Fermi National Accelerator Laboratory

FERMILAB–PUB–92/22–T

March, 1992

HIP — Symbolic High-Energy Physics Calculations Using Maple

Eran Yehudai[1]

Theory Group, M.S. 106

Fermi National Accelerator Laboraroty

P.O. Box 500, Batavia, IL 60510

Abstract

We present a *Maple* implementation of HIP — a package for symbolic calculation of Feynman diagrams and relativistic kinematics. The package was designed for performing calculations of tree-level high-energy processes, though it can be used for a much wider class of calculations.

[1] Internet: eran@fnth06.fnal.gov

Operated by Universities Research Association Inc. under contract with the United States Department of Energy

elliminated by calculating the helicity amplitudes without squaring them first. Several methods have been proposed for this purpose. One such method [8] has been implemented in the *Mathematica* version of HIP.

In the *Maple* implementation described here, two alternative techniques are available. The first is to substitute an explicit representation for the spinors, gamma matrices and other components of the Dirac algebra. We chose the representation described in reference [9]. This approach, applied after substituting explicit numeric values for the various four-vectors, serves as an excelent check to symbolic calculations. The second approach uses the Vector Equivalence technique [10]. We found this technique to be very useful in deriving symbolic expression at the matrix element level [11].

This paper proceeds as follows. In the next section we present a brief overview of HIP. In Section 3 we give a complete example of using HIP — the calculation of the cross-section for $e^+e^- \to \nu\bar{\nu}H$. Section 4 is the complete manual of HIP. Its contents are also available on-line. For each function in HIP one can find the appropriate calling sequence with a brief description of parameters, a synopsis, examples and cross-reference to related functions. In section 5 we conclude and give an outlook. A dictionary relating the *Mathematica* and *Maple* versions of HIP is given in the appendix.

2. Overview

HIP contains functions that can manipulate various mathematical objects occuring in high-energy physics such as four-vectors, spinors and gamma matrices. Rather than follow one strict path from input to output, the package allows the user to specify how a calculation proceeds (either interactively or in batch mode). A typical calculation might start with a matrix element, square it and sum over polarizations, construct

the phase-space integral and evaluate this integral to give a symbolic expression for the total cross-section.

The most fundamental component of any high-energy calculation is the manipulation of four-vectors. Basic objects such as the dot-product $(p \cdot q)$ (`p &. q`), the metric $g^{\mu\nu}$ (`metricg(mu, nu)` or `g(mu, nu)`) and the completely anti-symmetric Levi-Civita tensor $\epsilon^{\mu\nu\sigma\tau}$ (`eps(mu, nu, sig, tau)`) are defined, with some of their elementary properties (*e.g.* the dot-product is symmetric in its two arguments). Four-vectors can be specified in terms of their components. They can then be boosted (using the function `boost`), represented as sum of other four-vectors (`decay`), etc. In addition, four-vectors can also be treated without reference to the explicit components. Dot products can be given explicit values (`setdotproduct` or `setdp` and `setmass`), Mandelstam variables defined (`setmandelstam` or `setmand`), Lorentz indices defined (`setindex`) and contracted (`contract`).

The second component in HIP is the Dirac algebra. The basic objects involved are the Dirac gamma matrices γ^μ (`diracgamma(mu)` or `gm(mu)`), γ^5 (`diracgamma5` or `gm5`), the projection operators $P_\lambda = (1 + \lambda\gamma^5)/2$ (`helicityprojection(lambda)` or `proj(lambda)`) and $\not{p} = p_\mu\gamma^\mu$ (`diracgamma(p)` or `gm(p)`). The Dirac matrix multiplication is represented by the operator `&*`. The trace if a product of Dirac gamma matrices is computed using the function `gammatrace`.

Several functions are available for declaring properties of objects. A symbol or a function may be declared non-commutative (`setnoncommutative` or `setnc`), as a four-vector (`setfourvector` or `setfv`) or a Lorentz index (`setindex`). An expression can be declared real (`setreal`) and positive (`setpositive` or `setpos`).

Some programs handling Dirac algebra, notably Reduce, can only deal with gamma matrices. HIP can also work with spinors. The basic spinor objects $u(p)$ and $v(p)$ (`spinoru(p)` and `spinorv(p)` or `u(p)` and `v(p)`) and their conjugates $\bar{u}(p)$ and $\bar{v}(p)$ (`spinorubar(p)` and `spinorvbar(p)` or `ub(p)` and `vb(p)`) are defined. The function

`absolutevaluesquared` or `abssq` is used to square matrix element expressions which may include these spinors.

Expressions involving spinors do not have to be squared before they are calculated numerically. One possibility is to convert the various components of the Dirac algebra (spinors, Dirac matrices) into explicit vectors and matrices using the function `convert[explicit]`. The other approach is to use the function `convert[ve]` to convert the expression into the Vector Equivalence technique. Using this approach, each pair of external fermions is substituted by an equivalent four-vector. That four-vector (`vectorequivalenceV` or `VV`) can later be calculated explicitly in terms of the component of the fermion momenta.

Given an expression for the matrix element squared associated with a process, the calculation of physical observables such as cross sections and decay widths involves integration over the phase space of the out-going particles. The functions `crosssection` (or `cs`) and `decaywidth` (or `width`) set up the phase-space integral, and optionally evaluate it symbolically or numerically. Alternatively, one may convert the integrand to a C or Fortran expression for numeric integration.

3. Example: $e^+e^- \to \nu\bar{\nu}H$

In this section we give as an example a complete calculation of the matrix element for the process $e^+e^- \to \nu\bar{\nu}H$. To arrive at physical results one would only need to integrate over phase-space.

```
> setindex(mu, nu, tau, sig);
> setfv(p1, p2, p3, p4, p5);
> setmass([p1, p2, p3, p4, 0], [p5, mh]);
> setreal(mh, mw, G, p1, p2, p3, p4, p5);
```

Declare μ, ν, τ and σ as Lorentz indices; declare p_1, p_2, p_3, p_4 and p_5 as four-vectors; set the masses of external lines; declare the masses, coupling and four-vectors as real.

```
> mm := (ub(p3, -1) &* (-I * G/sqrt(2) * gm(mu) ** proj(-1)) &* u(p1)) *
  (-I * g(mu, nu)/((p1-p3)&.(p1-p3) - mw^2)) * (I*G*mw * g(nu, tau)) *
  (-I * g(tau, sig)/((p2-p4)&.(p2-p4)-mw^2)) * (ub(p2) &*
  (-I * G/sqrt(2) * gm(sig) ** proj(-1)) &* u(p4, -1)):
> mm := expand(contract(simplify(mm, '&*')));
```

$$
\begin{aligned}
\texttt{mm} := {} & \texttt{1/2 I } G^3 \texttt{ mw \&*(ub(p3, -1), gm(tau), proj(-1), u(p1))} \\
& \texttt{\&*(ub(p2), gm(tau), proj(-1), u(p4, -1))} \\
& \Big/ \Big/ \texttt{((- 2 (p1 \&. p3) - mw}^2\texttt{) (- 2 (p2 \&. p4) - mw}^2\texttt{))} \\
& \Big/
\end{aligned}
$$

Define the matrix element. After some simplification and contracting over Lorentz indices we get

$$\mathcal{M} = \frac{ig^3 m_W}{2} \frac{(\bar{u}_L(p_3)\gamma_\tau P_L u(p_1))(\bar{u}(p_2)\gamma_\tau P_L u_L(p_4))}{(-2(p_1 \cdot p_3) - {m_W}^2)(-2(p_2 \cdot p_4) - {m_W}^2)}.$$

```
> me2 := factor(contract(abssq(mm1)));
```

$$
\texttt{me2} := 4 \, \frac{G^6 \, \texttt{mw}^2 \, \texttt{(p2 \&. p3) (p1 \&. p4)}}{\texttt{(2 (p1 \&. p3) + mw}^2\texttt{)}^2 \, \texttt{(2 (p2 \&. p4) + mw}^2\texttt{)}^2}
$$

After squaring $\mathcal{M}$ (implicitly summing over fermion helicities) and contracting

Lorentz indices wee get

$$|\mathcal{M}|^2 = 4\frac{g^6 {m_w}^2 (p_2 \cdot p_3)(p_1 \cdot p_4)}{(2(p_1 \cdot p_3) + {m_w}^2)^2 (2(p_2 \cdot p_4) + {m_w}^2)^2}.$$

```
> cc := crosssection(me2, p1=[0, 0, sqrt(s)/2, sqrt(s)/2],
  p2=[0, 0, -sqrt(s)/2, sqrt(s)/2], cylindrical(p5, [p3, p4])):
> integrand := op(1, op(1, op(1, op(1, cc)))):
> subs(integrand = I, cc);
```

$$\int_0^{2\,Pi} \int_{-1}^{1} \int_{-1}^{1} \int_0^{(s^{1/2} - mh)^2} I \; dms1 \; dx2 \; dx1 \; dphi1$$

Calculate the total cross-section given initial momenta $p_1 = (0, 0, \sqrt{s}/2, \sqrt{s}/2)$ and $p_2 = (0, 0, -\sqrt{s}/2, \sqrt{s}/2)$. The result is an unevaluated integral

$$\sigma = \int_0^{2\pi} \int_{-1}^{1} \int_{-1}^{1} \int_0^{(\sqrt{s}-m_H)^2} I d(m_1^2)\, dx_2\, dx_1\, d\phi_1.$$

The integrand I is too long to print here. The integration variables are m_1^2 (the invarient mass of the $p_{34} = p_3 + p_4$), x_2 ($\cos\theta$ where θ is the angle between the beam pipe and p_5) and x_1 and ϕ_1 (specifying the direction of p_3 relative to p_{34} in the p_{34} center-of-mass frame.

```
> fortran(integrand, optimized);
      t1 = Pi**2
      t2 = t1**2

         :

      t166 = (t57-t137+t24)**2
      t169 = (t143+t161+t24)**2
      t179 = 1/t2*t20*t4*t22*t21*t24*(t57/2+t137/2)*(t143/2-t161/2)/t166
     #/t169/512
```

Using the *Maple* command `fortran` one can produce Fortran code to calculate the integrand for inclusion in a numeric integration program.

