

Forces between magnets and multipole arrays of magnets: A Matlab implementation

Will Robertson

July 2, 2018

Abstract

This is the user guide and documented implementation of a set of Matlab functions for calculating the forces (and stiffnesses) between cuboid permanent magnets and between multipole arrays of the same.

This document is still evolving. The documentation for the source code, especially, is rather unclear/non-existent at present. The user guide, however, should contain the bulk of the information needed to use this code.

Contents

I	User guide	2
1	Defining magnets and coils	2
2	Forces	3
2.1	Forces between magnets	3
2.2	Forces between multipole arrays of magnets	4
3	Meta-information	6
II	Typeset code / implementation	8
4	The magnetdefine() function	8
4.1	grade2magn	9
4.1.1	grade2magn	9
5	The magnetforces() function	10
5.1	Calculate for each displacement	14
5.2	Helpers	32
6	The multipoleforces function	33

Part I

User guide

(See Section 3 for installation instructions.)

1 Defining magnets and coils

```
magnet = magnetdefine('type',T,key1,val1,...)
```

'type' The possible options for T are: 'cuboid', 'cylinder', 'coil'. If 'type',T is omitted it will be inferred by the number of elements used to specify the dimensions of the magnets/coils.

Cuboid magnets For cuboid magnets, the following should be specified:

'dim' A (3×1) vector of the side-lengths of the magnet.

'grade' The 'grade' of the magnet as a string such as 'N42'.

'magdir' A vector representing the direction of the magnetisation. This may be either a (3×1) vector in cartesian coordinates or a (2×1) vector in spherical coordinates.

Instead of specifying a magnet grade, you may explicitly input the remanence magnetisation of the magnet direction with

'magn' The remanence magnetisation of the magnet in Tesla.

Note that when not specified, the **magn** value B_r is calculated from the magnet grade N using $B_r = 2\sqrt{N/100}$.

In cartesian coordinates, the 'magdir' vector is interpreted as a unit vector; it is only used to calculate the direction of the magnetisation. In other words, writing $[1;0;0]$ is the same as $[2;0;0]$, and so on. In spherical coordinates (θ, ϕ) , θ is the vertical projection of the angle around the x - y plane ($\theta = 0$ coincident with the x -axis), and ϕ is the angle from the x - y plane towards the z -axis. In other words, the following unit vectors are equivalent:

$$(1, 0, 0)_{\text{cartesian}} \equiv (0, 0)_{\text{spherical}}$$

$$(0, 1, 0)_{\text{cartesian}} \equiv (90, 0)_{\text{spherical}}$$

$$(0, 0, 1)_{\text{cartesian}} \equiv (0, 90)_{\text{spherical}}$$

N.B. θ and ϕ must be input in degrees, not radians. This seemingly odd decision was made in order to calculate quantities such as $\cos(\pi/2) = 0$ exactly rather than to machine precision.¹

If you are calculating the torque on the second magnet, then it is assumed that the centre of rotation is at the centroid of the second magnet. If this is not the case, the centre of rotation of the second magnet can be specified with

'lever' A (3×1) vector of the centre of rotation (or $(3 \times D)$ if necessary; see D below).

Cylindrical magnets/coils If the dimension of the magnet ('dim') only has two elements, or the 'type' is 'cylinder', the forces are calculated between two cylindrical magnets.

While coaxial and 'eccentric' geometries can be calculated, the latter is around 50 times slower; you may want to benchmark your solutions to ensure speed is acceptable. (In the not-too-near-field, you can sometimes approximate a cylindrical magnet by a cuboid magnet with equal depth and equal face area.)

¹Try for example comparing the logical comparisons `cosd(90)==0` versus `cos(pi)==0`.

'dim' A (2×1) vector containing, respectively, the magnet radius and length.
'dir' Alignment direction of the cylindrical magnets; 'x' or 'y' or 'z' (default). E.g., for an alignment direction of 'z', the faces of the cylinder will be oriented in the x - y plane.

A 'thin' magnetic coil can be modelled in the same way as a magnet, above; instead of specifying a magnetisation, however, use the following:

'turns' A scalar representing the number of axial turns of the coil.
'current' Scalar coil current flowing CCW-from-top.

A 'thick' magnetic coil contains multiple windings in the radial direction and requires further specification. The complete list of variables to describe a thick coil, which requires 'type' to be 'coil' are

'dim' A (3×1) vector containing, respectively, the inner coil radius, the outer coil radius, and the coil length.
'turns' A (2×1) containing, resp., the number of radial turns and the number of axial turns of the coil.
'current' Scalar coil current flowing CCW-from-top.

Again, only coaxial displacements and forces can be investigated at this stage.

2 Forces

2.1 Forces between magnets

The function `magnetforces` is used to calculate both forces and stiffnesses between magnets. The syntax is as follows:

```
forces = magnetforces(magnet_fixed, magnet_float, displ);
... = magnetforces( ... , 'force');
... = magnetforces( ... , 'stiffness');
... = magnetforces( ... , 'torque');
... = magnetforces( ... , 'x');
... = magnetforces( ... , 'y');
... = magnetforces( ... , 'z');
```

`magnetforces` takes three mandatory inputs to specify 'fixed' and 'floating' magnets and the displacement between them. Optional arguments appended indicate whether to calculate force and/or torque and/or stiffness and whether to calculate components in x - and/or y - and/or z - components respectively. The force² is calculated as that imposed on the second magnet; for this reason, I often call the first magnet the 'fixed' magnet and the second 'floating'.

Outputs You must match up the output arguments according to the requested calculations. For example, when only calculating torque, the syntax is

```
T = magnetforces(magnet_fixed, magnet_float, displ, 'torque');
```

Similarly, when calculating all three of force/stiffness/torque, write

```
[F, S, T] = magnetforces(magnet_fixed, magnet_float, displ, ...
    'force', 'stiffness', 'torque');
```

The ordering of 'force', 'stiffness', 'torque' affects the order of the output arguments. As shown in the original example, if no calculation type is requested then the forces only are calculated.

²From now I will omit most mention of calculating torques and stiffnesses; assume whenever I say 'force' I mean 'force and/or stiffness and/or torque'

Displacement inputs The third mandatory input is `displ`, which is a matrix of displacement vectors between the two magnets. `displ` should be a $(3 \times D)$ matrix, where D is the number of displacements over which to calculate the forces. The size of `displ` dictates the size of the output force matrix; `forces` (etc.) will be also of size $(3 \times D)$.

Example Using `magnetforces` is rather simple. A magnet is set up as a simple structure like

```
magnet_fixed = magnetdefine(...
    'dim'      , [0.02 0.012 0.006], ...
    'magn'     , 0.38, ...
    'magdir'   , [0 0 1] ...
);
```

with something similar for `magnet_float`. The displacement matrix is then built up as a list of (3×1) displacement vectors, such as

```
displ = [0; 0; 1]*linspace(0.01,0.03);
```

And that's about it. For a complete example, see `'examples/magnetforces_example.m'`.

