Automation of analytical calculations in particle physics and gravity with Redberry CAS

Stanislav Poslavsky

Institute for High Energy Physics, Protvino, Russia

23 January 2016

Plan

1- Inroduction

2- `Redberry` CAS

3- Deriving Feynman rules from arbitrary Lagrangians

4- Doing NLO calculations

5- Conclusions

Introduction

Which types of mathematical objects arise in HEP?

$$\mathcal{S} = \int d^4x\sqrt{-g}\left(\mathcal{R}/\kappa^2 - i\bar{\psi}\left(i\gamma^a e_a^{\mu}\left\{\partial_{\mu} + \frac{1}{4}\omega_{\mu}^{ab}[\gamma_a,\gamma_b]\right\} - m\right)\psi\right)$$

Vectors	*Spinors*	*Matrices*	*Tensors*
$x_\mu, p_\mu,$	$\psi(x), \bar{\psi}(x)$	$\sigma_\mu, \gamma_\mu,$	$g_{\mu\nu}, \mathcal{R}^a_{b\mu\nu},$
$\epsilon_\mu, \ldots$	$\bar{u}(p), v(p)$	T_a	$\epsilon_{\mu\nu\alpha\beta}$

Introduction

Which types of mathematical objects arise in HEP?

$$\mathcal{S} = \int d^4x \sqrt{-g} \left(\mathcal{R}/\kappa^2 - i\bar{\psi} \left(i\gamma^a e_a^\mu \left\{ \partial_\mu + \frac{1}{4}\omega_\mu^{ab}[\gamma_a, \gamma_b] \right\} - m \right) \psi \right)$$

Vectors	*Spinors*	*Matrices*	*Tensors*
$x_\mu, p_\mu,$	$\psi(x), \bar{\psi}(x)$	$\sigma_\mu, \gamma_\mu,$	$g_{\mu\nu}, \mathcal{R}^a_{b\mu\nu},$
$\epsilon_\mu, \ldots$	$\bar{u}(p), v(p)$	T_a	$\epsilon_{\mu\nu\alpha\beta}$

Introduction

Which types of mathematical objects arise in HEP?

$$\mathcal{S} = \int d^4x\sqrt{-g}\left(\mathcal{R}/\kappa^2 - i\bar{\psi}\left(i\gamma^a e_a^{\mu}\left\{\partial_{\mu} + \frac{1}{4}\omega_{\mu}^{ab}[\gamma_a,\gamma_b]\right\} - m\right)\psi\right)$$

Vectors	*Spinors*	*Matrices*	*Tensors*
$x_\mu, p_\mu,$	$\psi(x), \bar{\psi}(x)$	$\sigma_\mu, \gamma_\mu,$	$g_{\mu\nu}, \mathcal{R}^a_{b\mu\nu},$
$\epsilon_\mu, \ldots$	$\bar{u}(p), v(p)$	T_a	$\epsilon_{\mu\nu\alpha\beta}$

Introduction

Which types of mathematical objects arise in HEP?

$$\mathcal{S} = \int d^4x\sqrt{-g}\left(\mathcal{R}/\kappa^2 - i\bar{\psi}\left(i\gamma^a e_a^\mu\left\{\partial_\mu + \frac{1}{4}\omega_\mu^{ab}[\gamma_a,\gamma_b]\right\} - m\right)\psi\right)$$

Vectors	*Spinors*	*Matrices*	*Tensors*
$x_\mu, p_\mu,$ $\epsilon_\mu, \ldots$	$\psi(x), \bar{\psi}(x)$ $\bar{u}(p), v(p)$	$\sigma_\mu, \gamma_\mu,$ T_a	$g_{\mu\nu}, \mathcal{R}^a_{b\mu\nu},$ $\epsilon_{\mu\nu\alpha\beta}$

Introduction

Which types of mathematical objects arise in HEP?

$$\mathcal{S} = \int d^4x\sqrt{-g}\left(\mathcal{R}/\kappa^2 - i\bar{\psi}\left(i\gamma^a e_a^\mu\left\{\partial_\mu + \frac{1}{4}\omega_\mu^{ab}[\gamma_a,\gamma_b]\right\} - m\right)\psi\right)$$

Vectors	*Spinors*	*Matrices*	*Tensors*
$x_\mu, p_\mu,$	$\psi(x), \bar{\psi}(x)$	$\sigma_\mu, \gamma_\mu,$	$g_{\mu\nu}, \mathcal{R}^a_{b\mu\nu},$
$\epsilon_\mu, \ldots$	$\bar{u}(p), v(p)$	T_a	$\epsilon_{\mu\nu\alpha\beta}$

Introduction

Which types of mathematical objects arise in HEP?

$$\mathcal{S} = \int d^4x\sqrt{-g}\left(\mathcal{R}/\kappa^2 - i\bar{\psi}\left(i\gamma^a e_a^{\mu}\left\{\partial_{\mu} + \frac{1}{4}\omega_{\mu}^{ab}[\gamma_a,\gamma_b]\right\} - m\right)\psi\right)$$

Vectors	*Spinors*	*Matrices*	*Tensors*
$x_\mu, p_\mu,$	$\psi(x), \bar{\psi}(x)$	$\sigma_\mu, \gamma_\mu,$	$g_{\mu\nu}, \mathcal{R}^a_{b\mu\nu},$
$\epsilon_\mu, \ldots$	$\bar{u}(p), v(p)$	T_a	$\epsilon_{\mu\nu\alpha\beta}$

Objects with indices

Introduction

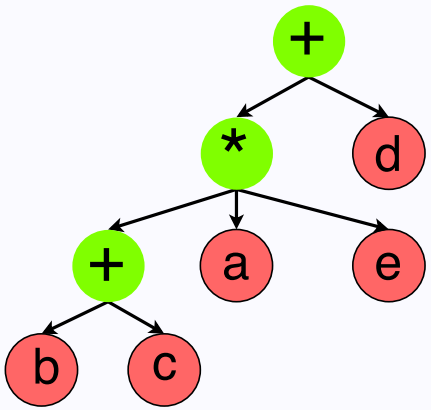
Which types of mathematical objects arise in HEP?

$$\mathcal{S} = \int d^4x\sqrt{-g}\left(\mathcal{R}/\kappa^2 - i\bar{\psi}\left(i\gamma^a e_a^{\mu}\left\{\partial_{\mu} + \frac{1}{4}\omega_{\mu}^{ab}[\gamma_a,\gamma_b]\right\} - m\right)\psi\right)$$

Vectors	*Spinors*	*Matrices*	*Tensors*
$x_\mu, p_\mu,$	$\psi(x), \bar{\psi}(x)$	$\sigma_\mu, \gamma_\mu,$	$g_{\mu\nu}, \mathcal{R}^a_{b\mu\nu},$
$\epsilon_\mu, \ldots$	$\bar{u}(p), v(p)$	T_a	$\epsilon_{\mu\nu\alpha\beta}$

Objects with indices

Data structures for scalars:



$(b + c) \times a \times e + d$

Introduction

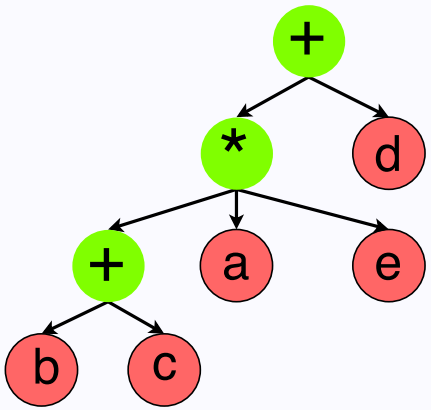
Which types of mathematical objects arise in HEP?

$$\mathcal{S} = \int d^4x\sqrt{-g}\left(\mathcal{R}/\kappa^2 - i\bar{\psi}\left(i\gamma^a e_a^\mu\left\{\partial_\mu + \frac{1}{4}\omega_\mu^{ab}[\gamma_a,\gamma_b]\right\} - m\right)\psi\right)$$

Vectors	*Spinors*	*Matrices*	*Tensors*
$x_\mu, p_\mu,$ $\epsilon_\mu, \ldots$	$\psi(x), \bar{\psi}(x)$ $\bar{u}(p), v(p)$	$\sigma_\mu, \gamma_\mu,$ T_a	$g_{\mu\nu}, \mathcal{R}^a_{b\mu\nu},$ $\epsilon_{\mu\nu\alpha\beta}$

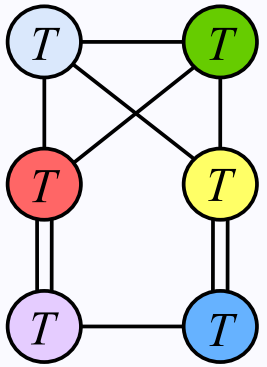
Objects with indices

Data structures for scalars:



$(b+c)\times a\times e+d$

Data structures for tensors:



$T_{abmj}\ T_{defi}\ T_{abc}\ T_{cef}\ T_{inm}\ T_{djn}$

Redberry

▶ Dummy indices and graph structure make usual tree-based CASs unnatural and at least very hard to use with indexed objects

✓ **`Redberry` treats all indexed expressions in a natural way:**

Define symmetries: $R_{abcd} = R_{cdab}$

```
addSymmetries 'R_abcd', [[0, 2], [1, 3]].p
```

Input expressions: $R_{abcd}R^{abcd} - T_{klmn}T^{mnkl}$

```
expr = 'R_abcd*R^cdab - T_klmn*T^mnkl'.t
```

Apply substitutions: $T_{pqrs} \to R_{pqrs}$

```
expr <<= 'T_pqrs = R_pqrs'.t
```

Simplify: $R_{abcd}R^{abcd} - R_{klmn}R^{mnkl} = 0$

```
println expr
```

▷ 0

- Input and output in a natural syntax close to LaTeX
- Indices are automatically treated as patterns; names of dummies are irrelevant
- All algebraic operations (`Expand`, `Differentiate` etc.) work with tensors
- Powerfull simplification algorithms
- Fast internal engine based on graph-theoretical approach

Deriving Feynman rules

EXAMPLE:

Let's derive Feynman rules for some compicated theory, e.g.:

$$\mathcal{L} = \sqrt{-g}\left(-\frac{c}{\kappa^2}\mathcal{R} - b\mathcal{R}^2 + a\mathcal{R}_{\mu\nu}\mathcal{R}^{\mu\nu} + \gamma\,\mathcal{R}\phi^2 + g^{\mu\nu}\partial_\mu\phi\partial_\nu\phi - m^2\phi^2\right),$$

$$\begin{aligned}
\mathcal{R} &= g^{\mu\nu}\mathcal{R}_{\mu\nu}, \\
\mathcal{R}_{\mu\nu} &= \mathcal{R}^{\sigma}{}_{\mu\sigma\nu}, \\
\mathcal{R}^{\lambda}{}_{\rho\alpha\beta} &= \partial_\alpha\Gamma^{\lambda}{}_{\rho\beta} - \partial_\beta\Gamma^{\lambda}{}_{\rho\alpha} + \Gamma^{\lambda}{}_{\alpha\sigma}\Gamma^{\sigma}{}_{\rho\beta} - \Gamma^{\lambda}{}_{\beta\sigma}\Gamma^{\sigma}{}_{\rho\alpha}, \\
\Gamma^{\lambda}{}_{\mu\nu} &= \frac{1}{2}g^{\lambda\sigma}\left(\partial_\mu g_{\sigma\nu} + \partial_\nu g_{\sigma\mu} - \partial_\sigma g_{\mu\nu}\right), \\
&\ldots
\end{aligned}$$