3. Reference Manual

1 HIP – Introduction to the HIP package

Calling Sequence:

function(args)
HIP[*function*](args)

Synopsis:

- The functions available are:

&.	&*	type/direction	type/explicitfv
type/fv	type/index	type/nc	type/pos
type/real	convert/diracgamma	convert/diracgamma5	convert/explicit
convert/helicityprojection	convert/ve	simplify/prod	simplify/fv
absolutevaluesquared	assumedpositive	boost	boostamount
commutator	commute	conj	contract
crosssection	decay	diracgammaexpand	energy
eps	fermionpropagator	gammatrace	mass
metricg	phasespacefactor	setaliases	setdotproduct
setfourvector	setindex	setmandelstam	setmass
setnoncommutative	setpositive	setreal	spacetimedimension
spinoru	vectorequivalenceU	vectorpolarization	

- The package implements the Dirac algebra using `&*` as a multiplication operator. Use `help(HIP, `&*`)` for more information.
- The package implements the Lorentz group using `&.` as the dot-product operator. Use `help(HIP, `&.`)` for more information.
- Many of the functions in HIP can have an alias defined for them. To use these aliases, invoke the command `setaliases()`.
- As an example, to find the trace of the expression `diracgamma(p) &* diracgamma(q)`, type:

```
setaliases(): setfv(p, q): gammatrace(gm(p) &* gm(q));
```

2 &. – Dot product in Minkowsky space

Calling Sequence:

p `&.` *k*
`&.`(*p*, *k*)

Parameters:

p, *k* – four-vectors

Synopsis:

- The operator `&.` is the dot-product defined in Minkovski space.
- The operator `&.` is binary and symmetric.
- If *p* and *k* are given explicitly, the dot-product is evaluated.

- The expression (*mu* **&.** *nu*) is automatically converted to **metricg**(*mu*, *nu*) if *mu* and *nu* are Lorentz indices.
- The expression (*p* **&.** *mu*) is automatically converted to *p*[*mu*] if *mu* is a Lorentz index and *p* is a four-vector, unless p is of type **function**.
- The expression *f*(...) **&.** *mu* where *f* is a four-vector valued function is **not** converted to *f*(...)[*mu*] (which is not allowed in maple).

Examples:

```
> setaliases():
> setfv(p, k):
  p &. k;                                  ⟶        k &. p
> [0, 0, p1, e1] &. [0, p2, p3, e2];       ⟶    e1 e2 - p1 p3
> setindex(mu);
  p &. mu;                                 ⟶        p[mu]
> mu &. mu;                                ⟶        g(mu, mu)
```

SEE ALSO: **metricg**, **type[fv]**, **type[index]**

3 &* – Dirac algebra multiplication

Calling Sequence:

x **&*** *y* **&*** ...
`&*`(*x*, *y*, ...)

Parameters:

x, *y*, ... – any expression

Synopsis:

- The operator **&*** is the non-commutative multiplication operator of the Dirac algebra.
- Objects which are scalars in the Dirac algebra can be commuted outside the **&*** operator using the command **simplify/&***.
- Objects belonging to the Dirac algebra may be commuted explicitly using the command **commute**.
- A **&*** product of sums of terms may be expanded using the command **expand**.
- New objects may be added to the Dirac algebra by using the command **setnoncommutative**.
- A **&*** product of Dirac algebra objects may be evaluated in a specific representation of the algebra using the command **convert/explicit**.
- A trace of a **&*** product may be evaluated using the command **gammatrace**.

SEE ALSO: **simplify[&*]**, **commute**, **commutator**, **convert[explicit]**, **setnoncommutative**, **type[nc]**, **diracgamma**, **diracgamma5**, **spinoru**, **helicityprojection**, **gammatrace**

4 type/direction – check for a direction specification

Calling Sequence:

type(*dir*, **direction**)

Parameters:

dir – any expression

Synopsis:

- The call `type(`*dir*`, direction)` checks to see if *dir* is of type `direction`. It returns true if *dir* is of type `direction`, and false otherwise.
- An expression is of type `direction` if it is a list of either two or three algebraic expressions.
- A direction can be specified either as a two component list (spherical coordinates) or a three component list (cartesian coordinates).
- In spherical coordinates, a direction is specified by the two component list `[`*cth*`, `*phi*`]`, where *cth* is the cosine of the angle between the direction and the *z* axis, and *phi* is the angle around the *z* axis.
- In cartesian coordinates, a direction is specified as the direction of the vector `[`*x*`, `*y*`, `*z*`]`. The length of the vector `[`*x*`, `*y*`, `*z*`]` is ignored.
- Functions requiring a direction specification typically allow a second direction specification *zaxis* specifying the axis with respect to which the first direction is measured.

Examples:

```
> setaliases():
> type([cth, phi], direction);          ⟶  true
> type([0, 0, 1], direction);           ⟶  true
> type([a, b, d, e], direction);        ⟶  false
```

SEE ALSO: `decay`, `boost`

5 type/explicitfv – check for an explicit fourvector

Calling Sequence:

`type(`*v*`, explicitfv)`

Parameters:

v – any expression

Synopsis:

- The call `type(`*v*`, explicitfv)` checks to see if *v* is of type `explicitfv`. It returns true if *v* is of type `explicitfv`, and false otherwise.
- An expression is of type `explicitfv` if it is a list with exactly four components.

Examples:

```
> setaliases():
> setfv(p):
  type([a, b, 1, m], explicitfv);       ⟶  true
> type(p, explicitfv);                  ⟶  false
> type(p, fv);                          ⟶  true
```

SEE ALSO: `type[fv]`, `energy`, `momentum`

6 type/fv – check for a fourvector

Calling Sequence:

`type(v, fv)`

Parameters:

v – any expression

Synopsis:

- The call `type(v, fv)` checks to see if *v* is of type `fv`. It returns true if *v* is of type `fv`, and false otherwise.
- An expression is of type `fv` if it was declared as such using `setfourvector` or if it of type `explicitfv`.
- A function can also be declared to be of type `fv`.
- A sum of fourvectors is not considered of type `fv`.

Examples:

```
> setaliases():
> setfv(p, kf):
  type(p, fv):                    ⟶   true
> type(kf(1, 3), fv);             ⟶   true
> type([a, b, 0, m], fv);         ⟶   true
> type(2*p, fv);                  ⟶   false
```

SEE ALSO: `setfourvector`, `type[explicitfv]`

7 type/index – check for a Lorentz index

Calling Sequence:

`type(mu, index)`

Parameters:

mu – any expression

Synopsis:

- The call `type(mu, index)` checks to see if *mu* is of type `index`. It returns true if *mu* is of type `index`, and false otherwise.
- An expression is of type `index` if it was declared as such using `setindex`.
- The expression `conj(mu)` is of type `index` if *mu* is of type `index`.

Examples:

```
> setaliases():
> setindex(mu, nu);
  type(mu, index);                ⟶   true
> type(conj(nu), index);          ⟶   true
> type(sig, index);               ⟶   false
```

SEE ALSO: `setindex`

8 type/nc – check for a non commuting object

Calling Sequence:

`type(`*expr*`, nc)`

Parameters:

expr – any expression

Synopsis:

- The call `type(`*expr*`, nc)` checks to see if *expr* is of type `nc`. It returns true if *expr* is of type `nc`, and false otherwise.
- An expression is of type `nc` if it was declared as such by `setnoncommutative`.
- The names `diracgamma`, `diracgamma5`, `spinoru`, `spinorubar`, `spinorv`, `spinorvbar`, `helicityprojection` and `&*` itself are initially of type `nc`.
- Sums and products of expressions of type `nc` are also of type `nc`.
- Objects which are not of type `nc` are taken out of a `&*` product by invoking `simplify/&*`.

Examples:

```
> setaliases():
> setnc(gm6, gm7);
  type(gm6, nc);                    ⟶   true
> type(gm7(p), nc);                 ⟶   true
> type(2*gm(p), nc);                ⟶   true
> type(p+q, nc);                    ⟶   false
```

SEE ALSO: `setnoncommutative`, `&*`, `diracgamma`, `diracgamma5`, `spinoru`, `spinorubar`, `helicityprojection`, `simplify[&*]`

9 type/pos – check for a positive symbolic expression

Calling Sequence:

`type(`*expr*`, pos)`
`type(`*expr*`, pos(`*a*`))`

Parameters:

expr – any expression
a – (optional) `true` or `false`

Synopsis:

- The call `type(`*expr*`, pos,` *a*`)` checks to see if *expr* is of type `pos` . It returns true if *expr* is of type `pos`, and false otherwise.
- An expression is of type `pos` if it was declared as such by `setpositive`, or if it was assumed as such by maple.
- The command `type/pos` always tries to factor an expression before determining whether it is of type `pos`. A product is considered positive if the product of the signs of its elements is 1.
- If *a* is true (default), and if *expr* was not declared as positive, nor was -*expr* declared as positive, maple assigns it a positive sign and prints a warning message.