2.2 Forces between multipole arrays of magnets

Because multipole arrays of magnets are more complex structures than single magnets, calculating the forces between them requires more setup as well. The syntax for calculating forces between multipole arrays follows the same style as for single magnets:

```
forces = multipoleforces(array_fixed, array_float, displ);
stiffnesses = multipoleforces( ... , 'stiffness');
[f s] = multipoleforces( ... , 'force', 'stiffness');
... = multipoleforces( ... , 'x');
... = multipoleforces( ... , 'y');
... = multipoleforces( ... , 'z');
```

Because multipole arrays can be defined in various ways, there are several overlapping methods for specifying the structures defining an array. Please excuse a certain amount of dryness in the information to follow; more inspiration for better documentation will come with feedback from those reading this document!

Linear Halbach arrays A minimal set of variables to define a linear multipole array are:

array.type Use `'linear'` to specify an array of this type.

array.align One of `'x'`, `'y'`, or `'z'` to specify an alignment axis along which successive magnets are placed.

array.face One of `'+x'`, `'+y'`, `'+z'`, `'-x'`, `'-y'`, or `'-z'` to specify which direction the 'strong' side of the array faces.

array.msize A (3×1) vector defining the size of each magnet in the array.

array.Nmag The number of magnets composing the array.

array.magn The magnetisation magnitude of each magnet.

array.magdir_rotate The amount of rotation, in degrees, between successive magnets.

Notes:

- The array must **face** in a direction orthogonal to its alignment.
- `'up'` and `'down'` are defined as synonyms for facing `'+z'` and `'-z'`, respectively, and `'linear'` for array type `'linear-x'`.

- Singleton input to `msize` assumes a cube-shaped magnet.

The variables above are the minimum set required to specify a multipole array. In addition, the following array variables may be used instead of or as well as to specify the information in a different way:

array.magdir_first This is the angle of magnetisation in degrees around the direction of magnetisation rotation for the first magnet. It defaults to $\pm 90^\circ$ depending on the facing direction of the array.

array.length The total length of the magnet array in the alignment direction of the array. If this variable is used then **width** and **height** (see below) must be as well.

array.width The dimension of the array orthogonal to the alignment and facing directions.

array.height The height of the array in the facing direction.

array.wavelength The wavelength of magnetisation. Must be an integer number of magnet lengths.

array.Nwaves The number of wavelengths of magnetisation in the array, which is probably always going to be an integer.

array.Nmag_per_wave The number of magnets per wavelength of magnetisation (e.g., `Nmag_per_wave` of four is equivalent to `magdir_rotate` of 90°).

array.gap Air-gap between successive magnet faces in the array. Defaults to zero.

Notes:

- `array.mlength+array.width+array.height` may be used as a synonymic replacement for `array.msize`.
- When using `Nwaves`, an additional magnet is placed on the end for symmetry.
- Setting `gap` does not affect `length` or `mlength`! That is, when `gap` is used, `length` refers to the total length of magnetic material placed end-to-end, not the total length of the array including the gaps.

Planar Halbach arrays Most of the information above follows for planar arrays, which can be thought of as a superposition of two orthogonal linear arrays.

array.type Use 'planar' to specify an array of this type.

array.align One of 'xy' (default), 'yz', or 'xz' for a plane with which to align the array.

array.width This is now the 'length' in the second spanning direction of the planar array. E.g., for the array 'planar-xy', 'length' refers to the *x*-direction and 'width' refers to the *y*-direction. (And 'height' is *z*.)

array.mwidth Ditto for the width of each magnet in the array.

All other variables for linear Halbach arrays hold analogously for planar Halbach arrays; if desired, two-element input can be given to specify different properties in different directions.

Planar quasi-Halbach arrays This magnetisation pattern is simpler than the planar Halbach array described above.

array.type Use 'quasi-halbach' to specify an array of this type.

array.Nwaves There are always four magnets per wavelength for the quasi-Halbach array. Two elements to specify the number of wavelengths in each direction, or just one if the same in both.

array.Nmag Instead of `Nwaves`, in case you want a non-integer number of wavelengths (but that would be weird).

Patchwork planar array

array.type Use ‘patchwork’ to specify an array of this type.

array.Nmag There isn’t really a ‘wavelength of magnetisation’ for this one; or rather, there is but it’s trivial. So just define the number of magnets per side, instead. (Two-element for different sizes of one-element for an equal number of magnets in both directions.)

Arbitrary arrays Until now we have assumed that magnet arrays are composed of magnets with identical sizes and regularly-varying magnetisation directions. Some facilities are provided to generate more general/arbitrary-shaped arrays.

array.type Should be ‘generic’ but may be omitted.

array.mcount The number of magnets in each direction, say (X, Y, Z) .

array.msize_array An $(X, Y, Z, 3)$ -length matrix defining the magnet sizes for each magnet of the array.

array.magdir_fn An anonymous function that takes three input variables (i, j, k) to calculate the magnetisation for the (i, j, k) -th magnet in the (x, y, z) -directions respectively.

array.magn At present this still must be singleton-valued. This will be amended at some stage to allow `magn_array` input to be analogous with `msize` and `msize_array`.

This approach for generating magnet arrays has been little-tested. Please inform me of associated problems if found.

3 Meta-information

Obtaining The latest version of this package may be obtained from the GitHub repository <http://github.com/wspr/magcode> with the following command:

```
git clone git://github.com/wspr/magcode.git
```

Installing It may be installed in Matlab simply by adding the ‘matlab/’ subdirectory to the Matlab path; e.g., adding the following to your `startup.m` file: (if that’s where you cloned the repository)

```
addpath ~/magcode/matlab
```

Licensing This work may be freely modified and distributed under the terms and conditions of the Apache License v2.0.³ This work is Copyright 2009–2010 by Will Robertson.

This means, in essence, that you may freely modify and distribute this code provided that you acknowledge your changes to the work and retain my copyright. See the License text for the specific language governing permissions and limitations under the License.

Contributing and feedback Please report problems and suggestions at the GitHub issue tracker.⁴

References

- [1] Gilles Akoun and Jean-Paul Yonnet. “3D analytical calculation of the forces exerted between two cuboidal magnets”. In: *IEEE Transactions on Magnetism* MAG-20.5 (Sept. 1984), pp. 1962–1964. DOI: [10.1109/TMAG.1984.1063554](https://doi.org/10.1109/TMAG.1984.1063554) (cit. on p. 19).

