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

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 `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

```matlab
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.

5 DiracBrackets

Description

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

$$\{F, G\}_D = \{F, G\} - \{F, \phi_a\}\,(C^{-1})^{ab}\,\{\phi_b, G\}, \tag{5.1}$$

and, in the field/distributional setting,

$$\{F, G\}_D = \{F, G\} - \int du\,dw\,\{F, \phi_a(u)\}\,(C^{-1})^{ab}(u, w)\,\{\phi_b(w), G\}, \tag{5.2}$$

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

```matlab
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,

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