- The sum of positive expressions is considered positive.

Examples:

```
> setaliases():
> setpositive(s, m);
  type(s, pos);                    ⟶   true
> type(s+m^2, pos);                ⟶   true
> type(s-m^2, pos(false));         ⟶   false
> type(s-m^2, pos(true));          ⟶        2
                                        true
```

SEE ALSO: **setpositive, assumedpositive**

10 type/real – check for a real symbolic expression

Calling Sequence:

type(*expr*, real)

Parameters:

expr - any expression

Synopsis:

- The call **type(*expr*, real)** checks to see if *expr* is of type **real**. It returns true if *expr* is of type **real**, and false otherwise.
- An expression is of type **real** if it was declared as such by **setreal**, it is the sum or product of expressions of type **real**. An integer power of an expression of type **real** and a fractional power of an expression of type pos are also of type **real**.

Examples:

```
> setaliases():
> setreal(x, y);
  type(x, real);                   ⟶   true
> type(x+2*y^2, real);             ⟶   true
> type(sqrt(x-y), real);           ⟶   Warning: assuming .(x - y). is positive
                                        true
> setpositive(y-x);
  type(sqrt(x-y), real);           ⟶   false
```

SEE ALSO: **setreal, setpositive, type[pos]**

11 convert/diracgamma – combine sums of diracgamma's

Calling Sequence:

convert(*expr*, diracgamma);

Parameters:

expr - any expression

Synopsis:

- The command **convert/diracgamma** converts sums of individual **diracgamma**'s of fourvectors into **diracgamma** of the sum of the fourvectors.
- The name **convert/gm** may be used as an alias to **convert/diracgamma** after first invoking **setaliases()**.

Examples:

```
> setaliases():
> convert(gm(mu) &* (gm(p) + gm(k)), gm);
                                gm(mu) &* gm(p + k)
```

12 convert/diracgamma5 – convert helicity projectors to diracgamma5

Calling Sequence:

convert(*expr*, diracgamma5)

Parameters:

expr - any expression

Synopsis:

- The command **convert/diracgamma5** converts all occurances of **helicityprojection** appearing in the expression to **diracgamma5**. All **diracgamma5**'s are then commuted to the left.
- The name **convert/gm5** may be used as an alias to **convert/diracgamma5** after first invoking **setaliases()**.

Examples:

```
> setaliases():
> convert(gm(p) &* gm(mu) &* proj(1), gm5);
             1/2 (gm(p) &* gm(mu)) + 1/2 I &*(gm5, gm(p), gm(mu))

> convert((gm(p) + m) &* gm5, gm5);
                             - (gm5 &* gm(p)) + m gm5
```

SEE ALSO: **convert[helicityprojection]**, **diracgamma5**, **helicityprojection**

13 convert/explicit – convert Dirac algebra objects to their explicit form

Calling Sequence:

convert(*expr*, explicit)
convert(*expr*, explicit, *mu*)
convert(*expr*, explicit, contract=*mu*)
convert(*expr*, explicit, contract=[mu_1, mu_2, ...])

Parameters:

expr - any expression

mu, mu_1, mu_2, ... - Lorentz indices

Synopsis:

- The command **convert/explicit** is used to convert Dirac algebra objects: **diracgamma**, **diracgamma5**, **helicityprojection**, **diracu**, **diracv**, **diracubar** and **diracvbar** to an explicit form of matrices and vectors in Dirac space.
- All the arguments to those functions must (where appropriate) be explicit fourvectors themselves. In addition, an explicit polarization value of 1 or -1 must be given in the case of **helicityprojection**, **diracu**, **diracv**, **diracubar** or **diracvbar**.
- Use **convert(** *expr*, **explicit**, *mu* **)** to convert an expression with one free Lorentz index into a fourvector.
- Use **convert(***expr*, **explicit**, **contract=***mu***)** to contract over the Lorentz index mu.
- Use **convert(***expr*, **explicit**, **contract=[***mu*$_1$, *mu*$_2$, ...**])** to contract over all Lorentz indices *mu*$_1$, *mu*$_2$, ...
- It is much more efficient to contract over Lorentz indices using **contract** before using **convert/explicit**.

Examples:

```
> setaliases():
> p := [0, 0, 0, 1];                                  ⟶   p := [0, 0, 0, 1]

> q := [0, 0, 1, 1];                                  ⟶   q := [0, 0, 1, 1]

> convert(ub(p, 1) &* u(q, -1), explicit)
  ;                                                   ⟶          1/2
                                                             2

> setindex(mu);
  convert(ub(p, -1) &* gm(mu) &* u(q, 1),
  explicit, mu);                                      ⟶                1/2     1/2
                                                      [0, 0, - 2    , - 2    ]

> k := [0, -1, 0, 1];                                 ⟶   k := [0, -1, 0, 1]

> convert(k[mu] * (ub(p, -1) &* gm(mu) &*
  u(q, -1)), explicit, contract=mu);                  ⟶              1/2
                                                           - I 2

# It is more efficient to use
  convert(contract( k[mu] * (ub(p, -1) &*
  gm(mu) &* u(q, -1)), mu), explicit);                ⟶              1/2
                                                           - I 2
```

SEE ALSO: **convert[ve]**, **contract**

14 convert/helicityprojection – convert diracgamma5 to helicity projectors

Calling Sequence:

convert(*expr*, **helicityprojection)**

Parameters:

expr – any expression

Synopsis:

- The command **convert/helicityprojection** converts all occurances of **diracgamma5** appearing in the expression to **helicityprojection**. Those are then commuted to the left.

- The name convert/proj may be used as an alias to convert/helicityprojection after first invoking setaliases().

Examples:

```
> setaliases():
> convert(gm(p) &* gm(mu) &* gm5, proj);
          &*(proj(1), gm(p), gm(mu)) - &*(proj(-1), gm(p), gm(mu))

> convert((gm(p) + m) &* proj(1), proj);
                       (proj(-1) &* gm(p)) + m proj(1)
```

SEE ALSO: convert[diracgamma5], helicityprojection, diracgamma5

15 convert/ve – convert to the Vector Equivalence technique

Calling Sequence:

convert(*expr*, ve)

Parameters:

expr – any expression

Synopsis:

- The command convert/ve expresses sub-expressions of the form spinorubar(p) &* ... &* spinoru(k) in terms of the Vector Equivalence functions vectorequivalenceU and vectorequivalenceV.
- If the four-vectors involved (and their helicities) are given explicitly then the Vector Equivalence functions are further evaluated in terms of the components of the four-vectors.
- If the helicity of a particular spinor is not given, a unique name is created for this helicity. A warning is printed with a reminder that that helicity has to be summed over **after** squaring the matrix element.

Examples:

```
> setaliases():
> setfv(p, k, q):
> cc := convert(ub(p, -1) &* (gm(k) + m) &* proj(1) &* v(q, 1), ve);
 cc := UU([p, -1, 1], [q, 1, -1], 1) m + (k &. VV([p, -1, 1], [q, 1, -1], -1))

> setpositive(e, pp, e+pp, e-pp, m):
> eval(subs(p=[0, 0, e, e], k=[0, 0, 0, m1], q=[0, 0, pp, e], cc));
                1/2  1/2         1/2          1/2  1/2          1/2
               2    e    (e - pp)    m - m1 2    e    (e + pp)

> setindex(mu);
> convert(ub(p) &* gm(mu) &* v(q), ve);
                     Warning: sum over l1 after squaring

                     Warning: sum over l2 after squaring

                  (mu &. VV([p, l1, 1], [q, l2, -1], -1))

                     + (mu &. VV([p, l1, 1], [q, l2, -1], 1))
```

SEE ALSO: convert[explicit], vectorequivalenceU, vectorequivalenceV

16 simplify/&* – simplify expressions involving &*

Calling Sequence:

`simplify(expr, `&*`)`

Parameters:

expr – any expression

Synopsis:

- The command `simplify/&*` performs several simplifications. It takes commuting objects outside the &* product, and applies several Dirac algebra identities

---

Examples:

```
> setaliases():
> setfv(p, k): setindex(mu):
> c1 := gm(p) &* m &* (k * gm(mu));
                          c1 := &*(gm(p), m, k gm(mu))

> simplify(c1, `&*`);
                               m k (gm(p) &* gm(mu))

> c2 := ub(k) &* gm(mu) &* gm(k) &* gm(k) &* gm(p) &* u(p, 1);
             c2 := &*(ub(k), gm(mu), gm(k), gm(k), gm(p), u(p, 1))

> simplify(c2, `&*`);
                             2
                   mass(k)  mass(p) &*(ub(k), gm(mu), u(p, 1))
```

SEE ALSO: `&*`, `setnoncommutative`, `type[nc]`

17 simplify/fv – simplify expressions involving explicit fourvectors

Calling Sequence:

`simplify(expr, fv)`

Parameters:

expr – any expression

Synopsis:

- The command `simplify/fv` attempts to simplify expressions involving explicit fourvectors by adding the components of sums of fourvectors and multiplying components of products of fourvectors and scalars.
- The expression may contain both explicit and non-explicit fourvectors.