³<http://www.apache.org/licenses/LICENSE-2.0>

⁴<http://github.com/wspr/magcode/issues>

- [2] J.L.G. Janssen et al. “Three-Dimensional Analytical Calculation of the Torque between Permanent Magnets in Magnetic Bearings”. In: *IEEE Transactions on Magnetics* 46.6 (June 2010). DOI: [10.1109/TMAG.2010.2043224](https://doi.org/10.1109/TMAG.2010.2043224) (cit. on p. 25).
- [3] Jean-Paul Yonnet and Hicham Allag. “Analytical Calculation of Cuboïdal Magnet Interactions in 3D”. In: *The 7th International Symposium on Linear Drives for Industry Application*. 2009 (cit. on p. 20).

Part II

Typeset code / implementation

4 The magnetdefine() function

```
9 function [mag] = magnetdefine(varargin)

12 if nargin == 1
13     mag = varargin{1};
14 else
15     mag = struct(varargin{:});
16 end

18 if ~isfield(mag,'type')
19     if length(mag.dim)== 2
20         mag.type = 'cylinder';
21     else
22         mag.type = 'cuboid';
23     end
24 end

26 if isfield(mag,'grade')
27     if isfield(mag,'magn')
28         error('Cannot specify both ''magn''and ''grade''.')
29     else
30         mag.magn = grade2magn(mag.grade);
31     end
32 end

35 if strcmp(mag.type,'cylinder')

37 % default to +Z magnetisation
38 if ~isfield(mag,'dir')
39     if ~isfield(mag,'magdir')
40         mag.dir = [0 0 1];
41         mag.magdir = [0 0 1];
42     else
43         mag.dir = mag.magdir;
44     end
45 else
46     if ~isfield(mag,'magdir')
47         mag.magdir = mag.dir;
48     else
49         mag.magdir = [0 0 1];
50     end
51 end

53 % convert from current/turns to equiv magnetisation:
54 if ~isfield(mag,'magn')
55     mag.magn = 4*pi*1e-7*mag.turns*mag.current/mag.dim(2);
56 end
```



```

58 end
60 mag.fundefined = true;
62 end

```

4.1 grade2magn

Magnet ‘strength’ can be specified using either **magn** or **grade**. In the latter case, this should be a string such as ‘N42’, from which the **magn** is automatically calculated using the equation

$$B_r = 2\sqrt{\mu_0[BH]_{\max}}$$

where $[BH]_{\max}$ is the numeric value given in the grade in MG Oe. I.e., an N42 magnet has $[BH]_{\max} = 42$ MG Oe. Since $1 \text{ MG Oe} = 100/(4\pi) \text{ kJ/m}^3$, the calculation simplifies to

$$B_r = 2\sqrt{N/100}$$

where N is the numeric grade in MG Oe. Easy.

```

79 function magn = grade2magn(grade)
81 if isnumeric(grade)
82     magn = 2*sqrt(grade/100);
83 else
84     if strcmp(grade(1), 'N')
85         grade = grade(2:end);
86     end
87     magn = 2*sqrt(str2double(grade)/100);
88 end
90 end

```

4.1.1 grade2magn

Magnet ‘strength’ can be specified using either **magn** or **grade**. In the latter case, this should be a string such as ‘N42’, from which the **magn** is automatically calculated using the equation

$$B_r = 2\sqrt{\mu_0[BH]_{\max}}$$

where $[BH]_{\max}$ is the numeric value given in the grade in MG Oe. I.e., an N42 magnet has $[BH]_{\max} = 42$ MG Oe. Since $1 \text{ MG Oe} = 100/(4\pi) \text{ kJ/m}^3$, the calculation simplifies to

$$B_r = 2\sqrt{N/100}$$

where N is the numeric grade in MG Oe. Easy.

5 The magnetforces() function

```
101 function [varargout] = magnetforces(magnet_fixed, magnet_float, displ, varargin)
```

We now have a choice of calculations to take based on the user input. This chunk and the next are used in both `magnetforces.m` and `multipoleforces.m`.

```
108 debug_disp = @(str)disp([]);
109 calc_force_bool = false;
110 calc_stiffness_bool = false;
111 calc_torque_bool = false;
```

Undefined calculation flags for the three directions:

```
114 calc_xyz = [false; false; false];
116 for iii = 1:length(varargin)
117     switch varargin{iii}
118         case 'debug',    debug_disp = @(str)disp(str);
119         case 'force',    calc_force_bool = true;
120         case 'stiffness', calc_stiffness_bool = true;
121         case 'torque',    calc_torque_bool = true;
122         case 'x', calc_xyz(1)= true;
123         case 'y', calc_xyz(2)= true;
124         case 'z', calc_xyz(3)= true;
125         otherwise
126             error(['Unknown calculation option ',varargin{iii},'])
127     end
128 end
```

If none of 'x', 'y', 'z' are specified, calculate all.

```
131 if all( ~calc_xyz )
132     calc_xyz = [true; true; true];
133 end
135 if ~calc_force_bool && ~calc_stiffness_bool && ~calc_torque_bool
136     varargin{end+1} = 'force';
137     calc_force_bool = true;
138 end
```

Gotta check the displacement input for both functions. After sorting that out, we can initialise the output variables now we know how big they need to be.

```
145 if size(displ,1)== 3
146     % all good
147 elseif size(displ,2)== 3
148     displ = transpose(displ);
149 else
150     error(['Displacements matrix should be of size (3, D)',...
151         'where D is the number of displacements.'])
152 end
154 Ndispl = size(displ,2);
156 if calc_force_bool
```

```

157     forces_out = nan([3 Ndispl]);
158 end

160 if calc_stiffness_bool
161     stiffnesses_out = nan([3 Ndispl]);
162 end

164 if calc_torque_bool
165     torques_out = nan([3 Ndispl]);
166 end

```

First of all, address the data structures required for the input and output. Because displacement of a single magnet has three components, plus sizes of the faces another three, plus magnetisation strength and direction (two) makes nine in total, we use a structure to pass the information into the function. Otherwise we'd have an overwhelming number of input arguments.

The input variables `magnet.dim` should be the entire side lengths of the magnets; these dimensions are halved when performing all of the calculations. (Because that's just how the maths is.)