Deriving Feynman rules

- **Step 1:** Input the model

```
1   setSymmetric 'h_ab[x_a]'
2   L = 'det * (Lg + Lf + Li + ...)'.t
3   //definitions
4   Lg = 'Lg = -c*R/k**2 - b*R**2 + a*R_ab*R_cd*m^ac[x_a]*m^bd[x_a]'.t
5   Lf = 'Lf = m^ab[x_a]*f~(1)_a[x_a]*f~(1)_b[x_a] - m**2*f[x_a]**2'.t
6   Li = 'Li  = g*R*f[x_a]**2'.t
7   Ri = 'R^a_bcd = G~(1)^{a}_{db c}[x_a] - ...'.t
8   Chr = ... ; Ric = ...; R = ...;

10  //weak field
11  WF = 'm^ab[x_a] = g^ab - h^ab[x_a] + h^a_d[x_a]*h^db[x_a] + ...'.t
12  DT = 'det = 1 + h^a_a[x_a] + ...'.t

14  //subtitute all into the Lagrangian
15  L <<= Lg & Lf & Li & R & Ric & Ri & Chr & WF & DT & ...
```

Deriving Feynman rules

- **Step 1:** Input the model

```
1   setSymmetric 'h_ab[x_a]'
2   L = 'det * (Lg + Lf + Li + ...)'.t
3   //definitions
4   Lg = 'Lg = -c*R/k**2 - b*R**2 + a*R_ab*R_cd*m^ac[x_a]*m^bd[x_a]'.t
5   Lf = 'Lf = m^ab[x_a]*f~(1)_a[x_a]*f~(1)_b[x_a] - m**2*f[x_a]**2'.t
6   Li = 'Li  = g*R*f[x_a]**2'.t
7   Ri = 'R^a_bcd = G~(1)^{a}_{db c}[x_a] - ...'.t
8   Chr = ... ; Ric = ...; R = ...;

10  //weak field
11  WF = 'm^ab[x_a] = g^ab - h^ab[x_a] + h^a_d[x_a]*h^db[x_a] + ...'.t
12  DT = 'det = 1 + h^a_a[x_a] + ...'.t

14  //subtitute all into the Lagrangian
15  L <<= Lg & Lf & Li & R & Ric & Ri & Chr & WF & DT & ...
```

  - `setSymmetric` tells Redberry that `h_ab[p_a]` is symmetric

Deriving Feynman rules

- **Step 1:** Input the model

```
1   setSymmetric 'h_ab[x_a]'
2   L = 'det * (Lg + Lf + Li + ...)'.t
3   //definitions
4   Lg = 'Lg = -c*R/k**2 - b*R**2 + a*R_ab*R_cd*m^ac[x_a]*m^bd[x_a]'.t
5   Lf = 'Lf = m^ab[x_a]*f~(1)_a[x_a]*f~(1)_b[x_a] - m**2*f[x_a]**2'.t
6   Li = 'Li  = g*R*f[x_a]**2'.t
7   Ri = 'R^a_bcd = G~(1)^{a}_{db c}[x_a] - ...'.t
8   Chr = ... ; Ric = ...; R = ...;

10  //weak field
11  WF = 'm^ab[x_a] = g^ab - h^ab[x_a] + h^a_d[x_a]*h^db[x_a] + ...'.t
12  DT = 'det = 1 + h^a_a[x_a] + ...'.t

14  //subtitute all into the Lagrangian
15  L <<= Lg & Lf & Li & R & Ric & Ri & Chr & WF & DT & ...
```

- `setSymmetric` tells Redberry that `h_ab[p_a]` is symmetric
- `m^ab[x_a]` is field of metric;

Deriving Feynman rules

- **Step 1:** Input the model

```
1   setSymmetric 'h_ab[x_a]'
2   L = 'det * (Lg + Lf + Li + ...)'.t
3   //definitions
4   Lg = 'Lg = -c*R/k**2 - b*R**2 + a*R_ab*R_cd*m^ac[x_a]*m^bd[x_a]'.t
5   Lf = 'Lf = m^ab[x_a]*f~(1)_a[x_a]*f~(1)_b[x_a] - m**2*f[x_a]**2'.t
6   Li = 'Li  = g*R*f[x_a]**2'.t
7   Ri = 'R^a_bcd = G~(1)^{a}_{db c}[x_a] - ...'.t
8   Chr = ... ; Ric = ...; R = ...;

10  //weak field
11  WF = 'm^ab[x_a] = g^ab - h^ab[x_a] + h^a_d[x_a]*h^db[x_a] + ...'.t
12  DT = 'det = 1 + h^a_a[x_a] + ...'.t

14  //subtitute all into the Lagrangian
15  L <<= Lg & Lf & Li & R & Ric & Ri & Chr & WF & DT & ...
```

- `setSymmetric` tells Redberry that `h_ab[p_a]` is symmetric
- `m^ab[x_a]` is field of metric; `g^ab` is a built-in Minkowski metric

Deriving Feynman rules

- **Step 1:** Input the model

```
1   setSymmetric 'h_ab[x_a]'
2   L = 'det * (Lg + Lf + Li + ...)'.t
3   //definitions
4   Lg = 'Lg = -c*R/k**2 - b*R**2 + a*R_ab*R_cd*m^ac[x_a]*m^bd[x_a]'.t
5   Lf = 'Lf = m^ab[x_a]*f~(1)_a[x_a]*f~(1)_b[x_a] - m**2*f[x_a]**2'.t
6   Li = 'Li  = g*R*f[x_a]**2'.t
7   Ri = 'R^a_bcd = G~(1)^{a}_{db c}[x_a] - ...'.t
8   Chr = ... ; Ric = ...; R = ...;

10  //weak field
11  WF = 'm^ab[x_a] = g^ab - h^ab[x_a] + h^a_d[x_a]*h^db[x_a] + ...'.t
12  DT = 'det = 1 + h^a_a[x_a] + ...'.t

14  //subtitute all into the Lagrangian
15  L <<= Lg & Lf & Li & R & Ric & Ri & Chr & WF & DT & ...
```

- `setSymmetric` tells Redberry that `h_ab[p_a]` is symmetric
- `m^ab[x_a]` is field of metric; `g^ab` is a built-in Minkowski metric
- `f~(1)_b[x_a]` is a short for $\partial_b f(x_a)$

Deriving Feynman rules

- **Step 1:** Input the model

```
1   setSymmetric 'h_ab[x_a]'
2   L = 'det * (Lg + Lf + Li + ...)'.t
3   //definitions
4   Lg = 'Lg = -c*R/k**2 - b*R**2 + a*R_ab*R_cd*m^ac[x_a]*m^bd[x_a]'.t
5   Lf = 'Lf = m^ab[x_a]*f~(1)_a[x_a]*f~(1)_b[x_a] - m**2*f[x_a]**2'.t
6   Li = 'Li  = g*R*f[x_a]**2'.t
7   Ri = 'R^a_bcd = G~(1)^{a}_{db c}[x_a] - ...'.t
8   Chr = ... ; Ric = ...; R = ...;

10  //weak field
11  WF = 'm^ab[x_a] = g^ab - h^ab[x_a] + h^a_d[x_a]*h^db[x_a] + ...'.t
12  DT = 'det = 1 + h^a_a[x_a] + ...'.t

14  //subtitute all into the Lagrangian
15  L <<= Lg & Lf & Li & R & Ric & Ri & Chr & WF & DT & ...
```

- `setSymmetric` tells Redberry that `h_ab[p_a]` is symmetric
- `m^ab[x_a]` is field of metric; `g^ab` is a built-in Minkowski metric
- `f~(1)_b[x_a]` is a short for $\partial_b f(x_a)$
- `G~(1)^a_dbc[x_a]` is a short for Christoffel $\partial_c \Gamma^a_{db}$

Deriving Feynman rules

- **Step 2:** Fourier transform
- Select terms up to e.g. cubic interactions ($\sim hhh$, $\sim hh\phi$, $\sim h\phi\phi$):

```
16  $fields = ['h_ab[x_a]', 'f[x_a]'].t
17  $Degree = { expr -> Count(expr, 1, $fields, true) }

19  SelectCubic = { expr ->
20      expr.class == Product && $Degree(expr) > 3 ? 0.t : expr
21  }

23  //do "clever" expand
24  L <<= ExpandAndEliminate[SelectCubic]
```

Deriving Feynman rules

- **Step 2:** Fourier transform
- Select terms up to e.g. cubic interactions ($\sim hhh$, $\sim hh\phi$, $\sim h\phi\phi$):

```
16  $fields = ['h_ab[x_a]', 'f[x_a]'].t
17  $Degree = { expr -> Count(expr, 1, $fields, true) }

19  SelectCubic = { expr ->
20      expr.class == Product && $Degree(expr) > 3 ? 0.t : expr
21  }

23  //do "clever" expand
24  L <<= ExpandAndEliminate[SelectCubic]
```

  - `{x -> ...}` defines a function using closure

Deriving Feynman rules

- **Step 2:** Fourier transform
- Select terms up to e.g. cubic interactions ($\sim hhh$, $\sim hh\phi$, $\sim h\phi\phi$):

```
16  $fields = ['h_ab[x_a]', 'f[x_a]'].t
17  $Degree = { expr -> Count(expr, 1, $fields, true) }

19  SelectCubic = { expr ->
20      expr.class == Product && $Degree(expr) > 3 ? 0.t : expr
21  }

23  //do "clever" expand
24  L <<= ExpandAndEliminate[SelectCubic]
```

  - `{x -> ...}` defines a function using closure
  - `ExpandAndEliminate[tr]` applies `tr` at each intermediate step

Deriving Feynman rules

- **Step 2:** Fourier transform
- Select terms up to e.g. cubic interactions ($\sim hhh$, $\sim hh\phi$, $\sim h\phi\phi$):

```
16  $fields = ['h_ab[x_a]', 'f[x_a]'].t
17  $Degree = { expr -> Count(expr, 1, $fields, true) }

19  SelectCubic = { expr ->
20      expr.class == Product && $Degree(expr) > 3 ? 0.t : expr
21  }

23  //do "clever" expand
24  L <<= ExpandAndEliminate[SelectCubic]
```

  - `{x -> ...}` defines a function using closure
  - `ExpandAndEliminate[tr]` applies `tr` at each intermediate step
  - This will speed up calculation up to $\sim 10^2$ times

Deriving Feynman rules

- **Step 2:** Fourier transform
- Select terms up to e.g. cubic interactions ($\sim hhh$, $\sim hh\phi$, $\sim h\phi\phi$):

```
16  $fields = ['h_ab[x_a]', 'f[x_a]'].t
17  $Degree = { expr -> Count(expr, 1, $fields, true) }

19  SelectCubic = { expr ->
20      expr.class == Product && $Degree(expr) > 3 ? 0.t : expr
21  }

23  //do "clever" expand
24  L <<= ExpandAndEliminate[SelectCubic]
```

  - `{x -> ...}` defines a function using closure
  - `ExpandAndEliminate[tr]` applies `tr` at each intermediate step
  - This will speed up calculation up to $\sim 10^2$ times

- Finally, apply Fourier transform:

```
25  L <<= LagrangeFourier
26  println L
```

```
▷ h_{ab}[k_a]*h_{cd}[-k_a]*k^a*k^c*g^bd... + h_{ab}[k_a]*f[p_a]*
      f[-p_a-k_a]*p^a*k^b... +...
```