Examples:

```
> setaliases():
> setfv(p):
> c := [0, 0, 2, 3] + [-2, 1, 0, 4] + p + b * [0, 0, 0, m];
              c := [0, 0, 2, 3] + [-2, 1, 0, 4] + p + b [0, 0, 0, m]

> simplify(c, fv);
                              [-2, 1, 2, b m + 7] + p
```

SEE ALSO: `setfourvector`

18 absolutevaluesquared – the square of the absolute value of an expression

Calling Sequence:

absolutevaluesquared(*expr*)

Parameters:

expr – any expression

Synopsis:

- The function **absolutevaluesquared** finds the square of the absolute value of an expression.
- The function **absolutevaluesquared** sums over polarizations of external spinors and vectors for which no explicit polarization is given.
- The **absolutevaluesquared** of an expression with spinors typically involves **gammatrace** of gamma matrices.
- When conjugating an expression with free Lorentz indices, **conj**(*mu*) is substituted for the index *mu*.
- The name **abssq** may be used as an alias to **absolutevaluesquared** after first invoking **setaliases()**.

Examples:

```
> setaliases():
> setfv(p, k);
> setmass([k, 0]);
> setindex(mu);
> abssq(ub(p) &* gm(mu) &* u(k));
     4 p[mu] k[conj(mu)] + 4 k[mu] p[conj(mu)] - 4 (k &. p) g(mu, conj(mu))
> abssq((vecpol(p) &. mu));
                                        p[mu] p[conj(mu)]
                    - g(mu, conj(mu)) + -----------------
                                                 2
                                           mass(p)
> abssq((vecpol(k) &. mu));
                              - g(mu, conj(mu))
> abssq((vecpol(k, 1) &. mu));
            (vecpol(k, 1) &. mu) (conj(mu) &. conj(vecpol(k, 1)))
> abssq(ub(p, 1) &* u(k, 1));
                     (k &. p) - mass(p) (k &. vecpol(p, 0))
```

SEE ALSO: **spinoru, spinorubar, spinorv, spinorvbar, vectorpolarization, conj**

19 assumedpositive – return a list of expressions assumed positive

Calling Sequence:

assumedpositive()

Synopsis:

- The function **assumedpositive** returns the list of expressions which were arbitrarily assumed positive by maple.
- It is highly recommended that whenever possible, expressions be explicitly set positive using **setpositive**.
- An expression is assumed positive whenever it is tested by `**type/pos**` and it was not (nor was its negative) previously declared as positive by **setpositive** or assumed positive.

- A warning message is printed whenever an expression is assumed positive.
- The name **asspos** may be used as an alias to **assumedpositive** after first invoking **setaliases()**.

Examples:

```
> setaliases():
> type(x, pos);                    ⟶   Warning: assuming x is positive
                                                        true
> type(x^2-y^2, pos);              ⟶   Warning: assuming .(x - y). is positive
                                       Warning: assuming .(x + y). is positive
                                                        true
> type(sqrt(z), real);             ⟶   Warning: assuming z is positive
                                                        true
> asspos();                        ⟶          {z, x, x - y, x + y}
> setpositive(x, z);
  asspos();                        ⟶            {x - y, x + y}
```

SEE ALSO: **setpositive**, **type[pos]**, **type[real]**

20 boost – boost an explicit four-vector

Calling Sequence:

boost(v, rap, dir, zaxis)

Parameters:

v	–	an explicit four-vector
rap	–	an algebraic expression
dir	–	a direction specification
zaxis	–	(optional) a direction specification

Synopsis:

- The function **boost**(*v*, *rap*, *dir*) returns the expicit four-vector of *v*, boosted in the direction *dir* by an amount *rap*.
- Boost amount *rap* is the change in rapidity along direction *dir*.
- The boost direction can be specified in either spherical or cartesian coordinates. See help for **type[dir]** for discussion of specifying directions.
- The optional third parameter *zaxis* is the direction with respect to which *dir* is measured. If none is specified, the default is the *z* axis **[0, 0, 1]**.

Examples:

```
> setaliases():
> v := [0, 0, 0, m];               ⟶          v := [0, 0, 0, m]
> w := [0, 0, p, e];               ⟶          w := [0, 0, p, e]
> setpositive(m, e, 1-cth, 1+cth, e-p, p)
  ;
  boost(v, r, [1, 0]);             ⟶    [0, 0, m sinh(r), cosh(r) m]
```

```
> boost(w, r, [0, 1, 0]);                    ⟶   [0, e sinh(r), p, cosh(r) e]

> boost(v, op(boostamount(w)));              ⟶        /    2      \1/2
                                                     |   e       |          e m
                                               [0, 0, |------- - 1|   , ------------]
                                                     | 2    2    |       2    2 1/2
                                                     \e  - p     /      (e  - p )
```

SEE ALSO: `type[direction]`, `boostamount`

21 boostamount – return rapidity and direction of a four-vector

Calling Sequence:

boostamount(*v*)
boostamount(*v*, *zaxis*)

Parameters:

v – an explicit four-vector
zaxis – a direction

Synopsis:

- The function **boostamount** returns a list with two components. The first is the rapidity of *v*. The second is the direction of *v* with respect to the *z* axis. If a second parameter *zaxis* is specified, the direction is calculated with respect to it.
- The function **boostamount** is the opposite function to **boost** in the sense that for any four-vector *v*, *v* = **boost**([0, 0, 0, **mass**(*v*)], op(**boostamount**(*v*))).

Examples:

```
> setaliases():
> setpositive(e, e-p, p, 1-cth, 1+cth);
> k := [sqrt(1-cth^2)*p, 0, cth*p, e];
                                    2 1/2
                     k := [(1 - cth )    p, 0, cth p, e]

> boostamount(k);
                                e                    2 1/2
                [arccosh(------------), [(1 - cth )    , 0, cth]]
                           2    2 1/2
                         (e  - p )

> boost([0, 0, 0, sqrt(e^2-p^2)], op(''));
                          /    2      \1/2
                          |   e       |              2 1/2
                 2    2 1/2|------- - 1|     (1 - cth )    , 0,
               [(e  - p )   | 2    2    |
                          | \e  - p     /

                                /    2      \1/2
                          2    2 1/2|   e       |
                    cth (e  - p )   |------- - 1|    , e]
                                | 2    2    |
                                \e  - p     /
```

```
> simplify(");
                                        2 1/2
                    [- I p (- 1 + cth )    , 0, - cth p, e]
```

SEE ALSO: boost, type[direction], mass

22 commutator – the commutator of two objects

Calling Sequence:

`commutator(x, y)`

Parameters:

x, *y* – non-commutative expressions

Synopsis:

- The expression `commutator(x, y)` is equivalent to *x* `&*` *y*, but involves their `&*` product in reverse order.
- If *x* and *y* obey some commutation relation *x* `&*` *y* - *y* `&*` *x* = `C(x, y)` then `commutator(x, y)` is set to *y* `&*` *x* + `C(x, y)`, **not** just `C(x, y)`.
- If *x* and *y* obey some anticommutation relation *x* `&*` *y* + *y* `&*` *x* = `A(x, y)` then `commutator(x, y)` is set to -*y* `&*` *x* + `C(x, y)`.
- The commutators of `diracgamma` objects between themselves and with `diracgamma5` are pre-defined.
- `commutator` is used by the function `commute` to commute any two adjacent non-commuting objects.

Examples:

```
> setaliases():
> setfv(p):
  setindex(mu):
  commutator(gm(mu), gm(p));          ⟶      - (gm(p) &* gm(mu)) + 2 p[mu]
> commutator(gm(mu), gm5);            ⟶           - (gm5 &* gm(mu))
```

SEE ALSO: setnoncommutative, commute, diracgamma, diracgamma5, &*

23 commute – commute two adjacent non-commuting objects

Calling Sequence:

`commute([`x_1`, `y_1`], [`x_2`, `y_2`], ..., `*expr*`)`

Parameters:

x_1, y_1, x_2, y_2, ... – any non-commuting objects

expr – any expression

Synopsis:

- The command `commute` commutes all adjacent occurances of x_i and y_i in a `&*` product by their commutator `commutator(`x_i`, `y_i`)`.
- If more then one pair is specified, they are all commuted sequentially.

Examples:

```
> setaliases():
> setindex(mu, nu):
> setfv(p, k):
> commute([gm(mu), gm(nu)], ub(p) &* gm(mu) &* gm(nu) &* v(k));
             &*(ub(p), - (gm(nu) &* gm(mu)) + 2 g(mu, nu), v(k))
> expand(");
       - &*(ub(p), gm(nu), gm(mu), v(k)) + 2 g(mu, nu) (ub(p) &* v(k))
```

SEE ALSO: commutator, setnoncommutative, diracgamma, diracgamma5, &*

24 conj – symbolic complex conjugate

Calling Sequence:

conj(*expr*)

Parameters:

expr – any expression

Synopsis:

- The function conj(*expr*) returns the complex conjugate of *expr*.
- If *r* is of type real then conj(*r*) returns *r*.
- If *mu* is of type index then conj(*mu*) is also of type index.
- In the case of a Dirac algebra object *d*, conj(*d*) is actually the result of commuting diracgamma(0) with conjugate(*d*).