We use spherical coordinates to represent magnetisation angle, where `phi` is the angle from the horizontal plane ($-\pi/2 \leq \phi \leq \pi/2$) and `theta` is the angle around the horizontal plane ($0 \leq \theta \leq 2\pi$). This follows Matlab's definition; other conventions are commonly used as well. Remember:

$$\begin{aligned}
 (1, 0, 0)_{\text{cartesian}} &\equiv (0, 0, 1)_{\text{spherical}} \\
 (0, 1, 0)_{\text{cartesian}} &\equiv (\pi/2, 0, 1)_{\text{spherical}} \\
 (0, 0, 1)_{\text{cartesian}} &\equiv (0, \pi/2, 1)_{\text{spherical}}
 \end{aligned}$$

Cartesian components can also be used as input as well, in which case they are made into a unit vector before multiplying it by the magnetisation magnitude. Either way (between spherical or cartesian input), J1 and J2 are made into the magnetisation vectors in cartesian coordinates.

```

194 if ~isfield(magnet_fixed, 'fundefined')
195     magnet_fixed = magnetdefine(magnet_fixed);
196 end
197 if ~isfield(magnet_float, 'fundefined')
198     magnet_float = magnetdefine(magnet_float);
199 end

202 coil_bool = false;

204 if strcmp(magnet_fixed.type, 'coil')

206     if ~strcmp(magnet_float.type, 'cylinder')
207         error('Coil/magnet forces can only be calculated for cylindrical magnets.')
208     end

210     coil_bool = true;
211     coil = magnet_fixed;
212     magnet = magnet_float;
213     magtype = 'cylinder';
214     coil_sign = +1;

216 end

218 if strcmp(magnet_float.type, 'coil')

220     if ~strcmp(magnet_fixed.type, 'cylinder')
221         error('Coil/magnet forces can only be calculated for cylindrical magnets.')
222     end

```

```

224     coil_bool = true;
225     coil = magnet_float;
226     magnet = magnet_fixed;
227     magtype = 'cylinder';
228     coil_sign = -1;

230 end

232 if coil_bool
234     error('to do')
236 else

238     if ~strcmp(magnet_fixed.type, magnet_float.type)
239         error('Magnets must be of same type')
240     end
241     magtype = magnet_fixed.type;

244     if strcmp(magtype, 'cuboid')

246         size1 = reshape(magnet_fixed.dim/2, [3 1]);
247         size2 = reshape(magnet_float.dim/2, [3 1]);

249         J1 = resolve_magnetisations(magnet_fixed.magn, magnet_fixed.magdir);
250         J2 = resolve_magnetisations(magnet_float.magn, magnet_float.magdir);

252         if calc_torque_bool
253             if ~isfield(magnet_float, 'lever')
254                 magnet_float.lever = [0; 0; 0];
255             else
256                 ss = size(magnet_float.lever);
257                 if (ss(1)~=3)&& (ss(2)==3)
258                     magnet_float.lever = magnet_float.lever'; % attempt [3 M] shape
259                 end
260             end
261         end

263     elseif strcmp(magtype, 'cylinder')

265         size1 = magnet_fixed.dim(:);
266         size2 = magnet_float.dim(:);

268         if any(abs(magnet_fixed.dir)~= abs(magnet_float.dir))
269             error('Cylindrical magnets must be oriented in the same direction')
270         end
271         if any(abs(magnet_fixed.magdir)~= abs(magnet_float.magdir))
272             error('Cylindrical magnets must be oriented in the same direction')
273         end
274         if any(abs(magnet_fixed.dir)~= abs(magnet_fixed.magdir))
275             error('Cylindrical magnets must be magnetised in the same direction as their orientation
276 ')
277         end
278         if any(abs(magnet_float.dir)~= abs(magnet_float.magdir))
279             error('Cylindrical magnets must be magnetised in the same direction as their orientation
280 ')
281         end
282     end

```

```

281     cyldir = find(magnet_float.magdir ~= 0);
282     cylnotdir = find(magnet_float.magdir == 0);
283     if length(cyldir)~= 1
284         error('Cylindrical magnets must be aligned in one of the x, y or z directions')
285     end

287     magnet_float.magdir = magnet_float.magdir(:);
288     magnet_fixed.magdir = magnet_fixed.magdir(:);
289     magnet_float.dir = magnet_float.dir(:);
290     magnet_fixed.dir = magnet_fixed.dir(:);

292     J1 = magnet_fixed.magn*magnet_fixed.magdir;
293     J2 = magnet_float.magn*magnet_float.magdir;

295     end

297 end

300 magconst = 1/(4*pi*(4*pi*1e-7));

302 [index_i, index_j, index_k, index_l, index_p, index_q] = ndgrid([0 1]);
304 index_sum = (-1).^(index_i+index_j+index_k+index_l+index_p+index_q);

307 if strcmp(magtype, 'cuboid')

309     swap_x_y = @(vec)vec([2 1 3],:);
310     swap_x_z = @(vec)vec([3 2 1],:);
311     swap_y_z = @(vec)vec([1 3 2],:);

313     rotate_z_to_x = @(vec)[ vec(3,:); vec(2,:); -vec(1,:) ] ; % Ry( 90)
314     rotate_x_to_z = @(vec)[ -vec(3,:); vec(2,:); vec(1,:) ] ; % Ry(-90)

316     rotate_y_to_z = @(vec)[ vec(1,:); -vec(3,:); vec(2,:) ] ; % Rx( 90)
317     rotate_z_to_y = @(vec)[ vec(1,:); vec(3,:); -vec(2,:) ] ; % Rx(-90)

319     rotate_x_to_y = @(vec)[ -vec(2,:); vec(1,:); vec(3,:) ] ; % Rz( 90)
320     rotate_y_to_x = @(vec)[ vec(2,:); -vec(1,:); vec(3,:) ] ; % Rz(-90)

322     size1_x = swap_x_z(size1);
323     size2_x = swap_x_z(size2);
324     J1_x    = rotate_x_to_z(J1);
325     J2_x    = rotate_x_to_z(J2);

327     size1_y = swap_y_z(size1);
328     size2_y = swap_y_z(size2);
329     J1_y    = rotate_y_to_z(J1);
330     J2_y    = rotate_y_to_z(J2);

332 end

```

5.1 Calculate for each displacement

The actual mechanics. The idea is that a multitude of displacements can be passed to the function and we iterate to generate a matrix of vector outputs.

```
339 if coil_bool
341     forces_out = coil_sign*coil.dir*...
342     forces_magcyl_shell_calc(mag.dim, coil.dim, squeeze(displ(cyldir,:)), J1(cyldir),
        coil.current, coil.turns);
344 else
346     if strcmp(magtype,'cuboid')
348         if calc_force_bool
349             for iii = 1:Ndispl
350                 forces_out(:,iii)= single_magnet_force(displ(:,iii));
351             end
352         end
354         if calc_stiffness_bool
355             for iii = 1:Ndispl
356                 stiffnesses_out(:,iii)= single_magnet_stiffness(displ(:,iii));
357             end
358         end
360         if calc_torque_bool
361             torques_out = single_magnet_torque(displ,magnet_float.lever);
362         end
364     elseif strcmp(magtype,'cylinder')
366         if calc_force_bool
367             for iii = 1:Ndispl
368                 forces_out(:,iii)= single_magnet_cyl_force(displ(:,iii));
369             end
370         end
372         if calc_stiffness_bool
373             error('Stiffness cannot be calculated for cylindrical magnets yet.')
374         end
376         if calc_torque_bool
377             error('Torques cannot be calculated for cylindrical magnets yet.')
378         end
380     end
382 end
```

After all of the calculations have occurred, they're placed back into `varargout`. (This happens at the very end, obviously.) Outputs are ordered in the same order as the inputs are specified.

```
389 varargout = {};
391 for ii = 1:length(varargin)
392     switch varargin{ii}
393         case 'force'
394             varargout{end+1} = forces_out;
```

```

396     case 'stiffness'
397         varargout{end+1} = stiffnesses_out;
399     case 'torque'
400         varargout{end+1} = torques_out;
401     end
402 end

```

resolve_magnetisations Magnetisation directions are specified in either cartesian or spherical coordinates. Since this is shared code, it's sent to the end to belong in a nested function.