Deriving Feynman rules

- **Step 3:** Derive propagators
- Select quadratic part of the Lagrangian:

```
27  $hh = L
28  $hh <<= Differentiate['h_ab[p_a]', 'h_mn[-p_a]']
29  $hh <<= 'h_ab[p_a] = 0'.t & 'f[p_a] = 0'.t
30  $hh <<= ApplyDiracDeltas & 'DiracDelta[0] = 1'.t

32  //bind inverse propagator
33  $iP = 'iP^abcd[p_a]'.eq $hh
```

Deriving Feynman rules

- **Step 3:** Derive propagators
- Select quadratic part of the Lagrangian:

```
27  $hh = L
28  $hh <<= Differentiate['h_ab[p_a]', 'h_mn[-p_a]']
29  $hh <<= 'h_ab[p_a] = 0'.t & 'f[p_a] = 0'.t
30  $hh <<= ApplyDiracDeltas & 'DiracDelta[0] = 1'.t

32  //bind inverse propagator
33  $iP = 'iP^abcd[p_a]'.eq $hh
```

- `Differentiate` takes into account symetries and function arguments:

$$\frac{\delta \texttt{h\_ab[k\_a]}}{\delta \texttt{h\^{}mn[p\_a]}} = \frac{1}{2}\texttt{*(g\_am*g\_bn + g\_an*g\_bm)*DiracDelta[k\_a, p\_a]}$$

Deriving Feynman rules

- **Step 3:** Derive propagators
- Select quadratic part of the Lagrangian:

```
27  $hh = L
28  $hh <<= Differentiate['h_ab[p_a]', 'h_mn[-p_a]']
29  $hh <<= 'h_ab[p_a] = 0'.t & 'f[p_a] = 0'.t
30  $hh <<= ApplyDiracDeltas & 'DiracDelta[0] = 1'.t

32  //bind inverse propagator
33  $iP = 'iP^abcd[p_a]'.eq $hh
```

- `Differentiate` takes into account symetries and function arguments:

$$\frac{\delta \texttt{h\_ab[k\_a]}}{\delta \texttt{h\^{}mn[p\_a]}} = \frac{1}{2}\texttt{*(g\_am*g\_bn + g\_an*g\_bm)*DiracDelta[k\_a, p\_a]}$$

- `ApplyDiracDeltas` just removes delta-functions in appropriate way:

```
DiracDelta[k_a, p_a]*f[k_a]*... = f[p_a]*...
```

Deriving Feynman rules

- **Step 3:** Derive propagators
- Select quadratic part of the Lagrangian:

```
27  $hh = L
28  $hh <<= Differentiate['h_ab[p_a]', 'h_mn[-p_a]']
29  $hh <<= 'h_ab[p_a] = 0'.t & 'f[p_a] = 0'.t
30  $hh <<= ApplyDiracDeltas & 'DiracDelta[0] = 1'.t

32  //bind inverse propagator
33  $iP = 'iP^abcd[p_a]'.eq $hh
```

- `Differentiate` takes into account symetries and function arguments:

$$\frac{\delta \texttt{h\_ab[k\_a]}}{\delta \texttt{h\^{}mn[p\_a]}} = \frac{1}{2}\texttt{*(g\_am*g\_bn + g\_an*g\_bm)*DiracDelta[k\_a, p\_a]}$$

- `ApplyDiracDeltas` just removes delta-functions in appropriate way:

```
DiracDelta[k_a, p_a]*f[k_a]*... = f[p_a]*...
```

- `A.eq B` creates substitution `A = B`

Deriving Feynman rules

- **Step 3:** Derive propagators
- Select quadratic part of the Lagrangian:

```
27  $hh = L
28  $hh <<= Differentiate['h_ab[p_a]', 'h_mn[-p_a]']
29  $hh <<= 'h_ab[p_a] = 0'.t & 'f[p_a] = 0'.t
30  $hh <<= ApplyDiracDeltas & 'DiracDelta[0] = 1'.t

32  //bind inverse propagator
33  $iP = 'iP^abcd[p_a]'.eq $hh
```

  - `Differentiate` takes into account symetries and function arguments:

    $$\frac{\delta \texttt{h\_ab[k\_a]}}{\delta \texttt{h\^{}mn[p\_a]}} = \frac{1}{2}\texttt{*(g\_am*g\_bn + g\_an*g\_bm)*DiracDelta[k\_a, p\_a]}$$

  - `ApplyDiracDeltas` just removes delta-functions in appropriate way:

    ```
    DiracDelta[k_a, p_a]*f[k_a]*... = f[p_a]*...
    ```

  - `A.eq B` creates substitution `A = B`

- So, having the inverse propagator, the propagator itself is determined by the equation:

$$P^{(-1)}{}_{\mu\nu\,\alpha\beta}\, P^{\mu\nu\,\gamma\delta} = \left(\delta^{\gamma}_{\alpha}\delta^{\delta}_{\beta} + \delta^{\gamma}_{\beta}\delta^{\delta}_{\alpha}\right)/2$$

Deriving Feynman rules

- **Step 3:** Derive propagators
- Solve equation with `Redberry` using `Mathematica` as symbolic processor

```
34  addSymmetries 'P_abcd[k_a]', [[0,1]].p, [[0,2], [1,3]].p
35  $eq = 'iP^abcd[k_a]*P_abpq[k_a] = (d^c_p*d^d_q + d^c_q*d^d_p)/2'.t
36  $eq <<= iP //substitute the inverse propagator

38  $opts= [Transformations: 'd^n_n = 4'.t,
39          ExternalSolver: [Solver: 'Mathematica', Path: '/usr/bin/']]
40  $hhPropagator = Reduce([$eq], ['P_abcd[k_a]'], $opts)
41  println $hhPropagator
```

```
▷ P_{abcd}[k_{a}] = 2*(-c+(-6*ka**2*b+2*a*ka**2)*k_{g}*k^{g}...
```

Deriving Feynman rules

- **Step 3:** Derive propagators
- Solve equation with `Redberry` using `Mathematica` as symbolic processor

```
34  addSymmetries 'P_abcd[k_a]', [[0,1]].p, [[0,2], [1,3]].p
35  $eq = 'iP^abcd[k_a]*P_abpq[k_a] = (d^c_p*d^d_q + d^c_q*d^d_p)/2'.t
36  $eq <<= iP //substitute the inverse propagator

38  $opts= [Transformations: 'd^n_n = 4'.t,
39          ExternalSolver: [Solver: 'Mathematica', Path: '/usr/bin/']]
40  $hhPropagator = Reduce([$eq], ['P_abcd[k_a]'], $opts)
41  println $hhPropagator
```

```
▷ P_{abcd}[k_{a}] = 2*(-c+(-6*ka**2*b+2*a*ka**2)*k_{g}*k^{g}...
```

- `addSymmetries` set's up symmetries `P_{abcd}=P_{abcd}=P_{cdab}`

Deriving Feynman rules

- **Step 3:** Derive propagators
- Solve equation with `Redberry` using `Mathematica` as symbolic processor

```
34  addSymmetries 'P_abcd[k_a]', [[0,1]].p, [[0,2], [1,3]].p
35  $eq = 'iP^abcd[k_a]*P_abpq[k_a] = (d^c_p*d^d_q + d^c_q*d^d_p)/2'.t
36  $eq <<= iP //substitute the inverse propagator

38  $opts= [Transformations: 'd^n_n = 4'.t,
39          ExternalSolver: [Solver: 'Mathematica', Path: '/usr/bin/']]
40  $hhPropagator = Reduce([$eq], ['P_abcd[k_a]'], $opts)
41  println $hhPropagator
```

```
▷ P_{abcd}[k_{a}] = 2*(-c+(-6*ka**2*b+2*a*ka**2)*k_{g}*k^{g}...
```

- `addSymmetries` set's up symmetries `P_{abcd}=P_{abcd}=P_{cdab}`
- `Reduce` generates tensor taking into account symmetries:

```
P_{abcd}[k_a] = C0*(g_ab*k_c*k_d + g_cd*k_a*k_b) + ...
```

then generates a scalar system from `[$eq]` and solves it with respect to `C0, C1...` using `Mathematica` solver

Deriving Feynman rules

- **Step 3:** Derive propagators
- Solve equation with `Redberry` using `Mathematica` as symbolic processor

```
34  addSymmetries 'P_abcd[k_a]', [[0,1]].p, [[0,2], [1,3]].p
35  $eq = 'iP^abcd[k_a]*P_abpq[k_a] = (d^c_p*d^d_q + d^c_q*d^d_p)/2'.t
36  $eq <<= iP //substitute the inverse propagator

38  $opts= [Transformations: 'd^n_n = 4'.t,
39          ExternalSolver: [Solver: 'Mathematica', Path: '/usr/bin/']]
40  $hhPropagator = Reduce([$eq], ['P_abcd[k_a]'], $opts)
41  println $hhPropagator
```

```
▷ P_{abcd}[k_{a}] = 2*(-c+(-6*ka**2*b+2*a*ka**2)*k_{g}*k^{g}...
```

- `addSymmetries` set's up symmetries `P_{abcd}=P_{abcd}=P_{cdab}`
- `Reduce` generates tensor taking into account symmetries:

```
P_{abcd}[k_a] = C0*(g_ab*k_c*k_d + g_cd*k_a*k_b) + ...
```

then generates a scalar system from `[$eq]` and solves it with respect to `C0, C1...` using `Mathematica` solver

- We control the spacetime dimension just by applying `d^n_n = 4`

Deriving Feynman rules

- **Step 3:** Derive propagators
- Solve equation with `Redberry` using `Mathematica` as symbolic processor

```
34  addSymmetries 'P_abcd[k_a]', [[0,1]].p, [[0,2], [1,3]].p
35  $eq = 'iP^abcd[k_a]*P_abpq[k_a] = (d^c_p*d^d_q + d^c_q*d^d_p)/2'.t
36  $eq <<= iP //substitute the inverse propagator

38  $opts= [Transformations: 'd^n_n = 4'.t,
39          ExternalSolver: [Solver: 'Mathematica', Path: '/usr/bin/']]
40  $hhPropagator = Reduce([$eq], ['P_abcd[k_a]'], $opts)
41  println $hhPropagator
```

```
▷ P_{abcd}[k_{a}] = 2*(-c+(-6*ka**2*b+2*a*ka**2)*k_{g}*k^{g}...
```

$$P_{abcd}(k) = \frac{a\kappa^2 k^2 + c - 2\kappa^2 f^2}{2f^2 k^4 (a\kappa^2 k^2 + c)} (g_{ac}k_b k_d + g_{bc}k_a k_d + g_{bd}k_a k_c + g_{ad}k_b k_c)$$
$$+ \frac{(g_{ac}g_{bd} + g_{ad}g_{bc})}{(a\kappa^2 k^2 + c)\kappa^2 k^2} + \frac{\kappa^2(-\kappa^2 a k^2 + c + 4b\kappa^2 k^2) g_{ab}g_{cd}}{(\kappa^2 a k^2 + c)(2\kappa^2(a-3b)k^2 - c)k^2}$$
$$+ \frac{2\kappa^4(a-2b)\left(g_{cd}k_a k_b + g_{ab}k_c k_d + 2k_a k_b k_c k_d/k^2\right)}{(a\kappa^2 k^2 + c)(2\kappa^2(a-3b)k^2 - c)k^2}$$

- *Looks not easy to find that by hand!*

Deriving Feynman rules

- **Step 4:** Derive vertices
- Do the same to select *cubic* part of the Lagrangian:

```
42  $hff = L
43  $hff <<= Differentiate['h_ab[k_a]', 'f[p_a]', 'f[q_a]']
44  $hff <<= 'h_ab[p_a] = 0'.t & 'f[p_a] = 0'.t
45  $hff <<= ApplyDiracDeltas

47  //bind vertex
48  $v3 = 'V3^ab[p_a, q_a, k_a]'.eq $hff
49  println $v3
```

```
▷ V3^ab[p_a, q_a, k_a] = (4*g-1)*p^{l}*q_{l}*g^{ab}-m**2...
```