Examples:

```
> setaliases():
> setindex(mu);
  setfv(p, k);
  setreal(p);
  conj(gm(p) &* gm(k));                    ⟶   gm(conj(k)) &* gm(p)
> conj(p[mu]);                             ⟶   p[conj(mu)]
> conj(diracgamma5);                       ⟶   - gm5
> conj(conj(k));                           ⟶   k
> conj(ub(k, 1) &* gm(mu) &* v(p));        ⟶   &*(vb(p), gm(conj(mu)), u(conj(k), 1))
```

SEE ALSO: setreal, absolutevaluesquared, diracgamma, diracgamma5, setindex

25 contract – contract with respect to a Lorentz index

Calling Sequence:

contract(*expr*, *mu*, *nu*, ...)

Parameters:

expr – any expression

mu, nu, ... – (optional) Lorentz indices

Synopsis:

- The command `contract(`*expr*`,` *mu*`)` returns *expr*, contracted over the Lorentz index *mu*.
- If no indices are specified, `contract(`*expr*`)` contracts over all the indices declared with `setindex`.
- If more then one index is specified, contraction is carried out with respect to all indices sequentially.
- The index being contracted has to appear exactly twice in every product in *expr*.

Examples:

```
> setaliases():
> setindex(mu,nu);
  setfv(p, k);
  contract(p[mu]^2, mu);                          ⟶   p &. p
> contract(g(mu, nu) * p[nu] * k[mu], nu)
  ;                                               ⟶   k[mu] p[mu]
> contract('', mu);                               ⟶   k &. p
> contract(gm(mu) &* gm(nu) &* gm(p) &*
  gm(mu), mu);                                    ⟶   4 p[nu]
> dimension := d:
  contract(gm(mu) &* gm(p) &* gm(nu) &*
  gm(mu), mu);                                    ⟶   4 p[nu] + (d - 4) (gm(p) &* gm(nu))
```

SEE ALSO: `setindex`, `metricg`, `spacetimedimension`, `diracgamma`, `eps`

26 crosssection – construct a phase-space integral for a cross-section
decaywidth – construct a phase-space integral for a decay width

Calling Sequence:

`crosssection(`*me2*`,` p_1`=[`x_1`,` y_1`,` z_1`,` t_1`],` p_2`=[`x_2`,` y_2`,` z_2`,` t_2`],` *outgoing*`,` *vars*`,` *evaloption*`)`
`decaywidth(`*me2*`,` *p*`=[`*x*`,` *y*`,` *z*`,` *t*`],` *outgoing*`,` *vars*`,` *evaloption*`)`

Parameters:

me2 – matrix element squared

p_1, p_2, *p* – implicit initial four-vectors

x, y, x_1, y_1, ... – components of explicit initial four-vectors

outgoing – description of outgoing particles

vars – (optional) a set of Mandelstam-like variables

evaloption – (optional) an evaluation option

Synopsis:

- The functions `crosssection` and `decaywidth` return a phase-space integral. The argument *evaloption* determines whether and how this integral is evluated.
- The formula used for the evaluation of the cross-section is:

$$cs := \frac{(2\,Pi)^4}{2\,s}\,\frac{2\,abs(q_1)}{sqrt(s)}\,Int(me2\ d\ Phi\ (q_1+q_2\ ;\ p_1,\ \ldots,\ p_n)$$

where $Phi_n(q_1+q_2;\ p_1,\ \ldots,\ p_n)$ is the n dimensional phase-space element. q_1 and q_2 are the momenta of the incoming particles and p_1, ..., p_n are the momenta of the outgoing particles.

- The formula used for the evaluation of the decay-width is:

$$width := \frac{(2\,Pi)^4}{2\,M}\,Int(me2\ d\ Phi\ (P\ ;\ p_1,\ \ldots,\ p_n)$$

where $Phi_n(P;\ p_1,\ \ldots,\ p_n)$ is the n dimensional phase-space element, P is the momentum of the decaying particle and M is its mass, p_1, ..., p_n are the momenta of the outgoing particles.

- Incoming momenta are specified using p_i=[x_i, y_i, z_i, t_i] where p_i is an implicit four-vector appearing in *me2* and [x_i, y_i, z_i, t_i] are its explicit components.
- The argument *outgoing* specifies the order of phase-space evaluation. The phase-space specification is built recursively; it takes of of the forms [ps_1, ps_2], `cylindrical`(ps_1, ps_2) or `spherical`(ps_1, ps_2). Here ps_i is either one of the final momenta or itself a phase-space specification.
- The keywords `cylindrical` and `spherical` can be used to indicated the symmetry of a two-body decay. If none is use, no symmetry is assumed.
- Use the parameter *vars* to specify the set of Mandelstam variables i.e. those symbols which depend on the dot-product of the outgoing particles and thus cannot be treated as constants in the phase-space integral.
- The argument *evaloption* specifies how and if the phase-space integral is to be evaluated.
- If *evaloption* is omited or is `inert`, the functions return an inert `Int` integral.
- If *evaloption* is `evaluate`, the functions attempt to evaluate the integral symbolically.
- If *evaloption* is `numeric`, the functions attempt to evaluate the integral numerically.
- The name `cs` may be used as an alias to `crosssection` after first invoking `setaliases()`.
- The name `width` may be used as an alias to `decaywidth` after first invoking `setaliases()`.

Examples:

```
> setaliases():
#
# Muon decay: pmu -> pe + pnu + pnubar
#
# M = mass(pmu)
#
> setfv(pnu, pnubar, pe, pmu):
> setmass([pnu, pnubar, pe, 0], [pmu, M]):
> setpos(M):
> me2 := 128 * Gf^2 * (pe &. pnu) * (pmu &. pnubar);
                                  2
                    me2 := 128 Gf  (pe &. pnu) (pmu &. pnubar)
```

```
> dw := width(me2, pmu=[0, 0, 0, M], spherical(cylindrical(pe, pnubar),
>   pnu), evaluate);
                                          5   2
                                         M  Gf
                                dw := 1/96 ------
                                             3
                                           Pi

#
# Compton scattering:
#     p1 (electron) + p2 (photon) -> p3 (electron) + p4 (photon)
#  (neglecting electron mass)
#
> setmand([p1, p2, p3, p4], [0, 0, 0, 0], s, t, u):
> setpos(s, -t, -u):
> me2 := -8*u/s - 8*s/u;
                                me2 := - 8 u/s - 8 s/u
> cc := cs(me2, p1=[0, 0, sqrt(s)/2, sqrt(s)/2],
>   p2=[0,0,-sqrt(s)/2,sqrt(s)/2], cylindrical(p3, p4), {u}):
> simplify(cc);
                                1
                                /                    2
                               |       5 + 2 x2 + x2
                               |  1/8 -------------- dx2
                               |        Pi s (1 + x2)
                              /
                              -1
```

SEE ALSO: **setmandelstam**

27 decay – decompose four-vector into decay products

Calling Sequence:

decay(*p*, *dir*, [m_1, m_2], 'p_1', 'p_2', *zaxis*)

Parameters:

p	–	an explicit four-vector
dir	–	a direction specification
m_1, m_2	–	masses of decay products
p_1, p_2	–	symbols which will take the value of the decay products' momenta
zaxis	–	(optional) a direction specification

Synopsis:

- The function **decay** calculates the momenta of the two decay products as a function of the momentum of the decaying particle, the masses of the decay products and the direction of the decay.
- The direction of the decay is given by *dir* and is taken in the center-of-mass frame of the decaying particle.
- The optional parameter *zaxis* specifies the axis with respect to which *dir* is measured. If *zaxis* is not specified, this axis is taken to be [0, 0, 1].

Examples:

```
> setaliases():
> p := [0, 0, 0, M];
                                p := [0, 0, 0, M]
```

```
> setpositive(m, M, M - m);
> decay(p, [1, 0], [m, 0], 'p1', 'p2');
> p1;
                                  /      2 \       2    2
                                  |     m  |      M  + m
                    [0, 0, 1/2 M |1 - ----|, 1/2 -------]
                                  |      2 |         M
                                  \     M  /

> p2;
                                 /      2 \         2    2
                                 |     m  |        M  + m
                  [0, 0, - 1/2 M |1 - ----|, - 1/2 ------- + M]
                                 |      2 |           M
                                 \     M  /

> k := [0.3, -0.5, 1.2, 2.6];
                              k := [.3, -.5, 1.2, 2.6]

> decay(k, [0.4, 2.1], [0.4, 0.7], 'k1', 'k2');
> mass(k1);
                                    .4000000036

> mass(k2);
                                    .6999999986

> simplify(k-k1-k2, fv);
                                          0
```

SEE ALSO: **mass**, **type[direction]**, **boost**

28 diracgammaexpand – expand diracgamma of a sum of four-vectors

Calling Sequence:

diracgammaexpand(*expr*)

Parameters:

expr – any expression

Synopsis:

- The command **diracgammaexpand** expands all sub-expressions of the form **diracgamma**(p_1+p_2+...) to **diracgamma**(p_1)+**diracgamma**(p_2)+...
- The name **gmexpand** may be used as an alias to **diracgammaexpand** after first invoking **setaliases()**.