We don't use Matlab's `sph2cart` here, because it doesn't calculate zero accurately (because it uses radians and $\cos(\pi/2)$ can only be evaluated to machine precision of π rather than symbolically).

```

416 function J = resolve_magnetisations(magn,magdir)
418     if length(magdir)==2
419         J_r = magn;
420         J_t = magdir(1);
421         J_p = magdir(2);
422         J = [ J_r * cosd(J_p)* cosd(J_t); ...
423             J_r * cosd(J_p)* sind(J_t); ...
424             J_r * sind(J_p)];
425     else
426         if all(magdir == zeros(size(magdir)))
427             J = [0; 0; 0];
428         else
429             J = magn*magdir/norm(magdir);
430             J = reshape(J,[3 1]);
431         end
432     end
434 end

```

```

single_magnet_cyl_force0 function forces_out = single_magnet_cyl_force(displ)
442     forces_out = nan(size(displ));
444     ecc = sqrt(sum(displ(cylnotdir).^2));
446     if ecc < eps
447         forces_out = magnet_fixed.magdir*forces_cyl_calc(size1, size2, displ(cyldir), J1
(cyldir), J2(cyldir)).';
448     else
449         ecc_forces = forces_cyl_ecc_calc(size1, size2, displ(cyldir), ecc, J1(cyldir), J2
(cyldir)).';
450         forces_out(cyldir)= ecc_forces(2);
451         forces_out(cylnotdir(1))= displ(cylnotdir(1))/ecc*ecc_forces(1);
452         forces_out(cylnotdir(2))= displ(cylnotdir(2))/ecc*ecc_forces(1);
453 % not 100
454     end
456 end

```

single_magnet_force The x and y forces require a rotation to get the magnetisations correctly aligned. In the case of the magnet sizes, the lengths are just flipped rather than rotated (in rotation, sign is important). After the forces are calculated, rotate them back to the original coordinate system.

```

468 function force_out = single_magnet_force(displ)
470     force_components = nan([9 3]);
472     d_x = rotate_x_to_z(displ);
473     d_y = rotate_y_to_z(displ);
475     debug_disp(' ')
476     debug_disp('CALCULATING THINGS')
477     debug_disp('=====')
478     debug_disp('Displacement:')
479     debug_disp(displ)
480     debug_disp('Magnetisations:')
481     debug_disp(J1)
482     debug_disp(J2)
484     calc_xyz = swap_x_z(calc_xyz);
486     debug_disp('Forces x-x:')
487     force_components(1,:)= ...
488         rotate_z_to_x( forces_calc_z_z(size1_x,size2_x,d_x,J1_x,J2_x));
490     debug_disp('Forces x-y:')
491     force_components(2,:)= ...
492         rotate_z_to_x( forces_calc_z_y(size1_x,size2_x,d_x,J1_x,J2_x));
494     debug_disp('Forces x-z:')
495     force_components(3,:)= ...
496         rotate_z_to_x( forces_calc_z_x(size1_x,size2_x,d_x,J1_x,J2_x));
498     calc_xyz = swap_x_z(calc_xyz);
501     calc_xyz = swap_y_z(calc_xyz);
503     debug_disp('Forces y-x:')
504     force_components(4,:)= ...
505         rotate_z_to_y( forces_calc_z_x(size1_y,size2_y,d_y,J1_y,J2_y));
507     debug_disp('Forces y-y:')
508     force_components(5,:)= ...
509         rotate_z_to_y( forces_calc_z_z(size1_y,size2_y,d_y,J1_y,J2_y));
511     debug_disp('Forces y-z:')
512     force_components(6,:)= ...
513         rotate_z_to_y( forces_calc_z_y(size1_y,size2_y,d_y,J1_y,J2_y));
515     calc_xyz = swap_y_z(calc_xyz);
518     debug_disp('z-z force:')
519     force_components(9,:)= forces_calc_z_z( size1,size2,displ,J1,J2 );
521     debug_disp('z-y force:')
522     force_components(8,:)= forces_calc_z_y( size1,size2,displ,J1,J2 );

```



```

524     debug_disp('z-x force:')
525     force_components(7,:)= forces_calc_z_x( size1,size2,displ,J1,J2 );

528     force_out = sum(force_components);
529 end

single_magnet_torque = function torques_out = single_magnet_torque(displ,lever)
538     torque_components = nan([size(displ)9]);

541     d_x = rotate_x_to_z(displ);
542     d_y = rotate_y_to_z(displ);

544     l_x = rotate_x_to_z(lever);
545     l_y = rotate_y_to_z(lever);

548     debug_disp(' ')
549     debug_disp('CALCULATING THINGS')
550     debug_disp('=====')
551     debug_disp('Displacement:')
552     debug_disp(displ')
553     debug_disp('Magnetisations:')
554     debug_disp(J1')
555     debug_disp(J2')

558     debug_disp('Torque: z-z:')
559     torque_components(:, :, 9)= torques_calc_z_z( size1,size2,displ,lever,J1,J2 );

561     debug_disp('Torque z-y:')
562     torque_components(:, :, 8)= torques_calc_z_y( size1,size2,displ,lever,J1,J2 );

564     debug_disp('Torque z-x:')
565     torque_components(:, :, 7)= torques_calc_z_x( size1,size2,displ,lever,J1,J2 );

568     calc_xyz = swap_x_z(calc_xyz);

570     debug_disp('Torques x-x:')
571     torque_components(:, :, 1)= ...
572         rotate_z_to_x( torques_calc_z_z(size1_x,size2_x,d_x,l_x,J1_x,J2_x));

574     debug_disp('Torques x-y:')
575     torque_components(:, :, 2)= ...
576         rotate_z_to_x( torques_calc_z_y(size1_x,size2_x,d_x,l_x,J1_x,J2_x));

578     debug_disp('Torques x-z:')
579     torque_components(:, :, 3)= ...
580         rotate_z_to_x( torques_calc_z_x(size1_x,size2_x,d_x,l_x,J1_x,J2_x));

582     calc_xyz = swap_x_z(calc_xyz);

585     calc_xyz = swap_y_z(calc_xyz);

587     debug_disp('Torques y-x:')