Deriving Feynman rules

- **Step 4:** Derive vertices
- Do the same to select *cubic* part of the Lagrangian:

```
42  $hff = L
43  $hff <<= Differentiate['h_ab[k_a]', 'f[p_a]', 'f[q_a]']
44  $hff <<= 'h_ab[p_a] = 0'.t & 'f[p_a] = 0'.t
45  $hff <<= ApplyDiracDeltas

47  //bind vertex
48  $v3 = 'V3^ab[p_a, q_a, k_a]'.eq $hff
49  println $v3
```

```
▷ V3^ab[p_a, q_a, k_a] = (4*g-1)*p^{l}*q_{l}*g^{ab}-m**2...
```

- The result reads:

$$V^{ab}(p,q) = \quad \alpha\beta \quad = -2g\left(q^a q^b + p^a p^b\right) +$$

$$+ (1-2g)\left(q^a p^b + q^b p^a\right) + \left(2g(q^2+p^2) + (4g-1)\,q_l p^l - m^2\right) g^{ab}$$

Deriving Feynman rules

- **Step 4:** Derive vertices
- Do the same to select *cubic* part of the Lagrangian:

```
42  $hff = L
43  $hff <<= Differentiate['h_ab[k_a]', 'f[p_a]', 'f[q_a]']
44  $hff <<= 'h_ab[p_a] = 0'.t & 'f[p_a] = 0'.t
45  $hff <<= ApplyDiracDeltas

47  //bind vertex
48  $v3 = 'V3^ab[p_a, q_a, k_a]'.eq $hff
49  println $v3
```

```
▷ V3^ab[p_a, q_a, k_a] = (4*g-1)*p^{l}*q_{l}*g^{ab}-m**2...
```

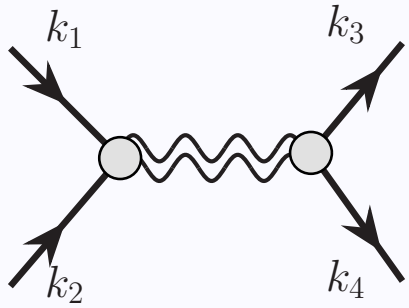
- The result reads:

$$V^{ab}(p,q) = \quad = -2g\left(q^a q^b + p^a p^b\right) + \\ + (1-2g)\left(q^a p^b + q^b p^a\right) + \left(2g(q^2+p^2) + (4g-1)q_l p^l - m^2\right) g^{ab}$$

- Other vertices ($h\,h\,h$, $h\,h\,\phi$, ...) can be found absolutely in the same manner

Deriving Feynman rules

- **Step 5:** Calculate processes
- We have all Feynman rules, let's calculate amplitude for some simple process

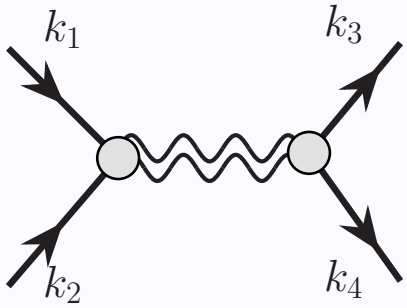



```
50  $scalars = setMandelstam([k1_a:'m',k2_a:'m',k3_a:'m',k4_a:'m'])
51  $amp = 'V3^ab[k1_a,k2_a]*P_abcd[-k1_a - k2_a]*V3^cd[-k3_a,k4_a]'.t
52  $amp <<= $hhPropagator & $v3
53  $amp <<= ExpandAndEliminate[$scalars] & 'd^n_n = 4'.t
54  $amp <<= Factor
55  println $amp
```

```
▷ s**(-1)*(32*m**4*b*t*k**2+80*c*m**2*s*g**2-8*m**2*b*u*...
```

Deriving Feynman rules

- **Step 5:** Calculate processes
- We have all Feynman rules, let's calculate amplitude for some simple process



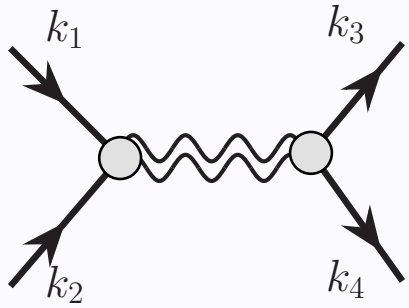

```
50  $scalars = setMandelstam([k1_a:'m',k2_a:'m',k3_a:'m',k4_a:'m'])
51  $amp = 'V3^ab[k1_a,k2_a]*P_abcd[-k1_a - k2_a]*V3^cd[-k3_a,k4_a]'.t
52  $amp <<= $hhPropagator & $v3
53  $amp <<= ExpandAndEliminate[$scalars] & 'd^n_n = 4'.t
54  $amp <<= Factor
55  println $amp
```

```
▷ s**(-1)*(32*m**4*b*t*k**2+80*c*m**2*s*g**2-8*m**2*b*u*...
```

- `setMandelstam` returns a set of rules $k_1^2 = m^2, (k_1 k_2) = s - 2m^2, \ldots$

Deriving Feynman rules

- **Step 5:** Calculate processes
- We have all Feynman rules, let's calculate amplitude for some simple process



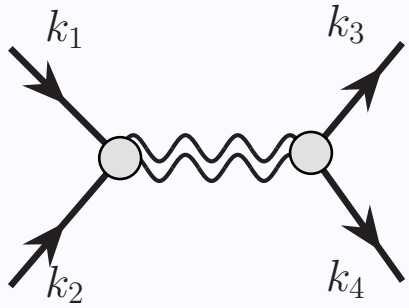

```
50  $scalars = setMandelstam([k1_a:'m',k2_a:'m',k3_a:'m',k4_a:'m'])
51  $amp = 'V3^ab[k1_a,k2_a]*P_abcd[-k1_a - k2_a]*V3^cd[-k3_a,k4_a]'.t
52  $amp <<= $hhPropagator & $v3
53  $amp <<= ExpandAndEliminate[$scalars] & 'd^n_n = 4'.t
54  $amp <<= Factor
55  println $amp
```

```
▷ s**(-1)*(32*m**4*b*t*k**2+80*c*m**2*s*g**2-8*m**2*b*u*...
```

- `setMandelstam` returns a set of rules $k_1^2 = m^2, (k_1 k_2) = s - 2m^2, \ldots$
- `ExpandAndEliminate[$scalars]` do expand and contractions with Minkowskii and substitute Mandelstam variables

Deriving Feynman rules

- **Step 5:** Calculate processes
- We have all Feynman rules, let's calculate amplitude for some simple process



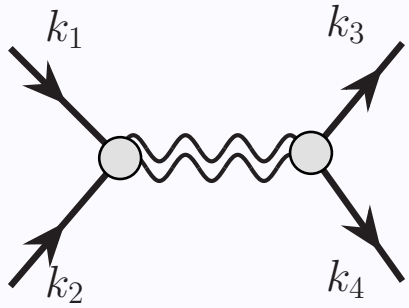

```
50  $scalars = setMandelstam([k1_a:'m',k2_a:'m',k3_a:'m',k4_a:'m'])
51  $amp = 'V3^ab[k1_a,k2_a]*P_abcd[-k1_a - k2_a]*V3^cd[-k3_a,k4_a]'.t
52  $amp <<= $hhPropagator & $v3
53  $amp <<= ExpandAndEliminate[$scalars] & 'd^n_n = 4'.t
54  $amp <<= Factor
55  println $amp
```

```
▷ s**(-1)*(32*m**4*b*t*k**2+80*c*m**2*s*g**2-8*m**2*b*u*...
```

- `setMandelstam` returns a set of rules $k_1^2 = m^2, (k_1 k_2) = s - 2m^2, \ldots$
- `ExpandAndEliminate[$scalars]` do expand and contractions with Minkowskii and substitute Mandelstam variables
- Again we control the dimension with just `d^n_n = 4`

Deriving Feynman rules

- **Step 5:** Calculate processes
- We have all Feynman rules, let's calculate amplitude for some simple process



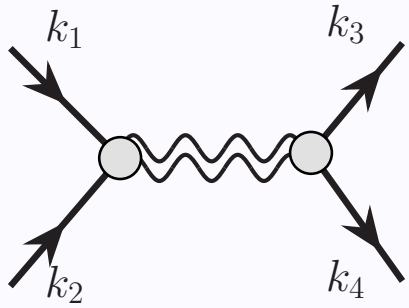

```
50  $scalars = setMandelstam([k1_a:'m',k2_a:'m',k3_a:'m',k4_a:'m'])
51  $amp = 'V3^ab[k1_a,k2_a]*P_abcd[-k1_a - k2_a]*V3^cd[-k3_a,k4_a]'.t
52  $amp <<= $hhPropagator & $v3
53  $amp <<= ExpandAndEliminate[$scalars] & 'd^n_n = 4'.t
54  $amp <<= Factor
55  println $amp
```

```
▷ s**(-1)*(32*m**4*b*t*k**2+80*c*m**2*s*g**2-8*m**2*b*u*...
```

- `setMandelstam` returns a set of rules $k_1^2 = m^2, (k_1 k_2) = s - 2m^2, \ldots$
- `ExpandAndEliminate[$scalars]` do expand and contractions with Minkowskii and substitute Mandelstam variables
- Again we control the dimension with just `d^n_n = 4`
- Finally, `Factor` will do polynomial factorization

Deriving Feynman rules

- **Step 5:** Calculate processes
- We have all Feynman rules, let's calculate amplitude for some simple process




```
50  $scalars = setMandelstam([k1_a:'m',k2_a:'m',k3_a:'m',k4_a:'m'])
51  $amp = 'V3^ab[k1_a,k2_a]*P_abcd[-k1_a - k2_a]*V3^cd[-k3_a,k4_a]'.t
52  $amp <<= $hhPropagator & $v3
53  $amp <<= ExpandAndEliminate[$scalars] & 'd^n_n = 4'.t
54  $amp <<= Factor
55  println $amp
```

```
▷ s**(-1)*(32*m**4*b*t*k**2+80*c*m**2*s*g**2-8*m**2*b*u*...
```

$$
\begin{aligned}
&-\frac{1}{s\left(a\kappa^2 s+c\right)\left(c-2\kappa^2 s(a-3b)\right)}\Big(\kappa^2\Big(\kappa^2 s\Big(a\Big(4(4\gamma-1)m^4\\
&\qquad+8m^2\left(10\gamma^2 s-6\gamma s+s-8\gamma t+2t\right)+(4(2-5\gamma)\gamma-1)s^2\\
&+4(4\gamma-1)st+4(4\gamma-1)t^2\Big)-2b(4\gamma-1)\left(6t\left(s-4m^2\right)+\left(s-4m^2\right)^2+6t^2\right)\Big)\\
&\qquad+2c\left((6-24\gamma)m^4+8\gamma(5\gamma-1)m^2 s+(4\gamma-1)t\left(4m^2-s\right)\right.\\
&\qquad\left.+2(1-5\gamma)\gamma s^2+(1-4\gamma)t^2\right)\Big)\Big)
\end{aligned}
$$

Deriving Feynman rules

- We have derived Feynman rules for a complicated theory with gravity with just about 50 lines of code
- No problem to do the same with spinors, γ-matrices etc. – all these imlemented in `Redberry`
- No problem to do the same automatically for almost any Lagrangian
- `Redberry` also contains tools for automatic comutation of one-loop counterterms from arbitrary Lagrangians in curved spacetime

