User Guide: `FDiracFormalism` Toolbox

Brief description

This document is a usage guide for the `FDiracFormalism` (MATLAB toolbox). The toolbox provides functions that automate the full Dirac constraint (Dirac–Bergmann) workflow for singular systems, and also includes several highly effective routines for simplifying symbolic expressions.

The package is designed to work with **indexed expressions** directly (using the `_` / `__` conventions described below), so in general you do *not* need to manually expand or rewrite indices before using the routines.

Conventions

Core symbolic syntax (indices, derivatives, and deltas)

Index notation in MATLAB strings

The toolbox reads index structure from the *string form* of symbolic expressions. The convention is:

- A **lower index** is written with a single underscore: `_`.
- An **upper index** is written with a double underscore: `__`.
- Multiple indices are written sequentially, e.g. `A_i_j`, `g__mu__nu`, etc.

Consistent naming is essential: if the same object is written with inconsistent index patterns, index-handling and simplification routines may become unreliable.

Dirac delta distributions

The toolbox inputs/outputs `dirac(...)` using MATLAB's standard notation. In field-theory applications, these should typically be interpreted as spatial Dirac deltas, i.e.

$$\delta(x-y) \equiv \delta^{(3)}(\mathbf{x}-\mathbf{y}),$$

unless otherwise stated in the specific workflow.

Partial derivatives

MATLAB does not allow `\partial` as a symbolic operator. For that reason, partial derivatives are represented using the symbol `d`. For example:

$$\partial_i f(x) \;\longleftrightarrow\; \texttt{d\_i(f(x))}, \qquad \partial_i^{(x)} f(x) \;\longleftrightarrow\; \texttt{d\_i\_\_x(f(x))}.$$

Some routines also support covariant-derivative wrappers written as `D_` (when enabled in the corresponding function).

Modeling conventions for canonical variables

Canonical fields and conjugate momenta (`pi`)

A key convention in the toolbox is:

- Canonical coordinates (fields) are written with **lower indices** (e.g. `A_i`, `A_mu`).
- Canonical momenta are written with **upper indices** and are denoted with `pi` (e.g. `pi__i`, `pi__mu`).

For correct usage of routines such as `PoissonBrackets` and `DiracBrackets`, you must follow this rule. For example:

$$\{A_j(x), \pi^i(y)\} \quad \longleftrightarrow \quad \texttt{PoissonBrackets(A\_j(x), pi\_\_i(y), ...)}.$$

Using a different momentum symbol or inconsistent index placement may cause the bracket-resolution logic to fail or return incorrect simplifications.

Index-position policy (recommended)

As a general rule, enter tensors and fields using *lower* indices (e.g. `A_mu`, `J_i`, `F_i_j`). Contravariant components must be written explicitly using the double-underscore convention (e.g. `pi__mu`, `F__i__j`) and only when a genuine upper index is intended.

Index naming, contractions, and internal dummy indices

Latin vs. Greek index names

It is recommended that *Latin* indices use at most two characters (e.g. `A_j`, `A_jk`). Longer multi-letter indices should be reserved for *Greek* names (e.g. `_mu`, `_nu`, `_sigma`). The internal parsers recognize Greek patterns such as `mu`, `nu`, `rho`, `sigma`, etc.; if a Latin index is written with many letters, the toolbox may mistakenly interpret it as Greek and treat it accordingly.

Mixed contractions (Latin/Greek)

The toolbox supports mixed-index contractions (Latin and Greek) at the symbolic level. When a contraction is detected, the code checks whether it is *admissible* (consistent index type and position). If valid, it is carried out automatically; if not, the term is left explicit, without forcing an incorrect simplification.

Dummy-index safety (recommended)

Several routines introduce dummy indices internally (e.g. `gamma`, `rho`, `sigma`, `lambda`) to perform transformations safely. To avoid index clashes, it is recommended that user input expressions do **not** already contain these internal dummy indices unless the specific function explicitly allows it.

Constants, parameters, and protected bases

Several routines (field derivatives, canonical momenta, Poisson/Dirac brackets, etc.) must distinguish between *dynamical fields* and *constant parameters*. In the current version there is **no built–in default list of constants**. Instead, the user must explicitly register which symbol bases are to be treated as constants through `DefineConstant`.
The recommended workflow is:

- At the beginning of *each* document/session where the toolbox is used, call

```
DefineConstant('set', {'Z','theta','C','c','O','H','g'});
```

  (or any list appropriate for your model) to declare all bases that must be treated as constants.
- You can inspect the current list at any time via

```
constList = DefineConstant('get');
```

Any base symbol included in this list is treated as a *constant* (i.e. not as a dynamical field) by the functional-derivative machinery. It is therefore strongly recommended to call `DefineConstant('set', ...)` at the start of every new document.

Scope limitations and practical recommendations

Potentials (momentum-dependent potentials are not supported)

The derivative/field helpers are **not designed** to interpret momentum-dependent potentials. If the Lagrangian contains a term of the form

$$V(A, \pi, \ldots) \quad (\text{e.g. } V = \tfrac{\alpha}{2}\,\pi_\mu \pi^\mu),$$

these routines may mis-handle dummy indices, drop contributions, or trigger internal errors. The recommended workflow is to use interaction potentials that depend only on the fields, e.g.

$$V_{\text{int}}(A) = \frac{\lambda}{4}\left(A_\mu A^\mu\right)^2,$$

and to insert such concrete potentials explicitly once the canonical structure has been constructed.

Higher-order partial derivatives

At the moment, third (and higher) order partial derivatives are not fully supported. Terms of the schematic form

$$d_i\big(d_j(d_k(A_\mu))\big)$$

can still be written and passed to the toolbox, but simplifications may be incorrect, and some internal routines can throw errors or behave unpredictably. In the current version it is strongly recommended to restrict inputs to at most second-order derivatives.

Avoid fractions when possible

Many internal simplification steps rely on pattern matching and term grouping. Whenever possible, avoid introducing explicit rational prefactors early in the workflow (e.g. write `(1/2)*...` only when necessary). This can significantly improve simplification stability.