```

```

588 torque_components(:,4)= ...
589     rotate_z_to_y( torques_calc_z_x(size1_y,size2_y,d_y,l_y,J1_y,J2_y));
591 debug_disp('Torques y-y:')
592 torque_components(:,5)= ...
593     rotate_z_to_y( torques_calc_z_z(size1_y,size2_y,d_y,l_y,J1_y,J2_y));
595 debug_disp('Torques y-z:')
596 torque_components(:,6)= ...
597     rotate_z_to_y( torques_calc_z_y(size1_y,size2_y,d_y,l_y,J1_y,J2_y));
599 calc_xyz = swap_y_z(calc_xyz);

602 torques_out = sum(torque_components,3);
603 end

609 function stiffness_out = single_magnet_stiffness(displ)
611     stiffness_components = nan([9 3]);

614     d_x = rotate_x_to_z(displ);
615     d_y = rotate_y_to_z(displ);

618     debug_disp(' ')
619     debug_disp('CALCULATING THINGS')
620     debug_disp('=====')
621     debug_disp('Displacement:')
622     debug_disp(displ)
623     debug_disp('Magnetisations:')
624     debug_disp(J1)
625     debug_disp(J2)

628     debug_disp('z-x stiffness:')
629     stiffness_components(7,:)= ...
630         stiffnesses_calc_z_x( size1,size2,displ,J1,J2 );
632     debug_disp('z-y stiffness:')
633     stiffness_components(8,:)= ...
634         stiffnesses_calc_z_y( size1,size2,displ,J1,J2 );
636     debug_disp('z-z stiffness:')
637     stiffness_components(9,:)= ...
638         stiffnesses_calc_z_z( size1,size2,displ,J1,J2 );
640     calc_xyz = swap_x_z(calc_xyz);
642     debug_disp('x-x stiffness:')
643     stiffness_components(1,:)= ...
644         swap_x_z( stiffnesses_calc_z_z( size1_x,size2_x,d_x,J1_x,J2_x ));
646     debug_disp('x-y stiffness:')
647     stiffness_components(2,:)= ...
648         swap_x_z( stiffnesses_calc_z_y( size1_x,size2_x,d_x,J1_x,J2_x ));
650     debug_disp('x-z stiffness:')
651     stiffness_components(3,:)= ...

```

```

652     swap_x_z( stiffnesses_calc_z_x( size1_x,size2_x,d_x,J1_x,J2_x ));
654     calc_xyz = swap_x_z(calc_xyz);
656     calc_xyz = swap_y_z(calc_xyz);
658     debug_disp('y-x stiffness:')
659     stiffness_components(4,:)= ...
660     swap_y_z( stiffnesses_calc_z_x( size1_y,size2_y,d_y,J1_y,J2_y ));
662     debug_disp('y-y stiffness:')
663     stiffness_components(5,:)= ...
664     swap_y_z( stiffnesses_calc_z_x( size1_y,size2_y,d_y,J1_y,J2_y ));
666     debug_disp('y-z stiffness:')
667     stiffness_components(6,:)= ...
668     swap_y_z( stiffnesses_calc_z_y( size1_y,size2_y,d_y,J1_y,J2_y ));
670     calc_xyz = swap_y_z(calc_xyz);

675     stiffness_out = sum(stiffness_components);
676 end

```

forces_calc_z_z The expressions here follow directly from Akoun and Yonnet [1].

Inputs:	size1=(a,b,c)	the half dimensions of the fixed magnet
	size2=(A,B,C)	the half dimensions of the floating magnet
	displ=(dx,dy,dz)	distance between magnet centres
	(J,J2)	magnetisations of the magnet in the z-direction
Outputs:	forces_xyz=(Fx,Fy,Fz)	Forces of the second magnet

```

695 function calc_out = forces_calc_z_z(size1,size2,offset,J1,J2)
697     J1 = J1(3);
698     J2 = J2(3);
700     if (J1==0 || J2==0)
701         debug_disp('Zero magnetisation.')
702         calc_out = [0; 0; 0];
703         return;
704     end
706     u = offset(1)+ size2(1)*(-1).^index_j - size1(1)*(-1).^index_i;
707     v = offset(2)+ size2(2)*(-1).^index_l - size1(2)*(-1).^index_k;
708     w = offset(3)+ size2(3)*(-1).^index_q - size1(3)*(-1).^index_p;
709     r = sqrt(u.^2+v.^2+w.^2);
712     if calc_xyz(1)
713         component_x = ...
714             + multiply_x_log_y( 0.5*(v.^2-w.^2), r-u )...
715             + multiply_x_log_y( u.*v, r-v )...
716             + v.*w.*atan1(u.*v,r.*w)...
717             + 0.5*r.*u;

```

```

718     end
720     if calc_xyz(2)
721         component_y = ...
722             + multiply_x_log_y( 0.5*(u.^2-w.^2), r-v )...
723             + multiply_x_log_y( u.*v, r-u )...
724             + u.*w.*atan1(u.*v,r.*w)...
725             + 0.5*r.*v;
726     end
728     if calc_xyz(3)
729         component_z = ...
730             - multiply_x_log_y( u.*w, r-u )...
731             - multiply_x_log_y( v.*w, r-v )...
732             + u.*v.*atan1(u.*v,r.*w)...
733             - r.*w;
734     end
737     if calc_xyz(1)
738         component_x = index_sum.*component_x;
739     else
740         component_x = 0;
741     end
743     if calc_xyz(2)
744         component_y = index_sum.*component_y;
745     else
746         component_y = 0;
747     end
749     if calc_xyz(3)
750         component_z = index_sum.*component_z;
751     else
752         component_z = 0;
753     end
755     calc_out = J1*J2*magconst .* ...
756         [ sum(component_x(:));
757           sum(component_y(:));
758           sum(component_z(:)) ] ;
760     debug_disp(calc_out')
762 end

```

forces_calc_z_y Orthogonal magnets forces given by Yonnet and Allag [3]. Note those equations seem to be written to calculate the force on the first magnet due to the second, so we negate all the values to get the force on the latter instead.

```

775 function calc_out = forces_calc_z_y(size1,size2,offset,J1,J2)
777     J1 = J1(3);
778     J2 = J2(2);