  (based on [Stepanyantz et al, Nucl.Phys. B485 (1997) 517-544])

Deriving Feynman rules

- We have derived Feynman rules for a complicated theory with gravity with just about 50 lines of code
- No problem to do the same with spinors, γ-matrices etc. – all these imlemented in `Redberry`
- No problem to do the same automatically for almost any Lagrangian
- `Redberry` also contains tools for automatic comutation of one-loop counterterms from arbitrary Lagrangians in curved spacetime

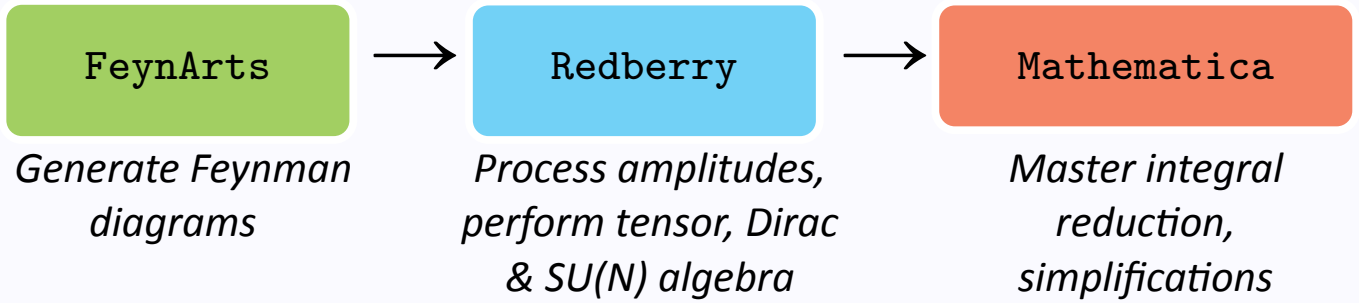
  (based on [Stepanyantz et al, Nucl.Phys. B485 (1997) 517-544])

?

What about calculating complicated processes, e.g. NLO?

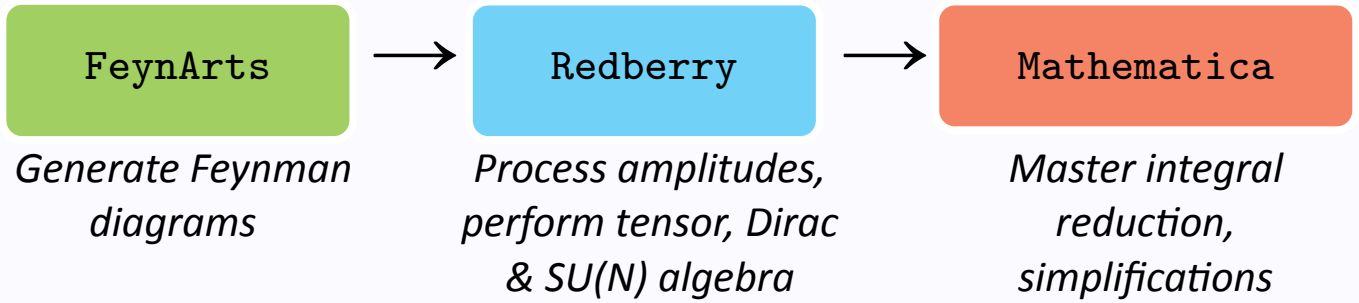
NLO computations with FeynArts + Redberry + FIRE

- Let's consider the process $e^+ e^- \to \gamma^* \to J/\psi + \eta_c$ in NLO QCD with the following pipeline:

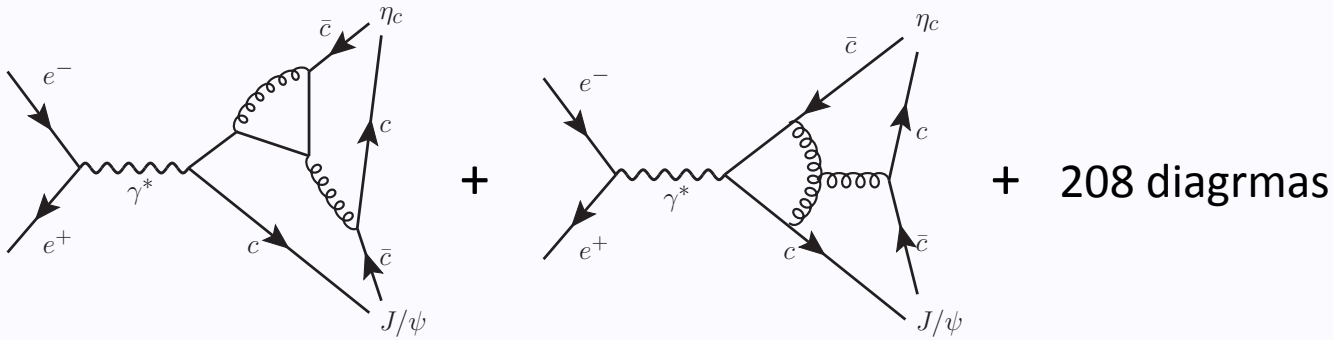


NLO computations with FeynArts + Redberry + FIRE

- Let's consider the process $e^+ e^- \to \gamma^* \to J/\psi + \eta_c$ in NLO QCD with the following pipeline:



- **Step 1**: Generate diagrams using FeynArts:



NLO computations with FeynArts + Redberry + FIRE

- **Step 2**: Convert from FeynArts to Redberry
- This is what we have from FeynArts:

```
-(FermionChain[NonCommutative[DiracSpinor[FourMomentum[Outgoing,
    3], MC]], (-I)*GS*NonCommutative[DiracMatrix[Index[Lorentz,
    2]], ChiralityProjector[-1]]*SUNT[Index[Gluon, 6], Index[
    Colour, 3], Index[Colour, 2]]...
```

- This is what we need in Redberry:

```
ubar[pEta1_a]*(-I)*gS*G_{b}*T_{F}*v[pPsi2_a]*ubar[pPsi1_a]*
((-2*I)/3)*e*G_{a}*(MC-DiracSlash[pEta1_{a}]-DiracSlash[pEta2_{a
    }]...
```

NLO computations with FeynArts + Redberry + FIRE

- **Step 2**: Convert from FeynArts to Redberry
- This is what we have from FeynArts:

```
-(FermionChain[NonCommutative[DiracSpinor[FourMomentum[Outgoing,
    3], MC]], (-I)*GS*NonCommutative[DiracMatrix[Index[Lorentz,
    2]], ChiralityProjector[-1]]*SUNT[Index[Gluon, 6], Index[
    Colour, 3], Index[Colour, 2]]...
```

- This is what we need in Redberry:

```
ubar[pEta1_a]*(-I)*gS*G_{b}*T_{F}*v[pPsi2_a]*ubar[pPsi1_a]*
((-2*I)/3)*e*G_{a}*(MC-DiracSlash[pEta1_{a}]-DiracSlash[pEta2_{a
    }]...
```

  - `ubar[p_a]` and `v[p_a]` are spinors

NLO computations with FeynArts + Redberry + FIRE

- **Step 2**: Convert from FeynArts to Redberry
- This is what we have from FeynArts:

```
-(FermionChain[NonCommutative[DiracSpinor[FourMomentum[Outgoing,
    3], MC]], (-I)*GS*NonCommutative[DiracMatrix[Index[Lorentz,
    2]], ChiralityProjector[-1]]*SUNT[Index[Gluon, 6], Index[
    Colour, 3], Index[Colour, 2]]...
```

- This is what we need in Redberry:

```
ubar[pEta1_a]*(-I)*gS*G_{b}*T_{F}*v[pPsi2_a]*ubar[pPsi1_a]*
((-2*I)/3)*e*G_{a}*(MC-DiracSlash[pEta1_{a}]-DiracSlash[pEta2_{a
    }]...
```

  - `ubar[p_a]` and `v[p_a]` are spinors
  - `G_a` and `T_A` are Dirac and SU(N) matrices

NLO computations with FeynArts + Redberry + FIRE

- **Step 2**: Convert from FeynArts to Redberry
- This is what we have from FeynArts:

```
-(FermionChain[NonCommutative[DiracSpinor[FourMomentum[Outgoing,
    3], MC]], (-I)*GS*NonCommutative[DiracMatrix[Index[Lorentz,
    2]], ChiralityProjector[-1]]*SUNT[Index[Gluon, 6], Index[
    Colour, 3], Index[Colour, 2]]...
```

- This is what we need in Redberry:

```
ubar[pEta1_a]*(-I)*gS*G_{b}*T_{F}*v[pPsi2_a]*ubar[pPsi1_a]*
((-2*I)/3)*e*G_{a}*(MC-DiracSlash[pEta1_{a}]-DiracSlash[pEta2_{a
    }]...
```

  - `ubar[p_a]` and `v[p_a]` are spinors
  - `G_a` and `T_A` are Dirac and SU(N) matrices
  - In fact, Redberry takes amplitudes in the form very close to what we have with paper and pencil

NLO computations with FeynArts + Redberry + FIRE

- **Step 3**: Basic setup in Redberry

```
1  setAntiSymmetric 'e_abcd', 'f_ABC'
2  setSymmetric 'd_ABC'

4  defineMatrices 'T_A', Matrix2.matrix,//unitary matrices
5      'G_a', 'G5', 'DiracSlash[p_a]', 'D[p_m, m]', Matrix1.matrix,
6      'v[p_a]', Matrix1.vector, Matrix2.vector, //quark
7      'ubar[p_a]', Matrix1.covector, Matrix2.covector, //antiquark
8      'ev[p_a]', 'eu[p_a]', Matrix1.vector, //electron
9      'evbar[p_a]', 'eubar[p_a]', Matrix1.covector, //positron

11 'DiracSlash[p_a] := G^a*p_a'.t
12 'Pair[p_a, q_a] := p_a*q^a'.t
```

NLO computations with FeynArts + Redberry + FIRE

- **Step 3**: Basic setup in Redberry

```
1  setAntiSymmetric 'e_abcd', 'f_ABC'
2  setSymmetric 'd_ABC'

4  defineMatrices 'T_A', Matrix2.matrix,//unitary matrices
5      'G_a', 'G5', 'DiracSlash[p_a]', 'D[p_m, m]', Matrix1.matrix,
6      'v[p_a]', Matrix1.vector, Matrix2.vector, //quark
7      'ubar[p_a]', Matrix1.covector, Matrix2.covector, //antiquark
8      'ev[p_a]', 'eu[p_a]', Matrix1.vector, //electron
9      'evbar[p_a]', 'eubar[p_a]', Matrix1.covector, //positron

11 'DiracSlash[p_a] := G^a*p_a'.t
12 'Pair[p_a, q_a] := p_a*q^a'.t
```

  - `T_A` are SU(N); `G_a`, `G5`, `...` are Dirac matrices

NLO computations with FeynArts + Redberry + FIRE

- **Step 3**: Basic setup in Redberry

```
1  setAntiSymmetric 'e_abcd', 'f_ABC'
2  setSymmetric 'd_ABC'

4  defineMatrices 'T_A', Matrix2.matrix,//unitary matrices
5      'G_a', 'G5', 'DiracSlash[p_a]', 'D[p_m, m]', Matrix1.matrix,
6      'v[p_a]', Matrix1.vector, Matrix2.vector, //quark
7      'ubar[p_a]', Matrix1.covector, Matrix2.covector, //antiquark
8      'ev[p_a]', 'eu[p_a]', Matrix1.vector, //electron
9      'evbar[p_a]', 'eubar[p_a]', Matrix1.covector, //positron

11 'DiracSlash[p_a] := G^a*p_a'.t
12 'Pair[p_a, q_a] := p_a*q^a'.t
```

