"]
```

```
loops = 2
```

[filter]

```
on_shell = true
```

```
self_loops = 0
```

```
$ feyngraph example.toml -o diagrams.json
```

Summary and Outlook

- High-performance and flexible Feynman diagram generator
 - ready to be used with effective field theories and BSM models
 - ready to be used in automation workflows
- Soon: Automatic translation of diagrams to analytic expressions

Summary and Outlook

- High-performance and flexible Feynman diagram generator
 - ready to be used with effective field theories and BSM models
 - ready to be used in automation workflows
- Soon: Automatic translation of diagrams to analytic expressions

Possible future improvements

- Interfaces to other programming languages, notably C to bridge the gap to C, C++, Fortran, Mathematica, ...
- Improve quality of diagram drawing
- Improve speed of diagram generation

Summary and Outlook

- High-performance and flexible Feynman diagram generator
 - ready to be used with effective field theories and BSM models
 - ready to be used in automation workflows
- Soon: Automatic translation of diagrams to analytic expressions

Possible future improvements

- Interfaces to other programming languages, notably C to bridge the gap to C, C++, Fortran, Mathematica, ...
- Improve quality of diagram drawing
- Improve speed of diagram generation

Source Code

<https://github.com/Jens-Braun/FeynGraph>

User Guide + Python API documentation

<https://jens-braun.github.io/FeynGraph/>

Thank you for for you attention!

Backup: Vacuum Diagrams

- Supports generation of vacuum diagrams out of the box

```
[1]: def no_quark(d: fg.Diagram) -> bool:
      return all(abs(p.particle().pdg()) > 6 for p in d.propagators())
s = fg.DiagramSelector()
s.select_coupling_power("QED", 0)
s.select_tadpoles(0)
s.add_custom_function(no_quark)
len(fg.DiagramGenerator([], [], 4, fg.Model(), selector = s).generate())
```

[1]: 65

⁵Example from [Navarrete and Schröder 2024](#)