Examples:

```
> setaliases():
> setfv(p1, p2, p3);
> mm := ub(p1) &* gm(p1 + p2 + p3) &* v(p2);
                    mm := &*(ub(p1), gm(p1 + p2 + p3), v(p2))

> gmexpand(mm);
                   &*(ub(p1), gm(p1) + gm(p2) + gm(p3), v(p2))

> expand(");
            mass(p1) (ub(p1) &* v(p2)) - mass(p2) (ub(p1) &* v(p2))
                 + &*(ub(p1), gm(p3), v(p2))
```

SEE ALSO: **diracgamma**, **expand**, **simplify[&*]**

29 energy – return the energy of a four-vector
momentum – return the space momentum of a four-vector

Calling Sequence:

energy(*p*)
momentum(*p*)

Parameters:

p – an explicit four-vector

Synopsis:

- The function **energy** returns the energy component of an explicit four-vector.
- The function **momentum** returns the space momentum of an explicit four-vector as a 3 component list.
- If *p* is not an explicit four-vector, the functions return unevaluated.

Examples:

```
> setaliases():
> setfv(k);
  p := [px, py, pz, e];          ⟶   p := [px, py, pz, e]
> energy(p);                     ⟶   e
> momentum(p);                   ⟶   [px, py, pz]
> energy(k);                     ⟶   energy(k)
```

30 eps – the completely anti-symmetric tensor

Calling Sequence:

eps(mu_1, mu_2, mu_3, mu_4)

Parameters:

mu_1, mu_2, mu_3, mu_4 – Lorentz indices or four-vectors

Synopsis:

- The tensor **eps** is antisymmetric in all its indices.
- The expression **eps**(*p*, ...) where *p* is a four-vector is equivalent to **eps**(*mu*, ...) * *p*[*mu*] contracted over *mu*.
- If all four arguments are explicit four-vectors, **eps** is calculated explicitly.
- If one of the arguments is a sum of four-vectors, the expression can be expanded using **expand**.

Examples:

```
> setaliases():
> setindex(mu, nu);
  setfv(p1, p2, p3, p4);
  eps(p1, p2, mu, nu);           ⟶   eps(p1, p2, mu, nu)
> contract(" * p3[mu], mu);      ⟶   - eps(p1, p2, nu, p3)
> subs(p3=2*p1-p2-p4, ");        ⟶   - eps(p1, p2, nu, 2 p1 - p2 - p4)
> expand(");                     ⟶   eps(p1, p2, nu, p4)
```

31 fermionpropagator – a propagator of a fermion line
scalarpropagator – a propagator of a scalar line
vectorpropagator – a propagator of a vector line

Calling Sequence:

`fermionpropagator(`*p*`, `*m*`)`
`scalarpropagator(`*p*`, `*m*`)`
`vectorpropagator(`*p*`, [`*mu*`, `*nu*`], `*m*`, `*ksi*`)`

Parameters:

p	–	a four-vector or a sum of four-vectors
m	–	(optional) an algebraic expression
mu, *nu*	–	(optional) Lorentz indices
ksi	–	(optional) an algebraic expression

Synopsis:

- The function **fermionpropagator** returns an expression for the propagator of a fermion line with a given momentum and mass.
- The function **scalarpropagator** returns an expression for the propagator of a scalar line with a given momentum and mass.
- The function **vectorpropagator** returns an expression for the propagator of a vector line with a given momentum and mass.
- If the mass *m* is not specified, it is taken to be 0.
- The indices *mu* and *nu* stand for the Lorentz indices at the two ends of the vector propagator.
- If the indices are not specified, the expression **vectorpropagator**(*p*, *m*) is the coefficient of **metricg**(*mu*, *nu*) in **vectorpropagator**(*p*, [*mu*, *nu*], *m*).
- The argument *ksi* specifies the gauge in which the vector propagator is returned.
- If *ksi* is not specified, the vector propagator is given in Feynman gauge (*ksi* = 1) for massless particles, and in Unitary gauge (*ksi* = **infinity**) for massive particles.
- The name **fermprop** may be used as an alias to **fermionpropagator** after first invoking **setaliases**().
- The name **scalprop** may be used as an alias to **scalarpropagator** after first invoking **setaliases**().
- The name **vecprop** may be used as an alias to **vectorpropagator** after first invoking **setaliases**().

Examples:

```
> setaliases():
> setfv(p);
  setindex(mu, nu);
  fermprop(p, m);                    ⟶        I (gm(p) + m)
                                              -------------
                                                          2
                                              (p &. p) - m
> fermprop(p);                       ⟶          I gm(p)
                                                -------
                                                 p &. p
> scalprop(p, m);                    ⟶            I
                                              -------------
                                                          2
                                              (p &. p) - m
```

```
> vecprop(p);                                  ⟶        I
                                                    - ------
                                                      p &. p

> vecprop(p, [mu, nu], m);                     ⟶          /            p[mu] p[nu]\
                                                      I |g(mu, nu) - ----------|
                                                        |                 2    |
                                                        \                m     /
                                                    - ------------------------
                                                                    2
                                                          (p &. p) - m

> vecprop(p, [mu, nu], m, ksi);                ⟶        /            (ksi - 1) p[mu] p[nu]\
                                                    I |g(mu, nu) + ---------------------|
                                                      |                              2  |
                                                      \               (p &. p) - ksi m   /
                                                  - ---------------------------------
                                                                    2
                                                          (p &. p) - m
```

SEE ALSO: **mass**, **type[index]**, **type[fv]**

32 gammatrace – trace a product of Dirac matrices

Calling Sequence:

gammatrace(*expr*)

Parameters:

expr – a product of Dirac matrices

Synopsis:

- The command **gammatrace** finds the trace of a product of matrices in Dirac space.
- The trace of the unit matrix **gammatrace(1)** is evaluated to the constant **gammasize**. The default value of **gammasize** is 4.

Examples:

```
> setaliases():
> setindex(mu, nu);
> setfv(p, k);
> gammatrace(gm(p) &* gm(k));
                                  4 (k &. p)

> gammatrace(gm(p) &* gm(mu) &* gm(k) &* gm(nu));
            4 p[mu] k[nu] + 4 k[mu] p[nu] - 4 (k &. p) g(mu, nu)

> gammatrace(gm(p) &* gm(mu) &* gm(k) &* gm(nu) &* proj(1));
  2 p[mu] k[nu] + 2 k[mu] p[nu] - 2 (k &. p) g(mu, nu) - 2 I eps(p, k, mu, nu)

> gammasize := gs:
> dimension := d:
> gammatrace(gm(mu) &* gm(k) &* gm(p) &* gm(mu));
                     4 gs (k &. p) + (d - 4) gs (k &. p)
```

SEE ALSO: **diracgamma**, **diracgamma5**, **helicityprojection**, **&***, **gammasize**, **spacetimedimension**

33 mass – the mass of a four-vector

Calling Sequence:

mass(*p*)

Parameters:

p – a four-vector

Synopsis:

- The function **mass** plays a double role. If the argument *p* is an explicit four-vector, the function returns **sqrt**(*p* **&.** *p*). If *p* is not an explicit four-vector, **mass**(*p*) stands for the mass of the particle carrying the momentum *p*.
- If *p* is the four-vector of an off-shell particle, **mass**(*p*) is **not** equal to **sqrt**(*p* **&.** *p*).
- The mass of a particle can be set using the command **setmass**.

Examples:

```
> setaliases():
> setfv(p);
> setmass([p, m]);
> mass(p);
                                    m

> mass([px, py, pz, e]);
                                2     2     2     2
          Warning: assuming .(e  - px  - py  - pz ). is positive

                              2     2     2     2 1/2
                            (e  - px  - py  - pz )
```

SEE ALSO: **energy**, **momentum**

34 metricg – the space-time metric

Calling Sequence:

metricg(*mu*, *nu*)

Parameters:

mu, *nu* – Lorentz indices

Synopsis:

- The function **metricg** is symmetric in its two arguments.
- Contracting **metricg**(*mu*, *mu*) with respect to *mu* returns **spacetimedimension**. The default value of **spacetimedimension** is 4.
- The name g may be used as an alias to **metricg** after first invoking **setaliases**().

Examples:

```
> setaliases():
> setindex(mu, nu);
  setfv(p);
  g(mu, nu) - g(nu, mu);                    ⟶              0

> contract(g(mu, nu) * p[mu], mu);          ⟶           p[nu]
```

```
> dimension := d;                          ⟶        dimension := d

> contract(g(mu, mu), mu);                 ⟶              d
```

SEE ALSO: **setindex, contract, &., spacetimedimension**

35 phasespacefactor – phase-space factor of a two body decay

Calling Sequence:

phasespacefactor(m, m_1, m_2)

Parameters:

m, m_1, m_2 – an algebraic expression

Synopsis:

- The function **phasespacefactor** returns the mass dependent part of the phase-space factor of a two body decay.
- The argument m is the mass of the decaying particle, m_1 and m_2 are the masses of the decay products.
- The function **phasespacefactor** is normalized to 1 if the two decay products are massless.