  - `T_A` are SU(N); `G_a, G5, ...` are Dirac matrices

NLO computations with FeynArts + Redberry + FIRE

- **Step 3**: Basic setup in Redberry

```
1  setAntiSymmetric 'e_abcd', 'f_ABC'
2  setSymmetric 'd_ABC'

4  defineMatrices 'T_A', Matrix2.matrix,//unitary matrices
5      'G_a', 'G5', 'DiracSlash[p_a]', 'D[p_m, m]', Matrix1.matrix,
6      'v[p_a]', Matrix1.vector, Matrix2.vector, //quark
7      'ubar[p_a]', Matrix1.covector, Matrix2.covector, //antiquark
8      'ev[p_a]', 'eu[p_a]', Matrix1.vector, //electron
9      'evbar[p_a]', 'eubar[p_a]', Matrix1.covector, //positron

11 'DiracSlash[p_a] := G^a*p_a'.t
12 'Pair[p_a, q_a] := p_a*q^a'.t
```

- `T_A` are SU(N); `G_a`, `G5`, `...` are Dirac matrices
- `v[p_a]` is a quark spinor; `ubar[p_a]` its conjugation

NLO computations with FeynArts + Redberry + FIRE

- **Step 3**: Basic setup in Redberry

```
1  setAntiSymmetric 'e_abcd', 'f_ABC'
2  setSymmetric 'd_ABC'

4  defineMatrices 'T_A', Matrix2.matrix,//unitary matrices
5      'G_a', 'G5', 'DiracSlash[p_a]', 'D[p_m, m]', Matrix1.matrix,
6      'v[p_a]', Matrix1.vector, Matrix2.vector, //quark
7      'ubar[p_a]', Matrix1.covector, Matrix2.covector, //antiquark
8      'ev[p_a]', 'eu[p_a]', Matrix1.vector, //electron
9      'evbar[p_a]', 'eubar[p_a]', Matrix1.covector, //positron

11 'DiracSlash[p_a] := G^a*p_a'.t
12 'Pair[p_a, q_a] := p_a*q^a'.t
```

- `T_A` are SU(N); `G_a`, `G5`, `...` are Dirac matrices
- `v[p_a]` is a quark spinor; `ubar[p_a]` its conjugation

NLO computations with FeynArts + Redberry + FIRE

- **Step 3**: Basic setup in Redberry

```
1  setAntiSymmetric 'e_abcd', 'f_ABC'
2  setSymmetric 'd_ABC'

4  defineMatrices 'T_A', Matrix2.matrix,//unitary matrices
5      'G_a', 'G5', 'DiracSlash[p_a]', 'D[p_m, m]', Matrix1.matrix,
6      'v[p_a]', Matrix1.vector, Matrix2.vector, //quark
7      'ubar[p_a]', Matrix1.covector, Matrix2.covector, //antiquark
8      'ev[p_a]', 'eu[p_a]', Matrix1.vector, //electron
9      'evbar[p_a]', 'eubar[p_a]', Matrix1.covector, //positron

11 'DiracSlash[p_a] := G^a*p_a'.t
12 'Pair[p_a, q_a] := p_a*q^a'.t
```

- `T_A` are SU(N); `G_a`, `G5`, `...` are Dirac matrices
- `v[p_a]` is a quark spinor; `ubar[p_a]` its conjugation
- Set up once the `DiracSlash` and `Pair` coming from FeynArts

NLO computations with FeynArts + Redberry + FIRE

- **Step 3**: Basic setup in Redberry

```
1  setAntiSymmetric 'e_abcd', 'f_ABC'
2  setSymmetric 'd_ABC'

4  defineMatrices 'T_A', Matrix2.matrix,//unitary matrices
5      'G_a', 'G5', 'DiracSlash[p_a]', 'D[p_m, m]', Matrix1.matrix,
6      'v[p_a]', Matrix1.vector, Matrix2.vector, //quark
7      'ubar[p_a]', Matrix1.covector, Matrix2.covector, //antiquark
8      'ev[p_a]', 'eu[p_a]', Matrix1.vector, //electron
9      'evbar[p_a]', 'eubar[p_a]', Matrix1.covector, //positron

11 'DiracSlash[p_a] := G^a*p_a'.t
12 'Pair[p_a, q_a] := p_a*q^a'.t
```

  - `T_A` are SU(N); `G_a`, `G5`, `...` are Dirac matrices
  - `v[p_a]` is a quark spinor; `ubar[p_a]` its conjugation
  - Set up once the `DiracSlash` and `Pair` coming from FeynArts

- Having all this, Redberry will internally handle

```
ubar[p_a]*T_A*G_a*v[k_a] ->
      -> ubar_{a'A'}[p_a]*T^{A'}_{B'A}*G^{a'}_{b'a}*v^{b'B'}[k_a]
```

NLO computations with FeynArts + Redberry + FIRE

- **Step 3**: Basic setup in Redberry

```
13  $proj = 'v[pPsi2_a]*ubar[pPsi1_a] =
14          (G^a*pPsi_a/2-MC)*G_m*epsPsi^m*(G^b*pPsi_b/2+MC)'.t & ...

16  $scalars  = setMandelstam([k1_a: 0, k2_a: 0, pPsi_a: '2*MC'...
17  $qScalars = 'q_a*pPsi^a = qp'.t & 'q_a*pEta^a = qe'.t & ...

19  $PaVe  = PassarinoVeltman([1, 2], 'q_a', ['pPsi_a', 'pEta_a'])
20  $dSimplify = DiracSimplify[[Dimension: 'd', TraceOfOne: 4]]
21  $dTrace = DiracTrace[[Dimension: 'd', TraceOfOne: 4]]
```

NLO computations with FeynArts + Redberry + FIRE

- **Step 3**: Basic setup in Redberry

```
13  $proj = 'v[pPsi2_a]*ubar[pPsi1_a] =
14          (G^a*pPsi_a/2-MC)*G_m*epsPsi^m*(G^b*pPsi_b/2+MC)'.t & ...

16  $scalars  = setMandelstam([k1_a: 0, k2_a: 0, pPsi_a: '2*MC'...
17  $qScalars = 'q_a*pPsi^a = qp'.t & 'q_a*pEta^a = qe'.t & ...

19  $PaVe  = PassarinoVeltman([1, 2], 'q_a', ['pPsi_a', 'pEta_a'])
20  $dSimplify = DiracSimplify[[Dimension: 'd', TraceOfOne: 4]]
21  $dTrace = DiracTrace[[Dimension: 'd', TraceOfOne: 4]]
```

  - `$proj` are J/ψ & η_c projectors: $v(p_1)\bar{u}(p_2) = \left(\not{P}/2 - M\right)\not{\epsilon}\left(\not{P}/2 + M\right)$

NLO computations with FeynArts + Redberry + FIRE

- **Step 3**: Basic setup in Redberry

```
13  $proj = 'v[pPsi2_a]*ubar[pPsi1_a] =
14          (G^a*pPsi_a/2-MC)*G_m*epsPsi^m*(G^b*pPsi_b/2+MC)'.t & ...

16  $scalars  = setMandelstam([k1_a: 0, k2_a: 0, pPsi_a: '2*MC'...
17  $qScalars = 'q_a*pPsi^a = qp'.t & 'q_a*pEta^a = qe'.t & ...

19  $PaVe  = PassarinoVeltman([1, 2], 'q_a', ['pPsi_a', 'pEta_a'])
20  $dSimplify = DiracSimplify[[Dimension: 'd', TraceOfOne: 4]]
21  $dTrace = DiracTrace[[Dimension: 'd', TraceOfOne: 4]]
```

- `$proj` are J/ψ & η_c projectors: $v(p_1)\bar{u}(p_2) = \left(\not{P}/2 - M\right)\not{\epsilon}\left(\not{P}/2 + M\right)$
- `$qScalars` replaces contractions with loop momenta

NLO computations with FeynArts + Redberry + FIRE

- **Step 3**: Basic setup in Redberry

```
13  $proj = 'v[pPsi2_a]*ubar[pPsi1_a] =
14          (G^a*pPsi_a/2-MC)*G_m*epsPsi^m*(G^b*pPsi_b/2+MC)'.t & ...

16  $scalars  = setMandelstam([k1_a: 0, k2_a: 0, pPsi_a: '2*MC'...
17  $qScalars = 'q_a*pPsi^a = qp'.t & 'q_a*pEta^a = qe'.t & ...

19  $PaVe  = PassarinoVeltman([1, 2], 'q_a', ['pPsi_a', 'pEta_a'])
20  $dSimplify = DiracSimplify[[Dimension: 'd', TraceOfOne: 4]]
21  $dTrace = DiracTrace[[Dimension: 'd', TraceOfOne: 4]]
```

- `$proj` are J/ψ & η_c projectors: $v(p_1)\bar{u}(p_2) = \left(\not{P}/2 - M\right)\not{\epsilon}\left(\not{P}/2 + M\right)$
- `$qScalars` replaces contractions with loop momenta
- `PassarinoVeltman` returns a list of substitutions:

```
q_a*q_b = (...)*pPsi_a*pEta_b + (...)*g_ab + ...
```

NLO computations with FeynArts + Redberry + FIRE

- **Step 3**: Basic setup in Redberry

```
13  $proj = 'v[pPsi2_a]*ubar[pPsi1_a] =
14          (G^a*pPsi_a/2-MC)*G_m*epsPsi^m*(G^b*pPsi_b/2+MC)'.t & ...

16  $scalars  = setMandelstam([k1_a: 0, k2_a: 0, pPsi_a: '2*MC'...
17  $qScalars = 'q_a*pPsi^a = qp'.t & 'q_a*pEta^a = qe'.t & ...

19  $PaVe  = PassarinoVeltman([1, 2], 'q_a', ['pPsi_a', 'pEta_a'])
20  $dSimplify = DiracSimplify[[Dimension: 'd', TraceOfOne: 4]]
21  $dTrace = DiracTrace[[Dimension: 'd', TraceOfOne: 4]]
```

- `$proj` are J/ψ & η_c projectors: $v(p_1)\bar{u}(p_2) = \left(\not{P}/2 - M\right)\not{\epsilon}\left(\not{P}/2 + M\right)$
- `$qScalars` replaces contractions with loop momenta
- `PassarinoVeltman` returns a list of substitutions:

```
q_a*q_b = (...)*pPsi_a*pEta_b + (...)*g_ab + ...
```

- `DiracSimplify` and `DiracTrace` will do the corresponding algebra in `d` dimensions (`d^a_a = d`) and following `Tr[1] = 4`

```
println $dTrace >> 'Tr[G_a*G_b*G^a*G_c]'.t
```

▷ `-4*(d-2)*g_{bc}`

NLO computations with FeynArts + Redberry + FIRE

- **Step 3**: Basic setup in Redberry

```
13  $proj = 'v[pPsi2_a]*ubar[pPsi1_a] =
14          (G^a*pPsi_a/2-MC)*G_m*epsPsi^m*(G^b*pPsi_b/2+MC)'.t & ...

16  $scalars  = setMandelstam([k1_a: 0, k2_a: 0, pPsi_a: '2*MC'...
17  $qScalars = 'q_a*pPsi^a = qp'.t & 'q_a*pEta^a = qe'.t & ...

19  $PaVe  = PassarinoVeltman([1, 2], 'q_a', ['pPsi_a', 'pEta_a'])
20  $dSimplify = DiracSimplify[[Dimension: 'd', TraceOfOne: 4]]
21  $dTrace = DiracTrace[[Dimension: 'd', TraceOfOne: 4]]
```