```

```

780     if (J1==0 || J2==0)
781         debug_disp('Zero magnetisation.')
782         calc_out = [0; 0; 0];
783         return;
784     end

786     u = offset(1)+ size2(1)*(-1).^index_j - size1(1)*(-1).^index_i;
787     v = offset(2)+ size2(2)*(-1).^index_l - size1(2)*(-1).^index_k;
788     w = offset(3)+ size2(3)*(-1).^index_q - size1(3)*(-1).^index_p;
789     r = sqrt(u.^2+v.^2+w.^2);

792     allag_correction = -1;

794     if calc_xyz(1)
795         component_x = ...
796             - multiply_x_log_y ( v .* w , r-u )...
797             + multiply_x_log_y ( v .* u , r+w )...
798             + multiply_x_log_y ( u .* w , r+v )...
799             - 0.5 * u.^2 .* atan1( v .* w , u .* r )...
800             - 0.5 * v.^2 .* atan1( u .* w , v .* r )...
801             - 0.5 * w.^2 .* atan1( u .* v , w .* r );
802         component_x = allag_correction*component_x;
803     end

805     if calc_xyz(2)
806         component_y = ...
807             0.5 * multiply_x_log_y( u.^2 - v.^2 , r+w )...
808             - multiply_x_log_y( u .* w , r-u )...
809             - u .* v .* atan1( u .* w , v .* r )...
810             - 0.5 * w .* r;
811         component_y = allag_correction*component_y;
812     end

814     if calc_xyz(3)
815         component_z = ...
816             0.5 * multiply_x_log_y( u.^2 - w.^2 , r+v )...
817             - multiply_x_log_y( u .* v , r-u )...
818             - u .* w .* atan1( u .* v , w .* r )...
819             - 0.5 * v .* r;
820         component_z = allag_correction*component_z;
821     end

824     if calc_xyz(1)
825         component_x = index_sum.*component_x;
826     else
827         component_x = 0;
828     end

830     if calc_xyz(2)
831         component_y = index_sum.*component_y;
832     else
833         component_y = 0;
834     end

836     if calc_xyz(3)

```

```

837     component_z = index_sum.*component_z;
838 else
839     component_z = 0;
840 end

842 calc_out = J1*J2*magconst .* ...
843     [ sum(component_x(:));
844       sum(component_y(:));
845       sum(component_z(:)) ] ;
847 debug_disp(calc_out')
849 end

forces_calc_z_x6 function calc_out = forces_calc_z_x(size1,size2,offset,J1,J2)

858     calc_xyz = swap_x_y(calc_xyz);
860     forces_xyz = forces_calc_z_y(...
861         swap_x_y(size1), swap_x_y(size2), rotate_x_to_y(offset),...
862         J1, rotate_x_to_y(J2));
864     calc_xyz = swap_x_y(calc_xyz);
865     calc_out = rotate_y_to_x( forces_xyz );
867 end

871 function calc_out = stiffnesses_calc_z_z(size1,size2,offset,J1,J2)

873     J1 = J1(3);
874     J2 = J2(3);

877     if (J1==0 || J2==0)
878         debug_disp('Zero magnetisation.')
879         calc_out = [0; 0; 0];
880         return;
881     end

883     u = offset(1)+ size2(1)*(-1).^index_j - size1(1)*(-1).^index_i;
884     v = offset(2)+ size2(2)*(-1).^index_l - size1(2)*(-1).^index_k;
885     w = offset(3)+ size2(3)*(-1).^index_q - size1(3)*(-1).^index_p;
886     r = sqrt(u.^2+v.^2+w.^2);

889     if calc_xyz(1)|| calc_xyz(3)
890         component_x = - r - (u.^2 .*v)./(u.^2+w.^2)- v.*log(r-v);
891     end

893     if calc_xyz(2)|| calc_xyz(3)
894         component_y = - r - (v.^2 .*u)./(v.^2+w.^2)- u.*log(r-u);
895     end

897     if calc_xyz(3)
898         component_z = - component_x - component_y;
899     end

```

```

902     if calc_xyz(1)
903         component_x = index_sum.*component_x;
904     else
905         component_x = 0;
906     end

908     if calc_xyz(2)
909         component_y = index_sum.*component_y;
910     else
911         component_y = 0;
912     end

914     if calc_xyz(3)
915         component_z = index_sum.*component_z;
916     else
917         component_z = 0;
918     end

920     calc_out = J1*J2*magconst .* ...
921         [ sum(component_x(:));
922           sum(component_y(:));
923           sum(component_z(:)) ] ;

925     debug_disp(calc_out')

927 end

```

stiffnesses_calc_z_y **function** calc_out = stiffnesses_calc_z_y(size1,size2,offset,J1,J2)

```

935     J1 = J1(3);
936     J2 = J2(2);

939     if (J1==0 || J2==0)
940         debug_disp('Zero magnetisation.')
941         calc_out = [0; 0; 0];
942         return;
943     end

945     u = offset(1)+ size2(1)*(-1).^index_j - size1(1)*(-1).^index_i;
946     v = offset(2)+ size2(2)*(-1).^index_l - size1(2)*(-1).^index_k;
947     w = offset(3)+ size2(3)*(-1).^index_q - size1(3)*(-1).^index_p;
948     r = sqrt(u.^2+v.^2+w.^2);

951     if calc_xyz(1)|| calc_xyz(3)
952         component_x = ((u.^2 .*v)./(u.^2 + v.^2))+ (u.^2 .*w)./(u.^2 + w.^2)...
953             - u.*atan1(v.*w,r.*u)+ multiply_x_log_y( w , r + v )+ ...
954             + multiply_x_log_y( v , r + w );
955     end

957     if calc_xyz(2)|| calc_xyz(3)
958         component_y = - v/2 + (u.^2 .*v)./(u.^2 + v.^2)- (u.*v.*w)./(v.^2 + w.^2)...
959             - u.*atan1(u.*w,r.*v)- multiply_x_log_y( v , r + w );

```

```

960     end

962     if calc_xyz(3)
963         component_z = - component_x - component_y;
964     end

967     if calc_xyz(1)
968         component_x = index_sum.*component_x;
969     else
970         component_x = 0;
971     end

973     if calc_xyz(2)
974         component_y = index_sum.*component_y;
975     else
976         component_y = 0;
977     end

979     if calc_xyz(3)
980         component_z = index_sum.*component_z;
981     else
982         component_z = 0;
983     end

985     calc_out = J1*J2*magconst .* ...
986         [ sum(component_x(:));
987           sum(component_y(:));
988           sum(component_z(:)) ] ;

990     debug_disp(calc_out')

992 end

```

```

stiffnesses_calc_z_x function calc_out = stiffnesses_calc_z_x(size1,size2,offset,J1
,J2)

1000     calc_xyz = swap_x_y(calc_xyz);

1002     stiffnesses_xyz = stiffnesses_calc_z_y(...
1003         swap_x_y(size1), swap_x_y(size2), rotate_x_to_y(offset),...
1004         J1, rotate_x_to_y(J2));

1006     calc_xyz = swap_x_y(calc_xyz);
1007     calc_out = swap_x_y(stiffnesses_xyz);

1009 end

```


torques_calc_z_z The expressions here follow directly from Janssen et al. [2]. The code below was largely written by Allan Liu; thanks! We have checked it against Janssen's own Matlab code and the two give identical output.