Debug / verbose mode

For high-level routines that *compute* objects (rather than just rewriting expressions) — such as `CalculateCanonicalMomentum`, `CalculateHessian`, and similar functions — user-facing messages are disabled by default. A standard call returns only the final symbolic result.

If you want to inspect the full internal procedure (intermediate simplifications, index manipulations, substitutions, etc.), call the routine with an additional input argument `'debug'` as the last parameter. When present, this flag re-enables verbose console output:

```
pi       = CalculateCanonicalMomentum(L, 'debug');
[H,dL]   = CalculateHessian(L, varList, 'debug');
Hc       = CalculateHamiltonian(L, 'debug');
```

1 Partial Derivation

1.1 `fieldDerivative`

```
function result = fieldDerivative(expr, expr1, extraIndex, options)
```

Description

`fieldDerivative` constructs a symbolic representation of a *field-like* **partial** derivative of an input expression with respect to a variable (or coordinate) `expr1`. The function applies basic differentiation rules (linearity, product rule, and power rule) by parsing the expression as a string, while treating certain symbols as constants according to internal heuristics. Since the output is generated using custom symbolic wrappers, the true partial-derivative symbol ∂ is not used; instead, the prefix `d_` is employed to denote partial derivatives. It also supports a `DC` mode, where the prefix is switched to `D_` to represent covariant (∇) derivatives.

Input arguments

Name	Type	Meaning
expr	*sym*	Symbolic expression to differentiate.
expr1	*char*	Differentiation variable provided as a character (e.g., 'x'). If it is a single letter, it is treated as a spatial coordinate, and the output is built as a wrapper like $d_x(expr)$ (or $D_x(expr)$ in DC mode).
	sym	Differentiation variable provided as a symbolic object. In this case, the function constructs a symbolic derivative with respect to that variable, using a notation of the form $\frac{d(a)}{d(b)}$.
extraIndex	*char (index tag)*	Additional index label appended to the derivative wrapper as __extraIndex (e.g., $d_i^x(expr)$). It is also used by the "constant" heuristic: if expr does not contain this index, the derivative may be set to zero.
	char (mode flag)	Special flags: 'DC' switches the prefix to D_ and suppresses index tags in the final wrapper (while still propagating DC through recursion); 'NExpand' returns a non-expanded derivative wrapper without applying sum/product/power rules.
options	*char (mode flag)*	Optional mode flags: 'DC' activates DC mode; 'NExpand' returns the non-expanded derivative wrapper.
	char (default)	Default " (empty): no special mode is applied.

Outputs

Call	Output
fieldDerivative(x_i, 'i')	$d_i\,(x_i)$
fieldDerivative(x_i, x_j)	$\frac{d(x_i)}{d(x_j)}$
fieldDerivative(str2sym('A(x)'), 'i', 'x')	$d_i^x\,(A\,(x))$
fieldDerivative(str2sym('A(x)'), 'i', 'y')	0
fieldDerivative(x, 'i', 'DC')	$D_i\,(x)$
fieldDerivative(x_i*x_j, 'k', 'NExpand')	$d_k\,(x_i x_j)$

The argument `expr` can be a simple symbol or a more complex symbolic expression composed of sums, subtractions, multiplications (and powers). The function parses `expr` and applies the appropriate differentiation rules for each case (linearity, product rule, and power rule), returning a fully expanded result. If you prefer a compact, non-expanded output, enable `NExpand` in `options` (equivalently, it can be provided via `extraIndex`).

Common pitfalls / error cases

- **Two symbolic inputs (symbolic differentiation variable).**
  If you call `fieldDerivative(expr, expr1)` with `expr1` being a `sym`, then `extraIndex` and `options` should be kept empty. Mixing a symbolic `expr1` with additional tags/modes may lead to inconsistent formatting or incorrect symbolic wrappers.
- **Dependent variables such as `A(x)`.**
  To represent expressions with explicit dependencies (e.g., `A(x)`), it is strongly recommended to define them using `str2sym('A(x)')`. Writing `A(x)` directly may cause MATLAB to interpret `A` as an undefined function or produce unexpected symbolic behavior.
- **Very complex expressions and `NExpand`.**
  For highly nested or long expressions, `NExpand` may not preserve the intended internal structure with full precision.
- **Constant or fractional prefactors.**

In some cases, providing constant prefactors or fractional expressions (e.g., `1/2`, `(1/2)*expr`, or more general rational factors) may lead to incorrect results. This is mainly due to the string-based parsing and heuristic rules used to classify terms during differentiation.

- **Recommendation (workaround).**
  It is recommended to avoid explicit fractional forms by clearing denominators before calling `fieldDerivative`. A practical approach is to multiply the full expression by a convenient factor that removes the fractions. For example, if the entire expression contains a global factor $k = \frac{1}{2}$, you may multiply the expression by $2k$ to convert the prefactor into an integer factor, reducing the chance of mis-parsing.

1.2 solveFieldDerivative

```
function resultado = solveFieldDerivative(expr)
```

Description

`solveFieldDerivative` solve symbolic *functional-derivative-like* expressions written in the custom notation `d( ...)/d( ...)` and `d(d_i(...))/d(d_m(...))`. When the input matches any supported pattern, the function converts it into products of Kronecker deltas (e.g., `delta_i_j`) and Dirac deltas (represented as `dirac(x-y)`), including derivatives of Dirac deltas through calls to `fieldDerivative`.

Input arguments

This function takes a single input argument, `expr`, which represents the expression to be processed. The input should preferably be a symbolic expression produced by `fieldDerivative`, i.e., using the custom derivative wrappers ($\frac{d(\dots)}{d(\dots)}$, $\frac{d(d\_\mu(\dots))}{d(\dots)}$, etc.) that `solveFieldDerivative` is designed to recognize and rewrite.