- `$proj` are J/ψ & η_c projectors: $v(p_1)\bar{u}(p_2) = \left(\not{P}/2 - M\right)\not{\epsilon}\left(\not{P}/2 + M\right)$
- `$qScalars` replaces contractions with loop momenta
- `PassarinoVeltman` returns a list of substitutions:

```
q_a*q_b = (...)*pPsi_a*pEta_b + (...)*g_ab + ...
```

- `DiracSimplify` and `DiracTrace` will do the corresponding algebra in `d` dimensions (`d^a_a = d`) and following `Tr[1] = 4`

```
println $dTrace >> 'Tr[G_a*G_b*G^a*G_c]'.t
```

▷ `-4*(d-2)*g_{bc}`

NLO computations with FeynArts + Redberry + FIRE

- **Step 4**: Main loop for NLO

```
22  $amps = __CallFeynArts__().t //import diagrams from FeynArts
23  $answNLO = 0.t
24  for($amp in $amps){
25      $amp <<= $proj
26      $num = Numerator >> $amp
27      $num <<= UnitaryTrace & UnitarySimplify
28      $num <<= $qScalars & $PaVe & ExpandAndEliminate & $qScalars
29      $num <<= $dSimplify & $dTrace & LeviCivitaSimplify
30      $num <<= 'd^n_n = d'.t
31      ...
32      $answNLO += Factor >> ($num / $den)
33  }
```

- So, for each diagram we

NLO computations with FeynArts + Redberry + FIRE

- **Step 4**: Main loop for NLO

```
22  $amps = __CallFeynArts__().t //import diagrams from FeynArts
23  $answNLO = 0.t
24  for($amp in $amps){
25      $amp <<= $proj
26      $num = Numerator >> $amp
27      $num <<= UnitaryTrace & UnitarySimplify
28      $num <<= $qScalars & $PaVe & ExpandAndEliminate & $qScalars
29      $num <<= $dSimplify & $dTrace & LeviCivitaSimplify
30      $num <<= 'd^n_n = d'.t
31      ...
32      $answNLO += Factor >> ($num / $den)
33  }
```

- So, for each diagram we
  - project quarks onto final quarkonia with `$proj`

NLO computations with FeynArts + Redberry + FIRE

- **Step 4**: Main loop for NLO

```
22  $amps = __CallFeynArts__().t //import diagrams from FeynArts
23  $answNLO = 0.t
24  for($amp in $amps){
25      $amp <<= $proj
26      $num = Numerator >> $amp
27      $num <<= UnitaryTrace & UnitarySimplify
28      $num <<= $qScalars & $PaVe & ExpandAndEliminate & $qScalars
29      $num <<= $dSimplify & $dTrace & LeviCivitaSimplify
30      $num <<= 'd^n_n = d'.t
31      ...
32      $answNLO += Factor >> ($num / $den)
33  }
```

- So, for each diagram we
  - project quarks onto final quarkonia with `$proj`
  - Perform SU(N) algebra

NLO computations with FeynArts + Redberry + FIRE

- **Step 4**: Main loop for NLO

```
22  $amps = __CallFeynArts__().t //import diagrams from FeynArts
23  $answNLO = 0.t
24  for($amp in $amps){
25      $amp <<= $proj
26      $num = Numerator >> $amp
27      $num <<= UnitaryTrace & UnitarySimplify
28      $num <<= $qScalars & $PaVe & ExpandAndEliminate & $qScalars
29      $num <<= $dSimplify & $dTrace & LeviCivitaSimplify
30      $num <<= 'd^n_n = d'.t
31      ...
32      $answNLO += Factor >> ($num / $den)
33  }
```

- So, for each diagram we
  - project quarks onto final quarkonia with `$proj`
  - Perform SU(N) algebra
  - Replace contractions of loop momenta with gammas with `$PaVe`

NLO computations with FeynArts + Redberry + FIRE

- **Step 4**: Main loop for NLO

```
22  $amps = __CallFeynArts__().t //import diagrams from FeynArts
23  $answNLO = 0.t
24  for($amp in $amps){
25      $amp <<= $proj
26      $num = Numerator >> $amp
27      $num <<= UnitaryTrace & UnitarySimplify
28      $num <<= $qScalars & $PaVe & ExpandAndEliminate & $qScalars
29      $num <<= $dSimplify & $dTrace & LeviCivitaSimplify
30      $num <<= 'd^n_n = d'.t
31      ...
32      $answNLO += Factor >> ($num / $den)
33  }
```

- So, for each diagram we
  - project quarks onto final quarkonia with `$proj`
  - Perform SU(N) algebra
  - Replace contractions of loop momenta with gammas with `$PaVe`
  - Perform `d`-dimensional Dirac algebra; again the rest dimension controled by just `d^n_n = d`

NLO computations with FeynArts + Redberry + FIRE

- **Step 4**: Main loop for NLO

```
22  $amps = __CallFeynArts__().t //import diagrams from FeynArts
23  $answNLO = 0.t
24  for($amp in $amps){
25      $amp <<= $proj
26      $num = Numerator >> $amp
27      $num <<= UnitaryTrace & UnitarySimplify
28      $num <<= $qScalars & $PaVe & ExpandAndEliminate & $qScalars
29      $num <<= $dSimplify & $dTrace & LeviCivitaSimplify
30      $num <<= 'd^n_n = d'.t
31      ...
32      $answNLO += Factor >> ($num / $den)
33  }
```

- So, for each diagram we
  - project quarks onto final quarkonia with `$proj`
  - Perform SU(N) algebra
  - Replace contractions of loop momenta with gammas with `$PaVe`
  - Perform `d`-dimensional Dirac algebra; again the rest dimension controled by just `d^n_n = d`

NLO computations with FeynArts + Redberry + FIRE

- **Step 5**: Reduction to master integrals
- For now we have the `answNLO` in the form

```
(...)*e_{abcd}*pPsi^b*pEta^c*epsPsi^d + (scalar) * (tensor) + ...
```

NLO computations with FeynArts + Redberry + FIRE

- **Step 5**: Reduction to master integrals
- For now we have the `answNLO` in the form

```
(...)*e_{abcd}*pPsi^b*pEta^c*epsPsi^d + (scalar) * (tensor) + ...
```

  - all similar `tensor` terms are collected automatically

NLO computations with FeynArts + Redberry + FIRE

- **Step 5**: Reduction to master integrals
- For now we have the `answNLO` in the form

```
(...)*e_{abcd}*pPsi^b*pEta^c*epsPsi^d + (scalar) * (tensor) + ...
```

  - all similar tensor terms are collected automatically
  - each scalar prefactor is a sum of loop integrals that should be reduced

NLO computations with FeynArts + Redberry + FIRE

- **Step 5**: Reduction to master integrals
- For now we have the `answNLO` in the form

```
(...)*e_{abcd}*pPsi^b*pEta^c*epsPsi^d + (scalar) * (tensor) ...
```

  - all similar tensor terms are collected automatically
  - each scalar prefactor is a sum of loop integrals that should be reduced
- Do the reduction with external routine

```
34  for($amp in $answNLO){
35      $int = $amp.indexlessSubProduct.toString(WolframMathematica)
36      __ReduceWithFIRE__($int)
37  }
```

  - I've used `Mathematica FIRE` *[V. A. Smirnov, JHEP0810:107,2008]* for the reduction to masters (with some additional code for e.g. partial fractioning)
  - For this simple process, I've substituted values for masters by hand

NLO computations with FeynArts + Redberry + FIRE

- **Step 6**: Final step – square matrix element

```
38  $answTree = ...
39  $conj = InvertIndices & Conjugate & ...
40  $M2  = ($conj >> $answNLO)  * $answTree
41  $M2 += ($conj >> $answTree) * $answNLO
42  $M2 <<= ExpandAndEliminate
43  $M2 <<= 'epsPsi_a*epsPsi_b = -g_ab + pPsi_a*pPsi_b/(4*MC**2)'.t
44  ...
45  $M2 <<= ExpandAndEliminate & $scalars & ...
```

NLO computations with FeynArts + Redberry + FIRE

- **Step 6**: Final step – square matrix element

```
38  $answTree = ...
39  $conj = InvertIndices & Conjugate & ...
40  $M2  = ($conj >> $answNLO)  * $answTree
41  $M2 += ($conj >> $answTree) * $answNLO
42  $M2 <<= ExpandAndEliminate
43  $M2 <<= 'epsPsi_a*epsPsi_b = -g_ab + pPsi_a*pPsi_b/(4*MC**2)'.t
44  ...
45  $M2 <<= ExpandAndEliminate & $scalars & ...
```

  - Tree level can be calculated in the absolutely same way

NLO computations with FeynArts + Redberry + FIRE

- **Step 6**: Final step – square matrix element

```
38  $answTree = ...
39  $conj = InvertIndices & Conjugate & ...
40  $M2  = ($conj >> $answNLO)  * $answTree
41  $M2 += ($conj >> $answTree) * $answNLO
42  $M2 <<= ExpandAndEliminate
43  $M2 <<= 'epsPsi_a*epsPsi_b = -g_ab + pPsi_a*pPsi_b/(4*MC**2)'.t
44  ...
45  $M2 <<= ExpandAndEliminate & $scalars & ...
```

  - Tree level can be calculated in the absolutely same way
  - Conjugate (invert indices of external particles, revert matrices, replace spinors etc.)

NLO computations with FeynArts + Redberry + FIRE

- **Step 6**: Final step – square matrix element

```
38  $answTree = ...
39  $conj = InvertIndices & Conjugate & ...
40  $M2  = ($conj >> $answNLO)  * $answTree
41  $M2 += ($conj >> $answTree) * $answNLO
42  $M2 <<= ExpandAndEliminate
43  $M2 <<= 'epsPsi_a*epsPsi_b = -g_ab + pPsi_a*pPsi_b/(4*MC**2)'.t
44  ...
45  $M2 <<= ExpandAndEliminate & $scalars & ...
```

- Tree level can be calculated in the absolutely same way
- Conjugate (invert indices of external particles, revert matrices, replace spinors etc.)
- Substitute polarization sums, e.g. for J/ψ:

$$\sum \epsilon_\mu \epsilon_\nu = -g_{\mu\nu} + P_\mu P_\nu / (4M_c)^2$$

NLO computations with FeynArts + Redberry + FIRE

- **Step 6**: Final step – square matrix element

```
38  $answTree = ...
39  $conj = InvertIndices & Conjugate & ...
40  $M2  = ($conj >> $answNLO)  * $answTree
41  $M2 += ($conj >> $answTree) * $answNLO
42  $M2 <<= ExpandAndEliminate
43  $M2 <<= 'epsPsi_a*epsPsi_b = -g_ab + pPsi_a*pPsi_b/(4*MC**2)'.t
44  ...
45  $M2 <<= ExpandAndEliminate & $scalars & ...
```

- Tree level can be calculated in the absolutely same way
- Conjugate (invert indices of external particles, revert matrices, replace spinors etc.)
- Substitute polarization sums, e.g. for J/ψ:

$$\sum \epsilon_\mu \epsilon_\nu = -g_{\mu\nu} + P_\mu P_\nu / (4M_c)^2$$

- Perform final simplifications

NLO computations with FeynArts + Redberry + FIRE

- **Step 6**: Final step – square matrix element

```
38  $answTree = ...
39  $conj = InvertIndices & Conjugate & ...
40  $M2  = ($conj >> $answNLO)  * $answTree
41  $M2 += ($conj >> $answTree) * $answNLO
42  $M2 <<= ExpandAndEliminate
43  $M2 <<= 'epsPsi_a*epsPsi_b = -g_ab + pPsi_a*pPsi_b/(4*MC**2)'.t
44  ...
45  $M2 <<= ExpandAndEliminate & $scalars & ...
```

  - Tree level can be calculated in the absolutely same way
  - Conjugate (invert indices of external particles, revert matrices, replace spinors etc.)
  - Substitute polarization sums, e.g. for J/ψ:

$$\sum \epsilon_\mu \epsilon_\nu = -g_{\mu\nu} + P_\mu P_\nu / (4M_c)^2$$

  - Perform final simplifications

- Export scalar squared and reduced `M2` to e.g. `Mathematica` or other external program

NLO computations with FeynArts + Redberry + FIRE

✓ All in all about 50 lines of code for the $e^+ e^- \to J/\psi + \eta_c$ in NLO within less than 10 minutes of running time

✓ No problem to consider more sophisticated processes: more external particles, spinors in the final state, tensor and higher spin particles

Conclusions

Done:

- ✓ The natural and universal way in which `Redberry` handles all indexed objects provides powerful CA tools for high-level HEP needs
- ✓ Internal engine based on graph-theoretical approach shows a really good performance
- ✓ The performance of algebra and simplifications with tensors is unmatched

Nearest plans:

- More tools for NLO (another schemes for γ_5, universal helicity method etc.)
- More tools for geneal relativity
- More tools for supersymmetry (better anticommuting tensors etc.)
- Some performance moments can be further improved

`http://redberry.cc`

Thank you for your attention!