Examples:

```
> setaliases():
> phasespacefactor(M, 0, 0);  ⟶  1

> phasespacefactor(M, m, m);  ⟶  /        2 \1/2
                                 |       m  |
                                 |1 - 4 ----|
                                 |       2  |
                                 \      M  /

> phasespacefactor(M, m, 0);  ⟶      2
                                    m
                                1 - ----
                                     2
                                    M

> phasespacefactor(M, m1, m2);  ⟶  4     4     4      2   2      2   2
                                  (M  + m1  + m2  - 2 M  m1  - 2 M  m2

                                        2   2        / 2
                                   - 2 m1  m2 )^1/2 /  M
                                              /
```

SEE ALSO: **crosssection, decaywidth, mass, setmass**

36 setaliases – define aliases to various functions

Calling Sequence:

setaliases()

Synopsis:

- **setaliases** defines aliases to many functions in the HIP package.
- The following aliases are defined:

UU = vectorequivalenceU	VV = vectorequivalenceV
abssq = absolutevaluesquared	asspos = assumedpositive
cs = crosssection	dimension = spacetimedimension
fermprop = fermionpropagator	g = metricg
gm5 = diracgamma5	gm = diracgamma
gmexpand = diracgammaexpand	proj = helicityprojection
setdp = setdotproduct	setfv = setfourvector
setmand = setmandelstam	setnc = setnoncommutative
setpos = setpositive	u = diracu
ub = diracubar	v = diracv
vb = diracvbar	vecpol = vectorpolarization
vecprop = vectorpropagator	width = decaywidth

Examples:

```
> setaliases():
> diracgamma(p);
                                    gm(p)
```

37 setdotproduct – give a value to a dot-product of two four-vectors

Calling Sequence:

setdotproduct(*p*, *k*, *dp*)
setdotproduct([p_1, k_1, dp_1], ...)

Parameters:

p, *k*, p_1, k_1, ... – four-vectors
dp, dp_1, ... – any expression

Synopsis:

- The command **setdotproduct**(*p*, *k*, *dp*) sets *p* **&.** *k* equal to *dp*.
- Use **setdotproduct**([p_1, k_1, dp_1], ...) to set several dot-products.
- The name **setdp** may be used as an alias to **setdotproduct** after first invoking **setaliases**().

Examples:

```
> setaliases():
> setfv(p, k);
  setdp(p, k, s/2);
  setdp([p, p, m^2], [k, k, 0]);
  (p + k) &. (p + k);                    ⟶        (p + k) &. (p + k)
> expand(");                             ⟶              2
                                                    m  + s
```

SEE ALSO: **setmandelstam**, **&.**, **setmass**, **setfourvector**

38 setfourvector – declare a name or a function as a four-vector

Calling Sequence:

setfourvector(p_1, p_2, ...)

Parameters:

p_1, p_2, ... – any expression

Synopsis:

- The command **setfourvector** is used to declare certain objects as being of type **fourvector**.
- See help for **type[fv]** for more information.
- The name **setfv** may be used as an alias to **setfourvector** after first invoking **setaliases()**.

SEE ALSO: **type[fourvector], setindex, type[index]**

39 setindex – declare a name as an index

Calling Sequence:

setindex(mu_1, mu_2, ...)

Parameters:

mu_1, mu_2, ... – any expression

Synopsis:

- The command **setindex** is used to declare certain objects as being of type **index**.
- See help for **type[index]** for more information.

SEE ALSO: **type[index], contract, setfourvector, type[fourvector]**

40 setmandelstam – declare masses and dot-products in a 2–>2 process

Calling Sequence:

setmandelstam([p_1, p_2, p_3, p_4], [m_1, m_2, m_3, m_4], s, t, u)

Parameters:

p_1, p_2, p_3, p_4	–	names
m_1, m_2, m_3, m_4, s, t	–	algebraic expressions
u	–	(optional) algebraic expression

Synopsis:

- The command **setmandelstam** makes several definitions related to the p_1 + p_2 --> p_3 + p_3.
- The command declares p_1, p_2, p_3 and p_4 to be of type **fourvector**.
- The command declares the masses of p_1, p_2, p_3 and p_4 to be m_1, m_2, m_3 and m_4 respectively.
- The command sets the dot-products of the various p's to be consistent with the following relations:

$$(p_1 + p_2)^2 = s$$
$$(p_1 - p_3)^2 = t$$
$$(p_1 - p_4)^2 = u$$

- If the argument u is omitted, it is replaced by $m_1^2 + m_2^2 + m_3^2 + m_4^2 - s - t$.
- The name **setmand** may be used as an alias to **setmandelstam** after first invoking **setaliases()**.

Examples:

```
> setaliases():
> setmand([p1, p2, p3, p4], [m1, m2, m3,
  m4], s, t, u);
  type(p1, fv);                          ⟶   true

> mass(p2);                              ⟶   m2

> p1 &. p4;                              ⟶                    2          2
                                          - 1/2 u + 1/2 m1  + 1/2 m4

> setmand([k1, k2, k3, k4], [m1, m2, m3,
  m4], s, t);
  k1 &. k4;                              ⟶          2          2
                                          - 1/2 m2  - 1/2 m3  + 1/2 s + 1/2 t
```

SEE ALSO: **setfourvector, setmass, setdotproduct**

41 setmass – set the mass of a particle

Calling Sequence:

setmass(p_1, ..., p_n, m)
setmass([p_1, ..., p_n, m_p], [k_1, ..., k_n, m_k], ...)

Parameters:

p_1, ..., p_n, k_1, ..., k_n, ... – four-vectors
m, m_p, m_k, ... – algebraic expression

Synopsis:

- The command **setmass** is used to set the mass of the particle carrying a particular momentum.
- See help for **mass** for more information.

SEE ALSO: **mass, setdotproduct, setmandelstam**

42 setnoncommutative – declare a name or a function as being of type nc

Calling Sequence:

setnoncommutative(x_1, x_2, ...)

Parameters:

x_1, x_2, ... – any expression

Synopsis:

- The command **setnoncommutative** is used to declare certain names or functions as being of type nc.
- See help for **type[nc]** for more information.
- The name **setnc** may be used as an alias to **setnoncommutative** after first invoking **setaliases()**.

SEE ALSO: **type[nc], &*, simplify[&*]**

43 setpositive – declare an expression as being positive

Calling Sequence:

setpositive($expr_1$, $expr_2$, ...)

Parameters:

$expr_1$, $expr_2$, ... – any expression

Synopsis:

- The command **setpositive** is used to declare certain expression as being of type **pos**.
- See the help for **type[pos]** for further information.
- The name **setpos** may be used as an alias to **setpositive** after first invoking **setaliases()**.

Examples:

```
> setaliases():
> setpos(e, p, m, e-p, e-m);
  type(e, pos);                    ⟶   true
> type(e^2-m^2, pos);              ⟶   true
> type(p-m, pos);                  ⟶   Warning: assuming .(p - m). is positive
                                       true
```

SEE ALSO: **type[pos]**

44 setreal – declare a name or a function as being of type real

Calling Sequence:

setreal(r_1, r_2, ...)

Parameters:

r_1, r_2, ... – any expression

Synopsis:

- The command **setreal** is used to declare certain names and functions as being of type **real**.
- See help for **type[real]** for more information.

SEE ALSO: **type[real]**

45 spacetimedimension – number of space-time dimensions gammasize – trace of unit matrix in Dirac space

Calling Sequence:

spacetimedimension
gammasize

Synopsis:

- The constant **spacetimedimension** is the dimensionality of space-time. Its default value is 4.
- The constant **gammasize** is the size of Dirac gamma matrices. Its default value is 4.
- The name **dimension** may be used as an alias to **spacetimedimension** after first invoking **setaliases()**.

Examples:

```
> setaliases():
> setindex(mu, nu);
  dimension := d;                    ⟶        dimension := d
> gammasize := 2^(d/2);              ⟶                  (1/2 d)
                                              gammasize := 2
> contract(g(mu, mu), mu);           ⟶              d
> gammatrace(gm(mu) &* gm(nu));      ⟶       (1/2 d)
                                             2        g(mu, nu)
```

SEE ALSO: **metricg, gammatrace**

46 vectorequivalenceU – scalar Vector Equivalence function U
vectorequivalenceV – vector Vector Equivalence function V

Calling Sequence:

vectorequivalenceU($[p_1, l_1, s_1]$, $[p_2, l_2, s_2]$, h)
vectorequivalenceV($[p_1, l_1, s_1]$, $[p_2, l_2, s_2]$, h)

Parameters:

p_1, p_2	–	four-vectors
l_1, l_2, s_1, s_2	–	(optional) a number, either -1 or 1
h	–	a number, either -1 or 1
mu, *nu*	–	either a Lorentz index or a four-vector

Synopsis:

- The functions **vectorequivalenceU** and **vectorequivalenceV** are the ingrediants in the Vector Equivalence technique for calculating Feynman diagrams at the matrix element level.
- The expression **vectorequivalenceU**($[p_1, l_1, 1]$, $[p_2, l_2, 1]$, h) is equal to **spinorubar**(p_1, l_1) **&* helicityprojection**(h) **&* spinoru**(p_2, l_2).
- The expression **vectorequivalenceV**($[p_1, l_1, 1]$, $[p_2, l_2, 1]$, h)[*mu*] is equal to **spinorubar**(p_1, l_1) **&* diracgamma**(*mu*) **&* helicityprojection**(h) **&* spinoru**(p_2, l_2).
- In each function, s_1 = -1 corresponds to replacing **spinorubar** by **spinorvbar**. Using s_2 = -1 implies substituting **spinorv** for **spinoru**.
- If any of the four-vectors is not given explicitly, or if any of the other arguments is not given a numeric value, the functions return unevaluated.
- The arguments l_1 and l_2 are optional. The argument l_1 defaults to h in **vectorequivalenceV** and to $-h$ in **vectorequivalenceU**. The argument l_2 defaults to h in both **vectorequivalenceU** and **vectorequivalenceV**.
- The arguments s_1 and s_2 are optional, and default to 1.
- Converting an ordinary expression to Vector Equivalence functions is done using the command **convert/ve**.
- The name **UU** may be used as an alias to **vectorequivalenceU** after first invoking **setaliases()**.
- The name **VV** may be used as an alias to **vectorequivalenceV** after first invoking **setaliases()**.