Outputs

Input (pattern)	Output
d(x_i)/d(x_j)	δ_{ij}
d(d_k(x_i))/d(d_m(x_j))	$\delta_{km}\,\delta_{ij}$
d(A_i(x))/d(A_j(y))	$\delta_{ij}\,\delta\,(x-y)$
d(d_k(A_i(x)))/d(A_j(y))	$\delta_{ij}\,d_k(\delta\,(x-y))$
d(d_k(A_i(x)))/d(d_m(A_j(y)))	$\delta_{ij}\,\delta_{mk}\,\delta\,(x-y)$

Error cases / limitations

- **Fractional (inverse) expressions.** Although `fieldDerivative` can usually handle inverse inputs such as `1/x` correctly, `solveFieldDerivative` does not currently parse or rewrite expressions that contain explicit fractions (e.g., `1/d(...)` or general rational factors) in a reliable way. Therefore, it is recommended to avoid inserting fractional forms into expressions that will be processed by `solveFieldDerivative`.
- **Recommendation (workaround).** When possible, eliminate fractional terms before calling `solveFieldDerivative` by multiplying the entire expression by an appropriate factor to clear denominators (including both constant and non-constant fractions). This helps ensure that the internal pattern matching remains consistent and avoids incorrect or incomplete rewrites.

1.3 Complete workflow

To perform the full partial-derivative operation, the two functions must be called consecutively: first `fieldDerivative` to construct the partial-derivative expression, and then `solveFieldDerivative` to rewrite the resulting expression into Kronecker and Dirac deltas. This two-step design is intentional: it gives the user finer control over each stage of the computation and makes the intermediate symbolic forms explicit, so the full process can be inspected and verified at every step.

2 contraction

Description

`contraction` performs index contractions on a symbolic expression by repeatedly applying a set of string-based rewrite rules (implemented through regular expressions). The function is designed to simplify long expressions containing Kronecker deltas (e.g., $\delta_{i,j}$), metric tensors (e.g., $g_{\mu,\nu}$, $g^{\mu\,\nu}$), and combinations such as `F` contracted with two metrics.

Function signature

```
function exprFinal = contraction(expr, varing)
```

Input arguments

Table 1 Add caption

Name	Type	Meaning
expr	*sym*	Symbolic expression to be contracted. The function converts the expression to text and applies contraction rules (e.g., Kronecker deltas, metric tensors, and F–g–g patterns).
varing	*char*	(Optional) Contraction mode selector. If varing = '' (default), the function applies both delta-type contractions and metric/F contractions. If varing = 'd', only delta-type contractions are applied (metric/F contractions are skipped).

Outputs

Input	Output
A_i*delta_i_j	A_j
F_i_j*delta_i_k*delta_j_l	F_{kl}
F_mu_nu*g__mu__alpha*g__nu__beta	$F^{\alpha\beta}$
F__mu__nu*g_mu_alpha*g_nu_beta	$F_{\alpha\beta}$

Error cases / limitations

- **Contractions inside derivatives.**
  If a Kronecker delta must contract with an index that appears *inside* a derivative wrapper (e.g., `d_i(A_k)*delta_k_j`), the contraction may fail to be applied or may trigger an unintended rewrite.
- **Metric contractions with `F` require two metrics.**
  Metric contractions involving `F` are only performed when *two* metric tensors `g` are present to contract the same `F`. In other words, the implemented rule applies only when both indices of `F` are raised or lowered simultaneously, so that the resulting tensor has both indices *up* or both indices *down*. Mixed-index contractions (one up and one down) are not performed.
- **Limited to `F` and `g`.**
  At the moment, metric contractions are implemented specifically for the electromagnetic field tensor `F` and the metric tensor `g`. If other tensors are multiplied by metrics, the function will not attempt to contract them.

3 FInt

Description

`FInt` is a high-level wrapper designed to evaluate integrals of distribution-like symbolic expressions, in particular terms of the form

$$A(x)\,\partial_i\big(\delta(x-y)\big)\,dy,$$

represented in this code through the custom derivative wrappers (e.g., `d_i(...)` or `D_i(...)` acting on `dirac(x-y)`). While MATLAB's native `int` can usually handle integrals such as $A(x)\,\delta(x-y)\,dy$ without issues, it often does not return the desired result when derivatives act on the Dirac delta.

Function signature

```
function result = FInt(expr, var)
```

Input arguments

Name	Type	Meaning
expr	*sym*	Symbolic expression to be integrated. Typically includes Dirac deltas and/or their wrapped derivatives (e.g., d_i(dirac(x-y)) or D_i(dirac(x-y))).
var	*sym*	Integration variable. The integral is taken with respect to this variable.

Output

Input	Output
$A(y)\,\delta(x-y)\,dy$	$A(x)$
$A(y)\,d_i(\delta(x-y))\;dy$	$d_i(A(x))$

4 CalculateHessian

Description

`CalculateHessian` computes the Hessian matrix of a symbolic Lagrangian `L` with respect to a list of variables `varing`. It first builds and stores the first derivatives, then constructs all second derivatives to populate the Hessian. The routine uses the package's functional/tensor derivative workflow (`fieldDerivative` $\rightarrow$ `solveFieldDerivative`) and simplifies intermediate expressions via contractions and optional electromagnetic-tensor rewriting. Finally, it enforces a space–time split by mapping internal dummy indices (`gamma`, `sigma`) to `0`.

Function signature

```
function [Hessian, firstDerivatives] = CalculateHessian(L, varing)
```

Input arguments

Name	Type	Meaning
`L`	*sym*	Symbolic Lagrangian to differentiate.
`varing`	*sym array*	List (vector) of variables with respect to which the first and second partial derivatives are computed. Each element is internally indexed (e.g. with `lambda`, `rho`) to form the Hessian entries.

Outputs

Name	Type	Meaning
`Hessian`	*sym matrix*	Hessian matrix assembled from second derivatives of `L` with respect to the indexed variables built from `varing`. The output is converted from an internal cell array to a symbolic matrix via `cell2sym`.
`firstDerivatives`	*sym column vector*	Column vector containing the stored first derivatives (one per variable in `varing`), also converted from a cell array using `cell2sym`.

Important (index conventions)

Important: this routine uses the dummy indices `gamma`, `sigma`, `lambda`, and `rho` as internal defaults. Therefore, to avoid index clashes, the input expressions (and the variables inside `varing`) **must not** already contain any of these indices.

Notes / behavior

- The routine prints intermediate results using `disp`.
- After constructing each Hessian entry, the internal dummy indices `gamma` and `sigma` are mapped to `0`, enforcing a space–time split.
- The function attempts to convert each tensor entry to a matrix using `M(expr, 0:3)`; if this fails, it keeps the entry as a tensor expression.