Backup slides

Issues with γ_5

- For the γ_5 in the dimensional regularization I use Chiholm-Kahane identity in d-dimensions:

$$\gamma_\alpha\gamma_\beta\gamma_\gamma = g_{\alpha\beta}\gamma_\gamma - g_{\alpha\gamma}\gamma_\beta + g_{\beta\gamma}\gamma_\alpha - i\epsilon_{\alpha\beta\gamma\delta}\gamma_5\gamma^\delta$$

  and thus

$$\gamma_5\gamma^\delta = \frac{-i\epsilon^{\alpha\beta\gamma\delta}\gamma_\alpha\gamma_\beta\gamma_\gamma}{(D-3)(D-2)(D-1)}$$

  what is called Larin-Gorishny-Akyeampong-Delburgo scheme

- At the moment this is the only possible scheme in `Redberry`
- In `Redberry`:

```
1  setAntiSymmetric 'e_abcd'
2  defineMatrices 'G_a', 'G5', Matrix1.matrix

4  dTrace = DiracTrace[[Dimension: 'd', TraceOfOne: 4]]
5  expr =  'Tr[G^e*G^f*G^g*G_a*G_b*G_c*G_d*G_e*G_f*G_g*G5]'.t
6  println dTrace >> expr
```

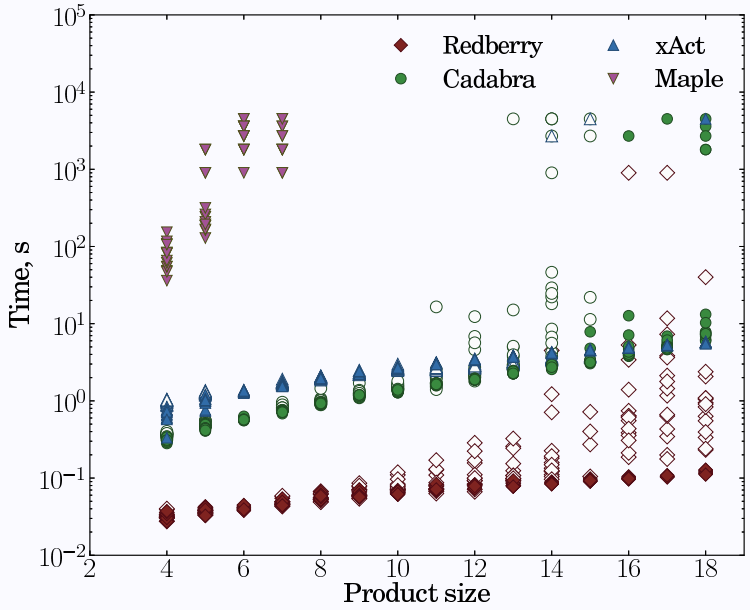
  ▷ `-96*I*(d**2-14*d+36)*e_{bdca}/(d-3)/(d-2)/(d-1)`

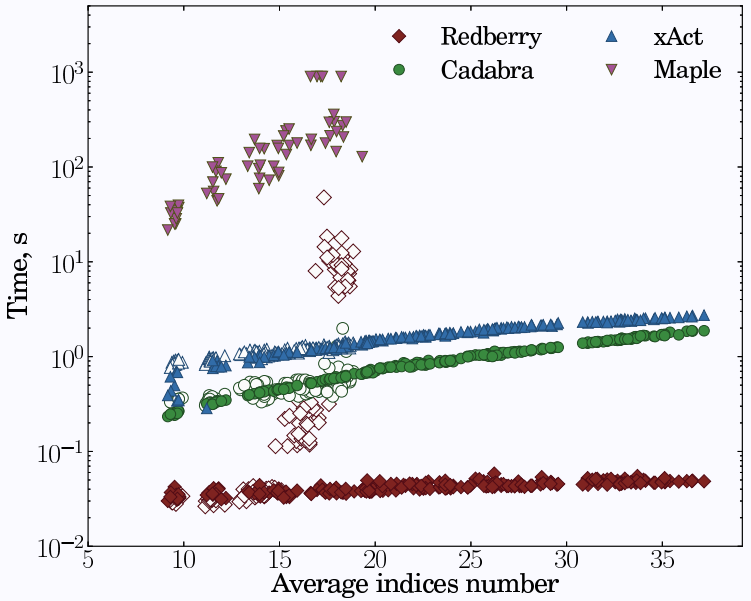
Interacting with Mathematica

- Mathematica provides a really good Java interface
- It can be used to do simplifications on scalar parts of expressions

```
1  mKernel = MathLinkFactory.createKernelLink(...)
2  mKernel.discardAnswer()
3  mFunc = { func, expr ->
4      mKernel.evaluateToInputForm(
5          func + '[' + expr.toString(WolframMathematica) + ']', 0).
6           replace('^', '**').t
7  }
8  mFactor   = {expr -> mFunc('Factor', expr) }
9  $factor   = Factor[[FactorizationEngine: mFactor]]
10 println $factor >> '(a**2-2*a*b+b**2)*g_mn + (a**2-a*c)*t_mn'.t

   ▷ (a-b)**2*g_mn + a*(a-c)*t_mn

13 mSimplify = {expr -> mFunc('Simplify', expr) }
14 $simplify = Factor[[FactorizationEngine: mSimplify]]
15 println $simplify >> '(a**2-2*a*b+b**2)*g_mn + (a**2-a*c)*t_mn'.t

   ▷ (a-b)**2*g_mn + a*(a-c)*t_mn
```

Performance I

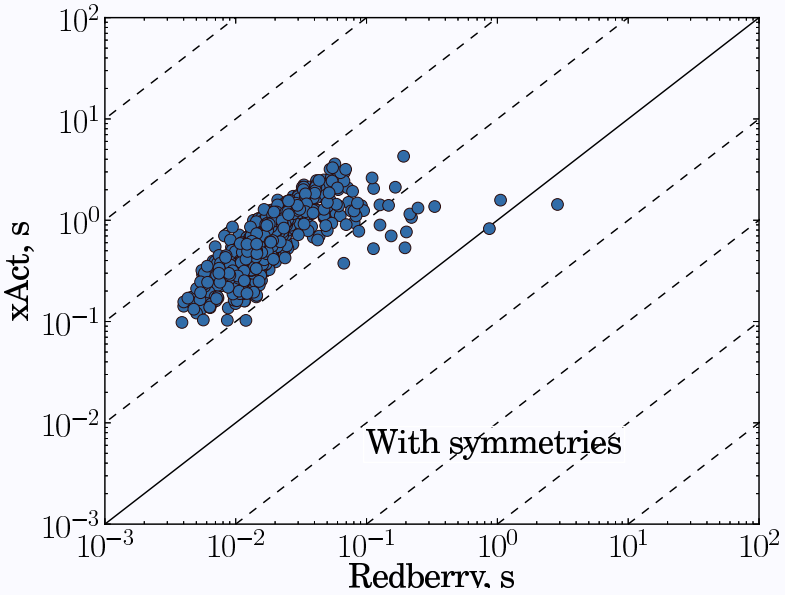


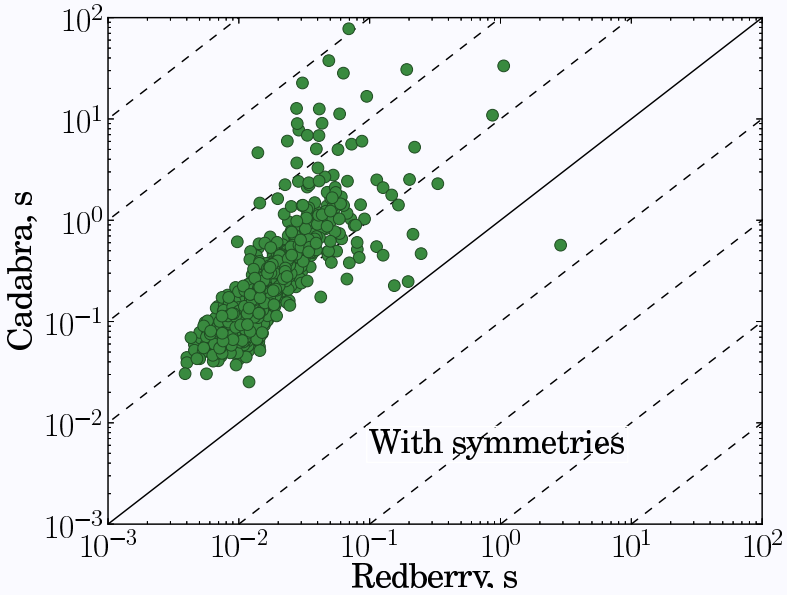


- **Test problem**: Simplify expression (collect same terms):

$$\left(F_{\mu\nu\alpha}\,T^{\nu}{}_{\beta}\,W^{\alpha\beta} + \ldots \text{(99 terms)}\right) - \left(T^{\alpha}{}_{\rho}\,W^{\nu\rho}\,F_{\mu\alpha\nu} + \ldots \text{(99 terms)}\right) = 0$$

Performance II





- **Test problem**: Simplify expression (expand and collect same terms):

$$\left(F_{\mu\nu}\left(T^{\mu\alpha}\,R^{\nu\beta}\,+\,\ldots\right)+\ldots\right)\left(R_{\alpha\rho}\left(F^{\rho}{}_{\beta}\,R_{\tau\gamma}\,+\,\ldots\right)+\ldots\right)\times\left(\ldots\right)-$$
$$\left(F_{\nu\mu}\,T^{\nu\beta}\,R^{\mu\alpha}\,R_{\beta\rho}\,F^{\rho}{}_{\alpha}\,R_{\tau\gamma}\,+\,\ldots\right)\,=\,0$$

Technical details

- `Redberry` is written in Java; the user interface is Groovy
- 150k lines of code with thousands of unit tests
- Runs on all platforms: Windows, Linux, OS X
- Free and open-source