Examples:

```
> setaliases():
> p := [0.2, 0.5, -0.6, 1.6];        ⟶        p := [.2, .5, -.6, 1.6]
```

```
> k := [0.7, -1.2, 0.3, 1.8];                          ⟶   k := [.7, -1.2, .3, 1.8]
> UU([p, 1, 1], [k, 1, -1], 1);                        ⟶   .08592627752 - .4763256523 I
> convert(ub(p, 1) &* proj(1) &* v(k, 1),
  explicit);                                           ⟶   .08592627752 - .4763256523 I
> q := [-0.4, 1.2, 0.9, 2.5];                          ⟶   q := [-.4, 1.2, .9, 2.5]
> VV([p, -1, -1], k, 1) &. q;                          ⟶   3.353465374 + 5.869476232 I
> convert(vb(p, -1) &* gm(q) &* proj(1) &*
  u(k, 1), explicit);                                  ⟶   3.353465376 + 5.869476233 I
> r := [0, 1.1, -0.1, 1.3];                            ⟶   r := [0, 1.1, -.1, 1.3]
```

SEE ALSO: **convert[ve]**, **convert[explicit]**

47 vectorpolarization – the polarization of a four-vector

Calling Sequence:

vectorpolarization(*p*, *h*)

Parameters:

p – a four-vector

h – (optional) -1, 0 or 1

Synopsis:

- The expression **vectorpolarization**(*p*) symbolizes any polarization vector of the four-vector *p*.
- If *h* is not specified, **absolutevaluesquared** sums over all polarizations.
- If *p* is an explicit four-vector and *h* is either -1, 0 or 1 then the function **vectorpolarization**(*p*, *h*) returns an explicit four-vector.
- The value *h*=0 corresponds to the longitudinal polarization. Values of -1 and 1 to *h* corresponde to the left and right handed polarizations respectively.
- If the mass of *p* is zero, **vectorpolarization**(*p*, 0) prints an error message.
- If *p* describes a particle at rest, **vectorpolarization**(*p*, 0) arbitrarily returns [0, 0, 1, 0] while **vectorpolarization**(*p*, +/-1) returns [1/sqrt(2), +/-1/sqrt(2), 0, 0].
- Specifying a Lorentz index should be done using **&.** as in (**vectorpolarization**(*p*) **&.** *mu*) rather than **vectorpolarization**(*p*)[*mu*].
- The dot product of a four-vector with any of its polarizations is always 0.
- The name **vecpol** may be used as an alias to **vectorpolarization** after first invoking **setaliases()**.

Examples:

```
> setaliases():
> setpositive(e, p, e-p);
  setreal(e, p);
  setfv(q1, q2);
  setindex(mu);
  setmass([q2, 0]);
  k := [0, 0, p, e];                                   ⟶   k := [0, 0, p, e]
```

```
> vecpol(k, 0);                 -->                  e                 p
                                      [0, 0, ------------, ------------]
                                               2    2 1/2     2    2 1/2
                                             (e  - p )      (e  - p )

> vecpol(k, 1);                 -->              1/2         1/2
                                          [1/2 2    , 1/2 2     I, 0, 0]

> abssq(vecpol(q1) &. mu);      -->                       q1[mu] q1[conj(mu)]
                                     - g(mu, conj(mu)) + -------------------
                                                                    2
                                                             mass(q1)

> abssq(vecpol(q2) &. mu);      -->            - g(mu, conj(mu))

> vecpol(q1) &. q1;             -->                    0
```

SEE ALSO: `&.`, `absolutevaluesquared`

5. Conclusion and Outlook

We developed HIP as an aid in the calculation of tree-level processes in high-energy physics. HIP's main feature is in providing an environment within *Maple* in which one can refer to objects and perform operations that occur frequently in this field. One can use HIP interactively to assist with small calculations, or set it up to automatically handle massive problems.

HIP fits naturally as one component in the process of a completely automatic calculation of Feynman diagrams. The program does not generate the Feynman diagrams, nor is it particularly suitable for multi-dimensional phase-space integration. Feynman diagram generation and conversion into a numeric integration code can be done manually. Ideally, however, HIP could be tied with existing programs for this purpose. It is hoped that in the near future, a set of standards is adopted to allow the various programs, including HIP, to share data.

HIP is available for distribution. The distribution package includes the HIP code and on-line documentation. Interested readers should contact the author for details.

Appendix

In this document we only describe the *Maple* implementation of HIP. The *Mathematica* implementation is described in [1]. The two implementations are essentially similar. The major differences are:

- Two new methods for calculating helicity amplitudes without squaring have been added: explicit method and the Vector Equivalence technique.
- The on-line help has been greatly extended and includes examples and cross-references.
- The spinor technique functions have been eliminated

- A compilation of Standard Model Feynman rules is not included with the package, though it can easily be added.

Most HIP-*Maple* functions have names which are just lower-cased versions of the HIP-*Mathematica* names. For example, `DiracGamma` in HIP-*Mathematica* corresponds to `diracgamma` in HIP-*Maple*. Table 1 is a translation table for the functions that do not fall under this rule.

Mathematica	*Maple*
`CommutativeQ`	`type[nc]`
`ConvertToChiralFermions`	`convert[helicityprojection]`
`ConvertToGamma5`	`convert[diracgamma5]`
`DiracGammaSize`	`gammasize`
`DotProduct`	`&.`
`EvaluatePhaseSpaceIntegral`	`crosssection, decaywidth`
`ExpandSlash`	`diracgammaexpand`
`G`	`metricg`
`HeavyVectorPolarization`	`vectorpolarization`
`LongitudinalPolarization`	
`MasslessVectorPolarization`	
`NEvaluatePhaseSpaceIntegrate`	`crosssection, decaywidth`
`NonCommutativeExpand`	`expand`
`PrepareIndex`	`setindex`
`Propagator`	`fermionpropagator`
	`scalarpropagator`
	`vectorpropagator`
`Slash`	`diracgamma`

Table 1: A translation between the *Mathematica* and *Maple* versions of HIP.

Functions in HIP-*Mathematica* with no equivalence in HIP-*Maple* are:
`CommutativeAllQ`, `ConvertToMassless`, `ConvertToST`, `DotProductRules`, `KobayashiMaskawa`, `LightlikeVectorDecayedFrom`, `LightlikeVectorNotCollinearWith`, `Mandelstam`, `MandelstamRules`, `NonCommutativeFactor`, `Opposite`, `PerpendicularMomentum`, `PhaseSpaceIntegral`, `PolarizationCombinations`, `STToTraces`, `SpaceDirection`, `SpinorS`, `SpinorT`, `Vertex` and `ZAxis`.

Functions in HIP-*Maple* with no equivalence in HIP-*Mathematica* are:
`convert[explicit]`, `convert[ve]`, `assumedpositive`, `setaliases`, `setfourvector`, `setpositive`, `vectorequivalenceU` and `vectorequivalenceV`.

References

[1] A. Hsieh and E. Yehudai, Report No. SLAC-PUB-5576 (1991) (to be published in *Computers in Physics*).

[2] *Mathematica*: A System for Doing Mathematics by Computer, S. Wolfram, Addison-Wesley Publishing Company, 1988.

[3] *Maple* by Waterloo Maple Corp., see MAPLE Reference Manual, Fifth Edition, Waterloo Maple Publishing, 1988.

[4] J. Küblbeck, M. Böhm and A. Denner, *Comp. Phys. Comm.* **60**(1990), 165.

[5] T. Kaneko and H. Tanaka, Report No. ICRR-238-91-7 (1991).

[6] E. E. Boos *et al.*, Report No. PREPRINT 89-63/140 (1989).

[7] H. Murayama, I. Watanabe, K. Hagiwara, "HELAS, HELicity Amplitude Subroutines for Feynman diagram evaluations", to appear in a KEK-Report.

[8] R. Kliess, W. J. Stirling, *Nucl. Phys.* **B262**(1985), 235.

[9] K. Hagiwara, D. Zeppenfeld, *Nucl. Phys.* **B274**(1986), 1.

[10] E. Yehudai, "The Vector Equivalence Technique", to appear as a Fermilab Report.

[11] E. Yehudai, Report No. SLAC-383, 1991 (Ph.D. dissertation).