5 CalculateCanonicalMomentum

Description

`CalculateCanonicalMomentum` takes a symbolic expression (typically the velocity-dependent part used to define canonical momenta) and produces a column vector containing the *temporal* and *spatial* canonical momentum components. Internally, it performs index contractions, simplifications and then enforces specific index substitutions (notably $\gamma \to 0$ and $\lambda \to 0$ / $\lambda \to i$). The routine also applies the package's standard normalization/simplification pipeline when needed.

Function signature

```
function canonicalMoments = CalculateCanonicalMomentum(expr)
```

Important: index conventions

This routine uses the dummy indices `gamma`, `rho`, `sigma`, and `lambda` as internal defaults. Therefore, to avoid index clashes, the input expression **must not** already contain any of these indices.

Input arguments

Name	Type	Meaning
`expr`	*sym*	Symbolic expression to be processed into canonical momentum components. The function contracts indices, may rewrite derivative patterns of the form `d_a(A_b)`, applies index substitutions ($\gamma \to 0$, then $\lambda \to 0$ and $\lambda \to i$), and runs the package's simplification pipeline as required.

Outputs

Name	Type	Meaning
`canonicalMoments`	*sym*	Column vector built as `[temporalMoment; spatialMoment]`. The first entry corresponds to the $\lambda = 0$ (temporal) component after processing; the second entry corresponds to the $\lambda = i$ (spatial) component (kept in indexed form).

Recommended workflow

`CalculateCanonicalMomentum` is intended to be used together with `CalculateHessian`. In particular, we recommend passing entries from the `firstDerivatives` output (returned by `CalculateHessian`) as inputs to `CalculateCanonicalMomentum`, so the canonical momenta are constructed from already-computed first partial derivatives of the Lagrangian.

6 `CalculateHamiltonian`

Description

`CalculateHamiltonian` is a *pre-processing* utility for a symbolic Lagrangian. Its main procedure is to explicitly separate spatial and temporal parts, this step **breaks manifest covariance** of the original (fully covariant) equations. This is intentional: the Hamiltonian/constraint workflow in this package is formulated after a $3 + 1$ decomposition, where time is treated differently from space.
Note: despite its name, this function does *not* compute the canonical momenta nor the Hamiltonian by itself; it prepares L so that the subsequent Hamiltonian workflow is more consistent.

Function signature

```
function L_final = CalculateHamiltonian(L)
```

Input arguments

Name	Type	Meaning
`L`	*sym*	Symbolic Lagrangian expression to be standardized and simplified. The function inspects index tags in `char(L)`, applies `SpatialAndTemporal(L)`, and then applies `CondicionesIniciales` to the result.

Outputs

Name	Type	Meaning
`L_final`	*sym*	Processed Lagrangian after applying `SpatialAndTemporal` and `CondicionesIniciales`.

7 PoissonBrackets

Description

`PoissonBrackets` evaluates Poisson brackets for fields by explicitly applying the standard field-theory definition

$$\{F,G\} = \int d^n z \left[\frac{\delta F}{\delta q(z)} \frac{\delta G}{\delta p(z)} - \frac{\delta F}{\delta p(z)} \frac{\delta G}{\delta q(z)} \right],$$

where $q(z)$ and $p(z)$ denote canonical field variables (and their conjugate momenta), and $\delta/\delta(\cdot)$ are functional derivatives.

This function implements this workflow using the previously defined functions.

Function signature

```
function result = PoissonBrackets(F, G, var, distVars, varargin)
```

Input arguments

Name	Type	Meaning
F	*sym*	First symbolic functional/field expression used in the Poisson bracket F,G.
G	*sym*	Second symbolic functional/field expression used in the Poisson bracket F,G.
var	*sym / char / list*	Additional variable(s) forwarded to internal routines as the canonical position list (used in solvePB_ind to iterate over canonical pairs).
distVars	*cell array of char*	Distributive variables (typically 'x','y'). If omitted or empty, defaults to 'x','y'.
	char (special)	If distVars = 'H', it is treated as empty and the flag 'H' is enabled (same behavior as passing 'H' in varargin).
	default	Default 'x','y' when not provided.
varargin	*char (flag)*	Optional flags. If any element equals 'H', the function applies a final integral FInt(..., distVars2) after resolving the Poisson bracket.
	empty (default)	No final integral is applied.

Output

Name	Type	Meaning
`result`	*sym*	Final symbolic expression for $\{F,G\}$ after: explicit PB expansion, initial-condition reduction, PB resolution (including wrapped derivatives `d_` / `D_`), optional final integration if `'H'` is enabled, index contractions, and derivative-wrapper simplification.

8 Dirac matrix

Important note. In this package, derivatives acting on Dirac deltas are *explicitly separated* and represented as *multiplicative operators*. Concretely, expressions of the form

$$d_{ix}\big(d_{jy}(\delta(x-y))\big) \quad \longrightarrow \quad d_{ix}\, d_{jy}\, \delta(x-y),$$

i.e.

`d_i__x(d_j__y(delta(x-y)))` $\mapsto$ `d_i__x*d_j__y*delta(x-y).`

This convention is required to keep the kernel-algebra consistent when constructing and inverting the constraint matrix in field theory.

In particular:

- When building the standard $\mathbf{W}$ matrix with `CalculateW`, any "loose" derivative factors (e.g. `d_i__x`, `D_i__x`) that appear multiplying $\delta(\cdot)$ should be understood as derivatives *acting on the Dirac delta kernel.*
- When computing $\mathbf{W}^{-1}$ with `CalculateWInverse`, those same loose derivative factors play the role of the nonlocal inverse operators (schematically ∇^{-2}-type structures) that typically arise in the inversion of distribution-valued constraint matrices in field theories.

8.1 `CalculateW`

Description

`CalculateW` constructs the *constraint (Poisson) matrix* W from an input vector of constraints (or generic expressions). Each entry is computed as a field Poisson bracket between components of the vector,

$$W_{ij}(x,y) = \{\phi_i(x), \phi_j(y)\},$$

using the package routine `PoissonBrackets`. After computing each off-diagonal element, the result is post-processed with `separateDirac` to factor derivative-wrapped delta distributions into a multiplicative form, and finally the full matrix is `collect`-ed with respect to $\delta(x-y)$ (built from `distVars`) to standardize its structure.
Optimization: the diagonal is set to zero explicitly ($W_{ii} = 0$) to avoid unnecessary work, consistent with $\{F, F\} = 0$.

Function signature

```
function W = CalculateW(vector, var, distVars)
```

Input arguments

Name	Type	Meaning
`vector`	*sym (vector)*	Vector of symbolic expressions (typically constraints ϕ_i). The function builds a square matrix with entries {`vector(i)`, `vector(j)`}.
`var`	*sym / char / list*	Extra variable(s) forwarded to `PoissonBrackets` as the canonical position list (used internally to iterate over canonical pairs).
`distVars`	*cell array of char*	Distributive variables used in the field Poisson brackets and in the final $\delta(\cdot)$ collection. Typical choice is {`'x'`,`'y'`}. If omitted or empty, defaults to {`'x'`,`'y'`}.

Output

Name	Type	Meaning
`W`	*sym (matrix)*	Symbolic matrix whose entries are Poisson brackets between the components of `vector`, post-processed by `separateDirac` and collected with respect to δ(`distVars`{1} − `distVars`{2}). The diagonal is forced to zero.

8.2 `CalculateWInverse`

Description

`CalculateWInverse` computes a symbolic *kernel inverse* of a constraint matrix W whose entries typically contain distributional factors such as $\delta(x-y)$.

$$\int d^3z\, W_{ij}(x,z)\, W^{-1}_{jk}(z,y) \;=\; \delta_{ik}\, \delta^{(3)}(x-y). \tag{8.1}$$

Note: This routine is designed for the package's distribution-aware workflow (Dirac kernels + indexed derivative operators). It is not a generic symbolic matrix inverse.

Function signature

```
function wMatrixFinal = CalculateWInverse(inputMatrix, distVars)
```

Input arguments

Name	Type	Meaning
`inputMatrix`	*sym (matrix)*	Symbolic matrix W to be inverted. It must contain at least one entry with a factor of the form `dirac(var1-var2)` so that the routine can identify the kernel variables. The matrix is expanded first via `expand`.
`distVars`	*cell array of char*	Target distributive variables for the output inverse kernel, typically something like {`u,w`}. This pair must share at least one variable with the detected `dirac(var1-var2)` inside `inputMatrix`. If omitted or empty, defaults to {`var2,'v'`} where `var2` is extracted from `dirac(var1-var2)`.

Output

Name	Type	Meaning
`wMatrixFinal`	*sym (matrix)*	Symbolic inverse-kernel matrix W^{-1} consistent with the detected Dirac structure. The result is obtained by solving a linear system for the unknown matrix w such that $W \cdot w$ reproduces an identity kernel proportional to $\delta(\texttt{var1} - \texttt{var2})\,\delta(\texttt{distVars}\{1\} - \texttt{distVars}\{2\})$.

9 DiracBrackets

Description

`DiracBrackets` computes the Dirac bracket between two field expressions by applying the standard definition

$$\{F, G\}_D = \{F, G\} - \{F, \phi_a\}\,(C^{-1})^{ab}\,\{\phi_b, G\}, \tag{9.1}$$

and, in the field/distributional setting,

$$\{F, G\}_D = \{F, G\} - \int du\,dw\,\{F, \phi_a(u)\}\,(C^{-1})^{ab}(u, w)\,\{\phi_b(w), G\}, \tag{9.2}$$

where ϕ_a are the (second-class) constraints and $(C^{-1})^{ab}$ is the inverse of the constraint matrix $C_{ab}(u, w) = \{\phi_a(u), \phi_b(w)\}$.

Function signature

```
function DiracBracket = DiracBrackets(q_i, P_j, phi, var, distVars, varargin)
```

Input arguments

Name	Type	Meaning
`q_i`	*sym*	First argument of the Dirac bracket (typically a canonical field variable or a functional built from it).
`P_j`	*sym*	Second argument of the Dirac bracket (typically the conjugate momentum field, or a functional built from it).
`phi`	*sym vector*	Vector of constraints ϕ_a used to build the constraint matrix $C_{ab} = \{\phi_a, \phi_b\}$ and its inverse.
`var`	*sym / char / list*	Additional variable(s) forwarded to internal routines (consistent with the $\{\cdot, \cdot\}$ evaluation in `PoissonBrackets`).
`distVars`	*cell array of char*	Distributive variables for the basic bracket (default `{'x','y'}`). Internally, dummy integration variables `'u'` and `'w'` are introduced for the correction term.
	default	Default `{'x','y'}` when not provided.
`varargin`	*any*	Extra arguments forwarded to `solveDB` and then to `PoissonBrackets` calls inside the Dirac-bracket correction term (e.g., flags such as `'H'` if your workflow uses them).

Output

Name	Type	Meaning
`DiracBracket`	*sym*	Symbolic result of the Dirac bracket $\{q_i, P_j\}_D$ obtained as $\{q_i, P_j\} - \int du\, dw\, \{q_i, \phi_a(u)\}(C^{-1})^{ab}(u, w)\{\phi_b(w), P_j\}$.

Notation and conventions (important)

For consistent handling of conjugate momenta throughout the toolbox (especially in `PoissonBrackets` and `DiracBrackets`), the conjugate momentum field should be represented using the reserved symbolic name

$$\pi \quad \equiv \quad \texttt{sym('pi')}.$$

Using `pi` ensures that the internal canonical-pair logic (i.e., the (q, π) pairing used when constructing functional derivatives) behaves as intended.

Consequently, if you want to compute a Poisson bracket between a coordinate field and its conjugate momentum, you should write it explicitly using π with indices; for instance,

$$\{x_j, \pi_i\} \quad \text{instead of} \quad \{x_j, P_i\} \text{ or any other momentum label.}$$

10 FunctionField

Description

`FunctionField` is a helper routine that rewrites indexed symbolic objects as *field-valued functions* of a chosen distributive variable. In practice, it converts occurrences of indexed symbols such as `A_i` (or `A__i` depending on your index convention) into the functional form `A_i(value)`, and it also updates wrapped derivatives accordingly, e.g.

$$d_k\big(A_i\big) \longrightarrow d_k\big(A_i(value)\big).$$

Additionally, it always processes the electromagnetic tensor components `F_i_j` and their derivatives, ensuring they are consistently treated as field functions `F_i_j(value)` when present.

This function is mainly used as a *pre-processing step* before functional derivatives, Poisson brackets, and Dirac brackets, so that all canonical fields explicitly depend on the integration/distribution variables.

Function signature

```
function result = FunctionField(expr, var, value)
```

Input arguments

Name	Type	Meaning
`expr`	*sym*	Symbolic expression to be rewritten in field-functional form. The routine searches for indexed occurrences of `var` and (always) `F_i_j`, and updates them to include explicit dependence on `value`.
`var`	*sym*	Base symbolic name of the field to functionalize (e.g. `A`, `pi`). The function targets indexed instances like `var_i` / `var__i`.
`value`	*sym*	Distributive/position variable inserted as the functional argument (e.g. `x`, `y`, `z`). This input must be symbolic.

Output

Name	Type	Meaning
`result`	*sym*	Expression where indexed symbols matching `var` are rewritten as `var_i(value)` (or updated to use `value` if already functional), and where `F_i_j` is similarly enforced as `F_i_j(value)`. Wrapped derivatives of the form `d_k(var_i)` or `d_k(F_i_j)` are updated consistently.

11 `changeFunctionField`

Description

`changeFunctionField` is a string-based utility that *renames the distributive variable* used inside field-like functional dependencies across a symbolic expression. Concretely, it performs two coordinated replacements:

- It rewrites derivative wrappers of the form `d_i__x(···)` (and also `D_i__x`, including multiple underscores) into `d_i__newVal(···)`.
- It rewrites simple function-like dependencies `A(x)` → `A(newVal)` for indexed field symbols and similar objects, while avoiding common non-field functions (e.g. `dirac`, `sin`, `exp`, etc.).

This routine is useful when you want to move an expression from one integration/distribution variable to another (e.g. from x to y) while keeping the internal representation consistent for later routines (functional derivatives, Poisson brackets, Dirac brackets, etc.).

Function signature

```
function out = changeFunctionField(expr, newVal)
```

Input arguments

Name	Type	Meaning
`expr`	*sym / char*	Symbolic expression (or something convertible to `sym`) containing field-like dependencies such as `A(x)` and derivative wrappers such as `d_i__x(...)` or `D_i__x(...)`.
`newVal`	*sym / char / string*	New distributive variable to be injected into dependencies and derivative wrappers (e.g. `'y'` or `sym('y')`). Internally, whitespace is removed from `char(newVal)`.

Output

Name	Type	Meaning
`out`	*sym*	Expression obtained after (i) replacing derivative wrapper subscripts `__oldVar` by `__newVal` in operators `d_` and `D_`, and (ii) replacing simple functional calls `Name(oldVar)` by `Name(newVal)` for field-like symbols not present in the internal blacklist.

Notes

- **Derivative wrappers.** The derivative-operator rewrite targets patterns like `d_i__x(` and `D_i__x(`, including variants with multiple underscores after `d`/`D`. This makes nested derivatives consistent, e.g.

$$d_i{}^x\big(d_j{}^x(J_0(x))\big) \;\mapsto\; d_i{}^y\big(d_j{}^y(J_0(y))\big)\,.$$

- **Function calls.** Only *simple* one-symbol arguments are rewritten (e.g. `A(x)`). This is intentional to avoid altering objects such as `dirac(x-y)` or other composite arguments.
- **Blacklist.** The routine skips common non-field functions (e.g. `dirac`, `sin`, `cos`, `exp`, `log`, `sqrt`, `diff`, `subs`, etc.). You can edit the internal `blacklist` to match your project.
- **String-based behavior.** Since this function relies on regex over `char(expr)`, it assumes the project naming conventions used throughout the toolbox (e.g. `d_i__x(...)`, indexed field names with underscores).

12 Simplification Fuctions

12.1 `transposeAs`

Description

`transposeAs` applies an *index-transposition* rule to tensor-like symbols that carry **two indices** (e.g., `g__mu__nu`, `F_mu_nu`). The function automatically detects which indexed symbols appear in the input expression and then transposes their indices according to a built-in classification:

- **Symmetric symbols** (default: `g`, `G`): swapping indices leaves the symbol unchanged.
- **Antisymmetric symbols** (default: `F`, `theta`): swapping indices introduces a minus sign.

It works term-by-term, handling sums and products, and can simplify expressions by identifying terms that become identical (or opposite) after transposition.

Main purpose

A key goal of `transposeAs` is to **detect and simplify index-equivalent terms** that the native MATLAB symbolic engine typically *cannot* simplify, because it treats different index orderings as unrelated symbols.

For example, MATLAB will not simplify

$$F_{ij} - F_{ji},$$

because it does not automatically use the antisymmetry property of F. In contrast, `transposeAs` recognizes that F is declared as **antisymmetric**, so

$$F_{ji} = -F_{ij},$$

and therefore it simplifies the expression to

$$F_{ij} - F_{ji} \;=\; F_{ij} - (-F_{ij}) \;=\; 2F_{ij}.$$

This behavior is especially useful when intermediate steps generate many terms with permuted indices, where manual index transposition would otherwise be required to reduce the expression.

Function signature

```matlab
function exprFinal = transposeAs(expr)
```

Input arguments

Name	Type	Meaning
`expr`	*sym*	Symbolic expression containing tensor-like symbols with exactly two indices, written with MATLAB-style index separators _ or __.

Output

Name	Type	Meaning
`exprFinal`	*sym*	Expression after transposing the detected indexed symbols according to symmetry/antisymmetry rules, with possible simplifications from term combination.

12.2 `simplifyS`

Description

`simplifyS` is a small helper that performs *iterative, chunked simplification* of a symbolic expression. Instead of simplifying the whole expression at once, it:

This approach can sometimes avoid overly aggressive or expensive global simplifications, while still reducing long sums progressively.

Main purpose

`simplifyS` is designed to simplify *large* or *structurally complicated* expressions more robustly than a single global call to MATLAB's native `simplify`.

Although `simplify` is generally very strong, it can fail to perform a desired reduction when the expression contains additional terms that are unrelated to the part that *can* be simplified. For example, a structure like

$$a^2 + 2ab + b^2$$

may simplify cleanly to $(a + b)^2$, but if the same block appears embedded inside a longer sum such as

$$a^2 + 2ab + b^2 + c,$$

then a single global `simplify` may leave the expression unchanged (depending on MATLAB's internal heuristics).

To address this, `simplifyS` performs simplification *incrementally* by splitting the expression into additive terms and simplifying them in small blocks, rebuilding the expression and repeating until no further changes are detected. This makes the simplification step more robust in workflows where expressions grow large and contain independent sub-blocks that should simplify locally.

Function signature

```matlab
function simplifiedExpr = simplifyS(expr)
```

Input arguments

Name	Type	Meaning
`expr`	*sym*	Symbolic expression. The function only proceeds if the expression contains additive structure ("+" or "-").

Output

Name	Type	Meaning
simplifiedExpr	*sym*	The final expression after repeated pairwise simplification of additive terms, stopping when no further changes are detected.

Notes

- If the expression has **no explicit "+" or "-"** in its string form, the function exits early and returns the original input.
- The simplification is done in **pairs of terms**. This is heuristic: it may simplify less than a full global `simplify`, but it can also be more stable for large expressions.
- The stopping criterion is exact symbolic equality (`newExpr == simplifiedExpr`). In some cases, mathematically equivalent expressions that differ syntactically may not be detected as "no change".

12.3 SimplifyFieldDerivative

Description

`SimplifyFieldDerivative` is a post-processing routine that attempts to *compress expanded product-rule results* produced by `fieldDerivative`. In practice, when an expression contains multiple terms of the form

$$M_1\, d_i(\,\cdot\,) \;+\; M_2\, d_i(\,\cdot\,),$$

this function searches for compatible pairs and rewrites them into a single derivative term, factoring out common multiplicative pieces whenever possible.

Function signature

```
function expr_simplificada = SimplifyFieldDerivative(expr)
```

Input arguments

Name	Type	Meaning
expr	*sym*	Symbolic expression (typically the output of `fieldDerivative` and/or `solveFieldDerivative`) that may contain expanded product-rule terms.

Output

Name	Type	Meaning
expr_simplificada	*sym*	The simplified expression after repeatedly combining compatible derivative terms.

Notes

- This routine is pattern-based (regex + symbolic checks). It is designed for the derivative syntax used throughout the toolbox, e.g. `d_i(...)` and optionally `d_i__x(...)`.
- It only combines terms when it can verify a consistent factor structure using `isequal`. If expressions are algebraically equivalent but not syntactically identical, they may not be merged.
- The simplification is performed iteratively: after each successful merge, the function re-parses the updated expression and tries again.

12.4 `indexChange`

Description

`indexChange` performs a **literal index renaming** inside an expression by replacing occurrences of indices written in the package notation as `_oldIdx` with `_newIdx`. This is mainly used to enforce a specific index convention (e.g., setting a dummy index to 0, or avoiding index clashes) before continuing with contractions or other symbolic routines.

Function signature

```
function exprFinal = indexChange(expr, varargin)
```

Inputs

Name	Type	Description
`expr`	`sym`	Expression where index replacements will be applied. If `expr` is symbolic, the routine converts it to a string internally, performs replacements, and converts back to `sym`.
`varargin`	pairs of `char`	Index replacement pairs `(oldIdx, newIdx)`. Must be provided as an even-length list: $$(\texttt{oldIdx}_1, \texttt{newIdx}_1,\ \texttt{oldIdx}_2, \texttt{newIdx}_2,\ \ldots)$$ Each pair triggers a literal replacement `_oldIdx` $\mapsto$ `_newIdx`.

Output

Name	Type	Description
`exprFinal`	`sym`	Symbolic expression after all requested index substitutions have been applied.

Example

$$\texttt{indexChange(F\_gamma\_lambda, 'gamma','0', 'lambda','i')} \quad \Rightarrow \quad F_{0i}.$$

Notes

- This is a **string-level** replacement. It does not check tensor meaning, index position, or whether the replacement produces invalid expressions.
- It only replaces indices that appear exactly as `_oldIdx`. If an index appears in another context (e.g., inside text, function names, or without the underscore convention), it will not be modified.
- Be careful with short indices (e.g., `i`) if they could accidentally match substrings in unintended places (this depends on how your expressions are written).

12.5 `T_matrix`

Description

`T_matrix` is a helper routine that **reorders (transposes) indexed objects inside products** so that contracted index pairs appear in a consistent orientation. It is designed for expressions written as products of rank-2 tensors (e.g., `g__mu__nu`, `F_i_j`, `theta_i_j`) and uses the symmetry/antisymmetry rules (via `transposeAs`) to decide whether a transpose introduces a minus sign.

This function reorders the indices of the tensors `theta`, `F`, and `g` so that repeated indices appear next to each other ("touching"). This enforces a consistent canonical pattern for products and improves the reliability and speed of subsequent simplification steps. For example, a product like `F_i_j*theta__i__j` is rewritten into an equivalent canonical form -F_i_j*theta__j__i so that later routines can recognize and simplify it consistently.

Function signature

```
function exprFinal = T_matrix(expr)
```

Inputs

- expr (sym): symbolic expression to be processed.

Output

- exprFinal (sym): expression where relevant factors may have been transposed to improve index alignment.

Notes / limitations

- This routine works primarily at the **string/pattern** level (splitting by * and matching index patterns), so it assumes expressions follow the package naming convention for indices.
- It focuses on **two-index** objects. Scalars, vectors, or higher-rank tensors may be left unchanged.
- Transposition is only applied when the routine detects that it helps align shared indices between factors; isolated indexed factors are deliberately excluded from the transpose procedure.

12.6 SimplifyContractedExpr

Description

SimplifyContractedExpr is a post-processing routine designed to simplify expressions that are already written as an *additive sum* of contracted products. The main idea is to **group and merge like terms**: for each additive term it separates the purely numeric coefficient from the symbolic core, builds a *signature* (regex fingerprint) for the core product, and then **sums the coefficients** of terms that share the same signature. Terms whose total coefficient becomes zero are removed.
This routine is especially useful after repeated applications of contraction/transposition steps, where many identical contracted products may appear with different numeric prefactors.

Importance. This is one of the *most important* routines in the package for simplification, because it is particularly effective when working with expressions containing **dummy (summed) indices**. After contractions/transpositions, the same tensor product may appear multiple times with different prefactors; SimplifyContractedExpr reliably detects these repeats (up to relabelings captured by its signature logic) and merges them into a single canonical contribution.

Function signature

```
function simplifiedExpr = SimplifyContractedExpr(expr)
```

Input arguments

Name	Type	Meaning
expr	*sym*	Symbolic expression expected to contain a sum of terms. The function expands expr to normalize subtraction, extracts additive terms via children(expr), and groups them by a structure-based signature that ignores numeric prefactors.

Outputs

Name	Type	Meaning
`simplifiedExpr`	*sym*	Simplified symbolic sum where terms with identical contracted structure have been merged by summing their numeric coefficients, and any zero-coefficient groups have been removed. If `expr` is not a sum (not of type `plus`), it is returned unchanged.

12.7 `CondicionesIniciales`

Description

`CondicionesIniciales` is a rule-based simplification routine that enforces a collection of *predefined algebraic/physical assumptions* ("initial conditions") on symbolic expressions. In practice, it removes terms that must vanish due to antisymmetry (e.g., repeated indices in antisymmetric tensors), truncation rules for `theta`, and several index-selection rules involving the metric `g` and Kronecker deltas `delta`.
The function works mostly by scanning the expression for specific patterns (via `symvar` + `regexprep`) and replacing them with 0 or 1 when the corresponding condition applies. It is typically called after expanding or computing Poisson/Dirac brackets to quickly eliminate forbidden index configurations and reduce clutter.
Important: this routine is designed specifically around the symbols and index conventions used in this package, in particular the electromagnetic tensor `F`, the noncommutative tensor `theta`, and the metric tensors `g`/`G`. If your expression uses different tensor names or different structural conventions, these pattern rules may not apply (or may apply incorrectly), so the function may *not* simplify such expressions as expected.

Function signature

```
function expr_simplificada = CondicionesIniciales(expr)
```

Properties enforced by `CondicionesIniciales`

- **Antisymmetry zeroing (built-in bases).** For antisymmetric tensors (by default `F` and `theta`), any component with equal indices is set to zero:

$$F_{\mu\mu} = 0, \qquad \theta_{\mu\mu} = 0.$$

- **Truncation in the noncommutative tensor `theta`.** Products/powers involving multiple factors of `theta` are truncated:
$$\theta\,\theta = 0, \qquad \theta^2 = 0, \qquad \theta^3 = 0,$$
implemented by pattern rules (independent of the explicit indices).
- **Temporal-index suppression for `theta`.** Any `theta` component carrying a time index is set to zero:

$$\theta_{0i} = 0, \qquad \theta_{i0} = 0.$$

- **Mixed time–space metric components vanish.** The routine enforces the block-diagonal $3+1$ convention for the metric by setting mixed components to zero:

$$g_{0i} = 0, \qquad g_{i0} = 0.$$

- **Off-diagonal purely spatial metric components vanish.** For numeric spatial indices, it removes off-diagonal terms:
$$g_{ij} = 0 \quad \text{for } i \neq j,\ i,j \in \{1,\dots,9\}.$$
- **Selected covariant/contravariant metric contraction shortcuts.** Specific patterns of metric contractions are replaced by 1 (as hard-coded identities in the package), e.g.

$$g^{ii}\,g_{ii} \;\to\; 1 \qquad (i \in \{1,\dots,9\}).$$

- **Delta-product exclusion rules involving time vs. spatial indices.** Certain products of Kronecker deltas with incompatible temporal/spatial matching are forced to zero (pattern-based), e.g.

$$\delta_{0a}\,\delta_{ba} \;\to\; 0 \quad \text{for alphanumeric } a, b,$$

  and related variants.
- **Poisson-bracket derivative cancellation (special hard-coded pattern).** A specific derivative-of-Poisson-bracket structure is set to zero, of the schematic form

$$d_i\big(\mathrm{PB}(F_{kl}(x), \pi_i(y))\big) \;=\; 0,$$

  matching the package naming conventions.
- **Delta squared simplification.** Any squared Kronecker delta factor is simplified as

$$\delta^2 \;\to\; 1,$$

  applied via pattern matching (not index-sensitive).

13 `NormalizeContractedIndices`

Description

`NormalizeContractedIndices` is a *canonicalization* utility designed to improve symbolic simplification in expressions containing **dummy (contracted) indices**. Its main goal is to systematically **rename contracted indices** so that equivalent terms differing only by dummy-index labels are mapped to the *same* representative form. This makes subsequent routines (e.g., grouping/collection and pattern-based simplifiers) much more effective.

The routine works *term-by-term*: it first expands the expression, then splits it into additive terms. For each term it detects index tokens written in the package convention (i.e., `_i`, `__mu`, `_0`, etc.), counts their occurrences, and classifies indices as

- **free indices**: appear exactly once in the term,
- **contracted indices**: appear two or more times in the term.

It then builds a *global* set of free indices across all terms, and excludes them from renaming to avoid clashes. Finally, each term's contracted Latin/Greek indices are renamed in order of first appearance to canonical pools (Latin: `i,j,k,...` and Greek: `mu,nu,rho,...`), using a **collision-safe** two-step renaming (old → temporary tag → new).

Function signature

```
function exprOut = NormalizeContractedIndices(expr)
```

Input arguments

Name	Type	Meaning
`expr`	*sym / char / string*	Symbolic expression (or string convertible with `str2sym`) written using the package index convention `_` / `__`. The function expands `expr`, splits it into additive terms, and renames *only* contracted indices. Numeric indices (e.g., `_0`, `_1`) are never renamed.

Outputs

Name	Type	Meaning
`exprOut`	*sym*	Expanded symbolic expression where dummy (contracted) indices have been renamed to a canonical set, consistently across terms, without touching globally free indices.