Inputs:	size1=(a1,b1,c1)	the half dimensions of the fixed magnet
	size2=(a2,b2,c2)	the half dimensions of the floating magnet
	displ=(a,b,c)	distance between magnet centres
	lever=(d,e,f)	distance between floating magnet and its centre of rotation
	(J,J2)	magnetisations of the magnet in the z-direction
Outputs:	forces_xyz=(Fx,Fy,Fz)	Forces of the second magnet

```

1033 function calc_out = torques_calc_z_z(size1,size2,offset,lever,J1,J2)
1034
1035     br1 = J1(3);
1036     br2 = J2(3);
1037
1038     if br1==0 || br2==0
1039         debug_disp('Zero magnetisation')
1040         calc_out = 0*offset;
1041         return
1042     end
1043
1044     a1 = size1(1);
1045     b1 = size1(2);
1046     c1 = size1(3);
1047
1048     a2 = size2(1);
1049     b2 = size2(2);
1050     c2 = size2(3);
1051
1052     a = offset(1,:);
1053     b = offset(2,:);
1054     c = offset(3,:);
1055
1056     d = a+lever(1,:);
1057     e = b+lever(2,:);
1058     f = c+lever(3,:);
1059
1060     Tx=zeros([1 size(offset,2)]);
1061     Ty=Tx;
1062     Tz=Tx;
1063
1064     for ii=[0,1]
1065         for jj=[0,1]
1066             for kk=[0,1]
1067                 for ll=[0,1]
1068                     for mm=[0,1]
1069                         for nn=[0,1]
1070
1071                             Cu=(-1)^ii.*a1-d;
1072                             Cv=(-1)^kk.*b1-e;
1073                             Cw=(-1)^mm.*c1-f;
1074
1075                             u=a-(-1)^ii.*a1+(-1)^jj.*a2;
1076                             v=b-(-1)^kk.*b1+(-1)^ll.*b2;

```

```

1077     w=c-(-1)^mm.*c1+(-1)^nn.*c2;
1079     s=sqrt(u.^2+v.^2+w.^2);
1081     Ex=(1/8).*(...
1082         -2.*Cw.*(-4.*v.*u+s.^2+2.*v.*s)-...
1083         w.*(-8.*v.*u+s.^2+8.*Cv.*s+6.*v.*s)+...
1084         2.*(2.*Cw+w).*(u.^2+w.^2).*log(v+s)+...
1085         4.*(...
1086         2.*Cv.*u.*w.*acoth(u./s)+ ...
1087         w.*(v.^2+2.*Cv.*v-w.*(2.*Cw+w)).*acoth(v./s)- ...
1088         u.*(...
1089         2*w.*(Cw+w).*atan(v./w)+ ...
1090         2*v.*(Cw+w).*log(s-u)+ ...
1091         (w.^2+2.*Cw.*w-v.*(2.*Cv+v)).*atan( u.*v./(w.*s))...
1092         )...
1093         )...
1094     );
1096     Ey=(1/8)*...
1097         ((2.*Cw+w).*u.^2-8.*u.*v.*(Cw+w)+8.*u.*v.*(Cw+w).*log(s-v)...
1098         +4.*Cw.*u.*s+6.*w.*s.*u+(2.*Cw+w).*(v.^2+w.^2)+...
1099         4.*w.*(w.^2+2.*Cw.*w-u.*(2.*Cu+u)).*acoth(u./s)+...
1100         4.*v.*(-2.*Cu.*w.*acoth(v./s)+2.*w.*(Cw+w).*atan(u./w)...
1101         +(w.^2+2.*Cw.*w-u.*(2.*Cu+u)).*atan(u.*v./(w.*s)))...
1102         -2.*(2.*Cw+w).*(v.^2+w.^2).*log(u+s)+8.*Cu.*w.*s);
1104     Ez=(1/36).*(-u.^3-18.*v.*u.^2-6.*u.*(w.^2+6.*Cu...
1105         .*v-3.*v.*(2.*Cv+v)+3.*Cv.*s)+v.*(v.^2+6.*(w.^2+...
1106         3.*Cu.*s))+6.*w.*(w.^2-3.*v.*(2.*Cv+v)).*atan(u./w)...
1107         -6.*w.*(w.^2-3.*u.*(2.*Cu+u)).*atan(v./w)-9.*...
1108         (2.*(v.^2+2.*Cv.*v-u.*(2.*Cu+u)).*w.*atan(u.*v./(w.*s))...
1109         -2.*u.*(2.*Cu+u).*v.*log(s-u)-(2.*Cv+v).*(v.^2-w.^2)...
1110         .*log(u+s)+2.*u.*v.*(2.*Cv+v).*log(s-v)+(2.*Cu+...
1111         u).*(u.^2-w.^2).*log(v+s)));
1113     Tx=Tx+(-1)^(ii+jj+kk+ll+mm+nn)*Ex;
1114     Ty=Ty+(-1)^(ii+jj+kk+ll+mm+nn)*Ey;
1115     Tz=Tz+(-1)^(ii+jj+kk+ll+mm+nn)*Ez;
1117     end
1118     end
1119     end
1120     end
1121     end
1122     end
1124     calc_out = real([Tx; Ty; Tz].*br1*br2/(16*pi^2*1e-7));
1126 end

```

```

torques_calc_z_y0    function calc_out = torques_calc_z_y(size1,size2,offset,lever,J1,
J2)

1132    if J1(3)~=0 && J2(2)~=0
1133        error('Torques cannot be calculated for orthogonal magnets yet.')
1134    end

1136    calc_out = 0*offset;

1138    end

torques_calc_z_x2    function calc_out = torques_calc_z_x(size1,size2,offset,lever,J1,
J2)

1144    if J1(3)~=0 && J2(1)~=0
1145        error('Torques cannot be calculated for orthogonal magnets yet.')
1146    end

1148    calc_out = 0*offset;

1150    end

forces_cyl_calc4    function calc_out = forces_cyl_calc(size1,size2,h_gap,J1,J2)

1156 % inputs

1158    r1 = size1(1);
1159    r2 = size2(1);

1161 % implicit

1163    z = nan(4,length(h_gap));
1164    z(1,:) = -size1(2)/2;
1165    z(2,:) = size1(2)/2;
1166    z(3,:) = h_gap - size2(2)/2;
1167    z(4,:) = h_gap + size2(2)/2;

1169    C_d = zeros(size(h_gap));

1171    for ii = [1 2]
1173        for jj = [3 4]

1175            a1 = z(ii,:)- z(jj,:);
1176            a2 = 1 + ( (r1-r2)./a1 ).^2;
1177            a3 = sqrt( (r1+r2).^2 + a1.^2 );
1178            a4 = 4*r1.*r2./ ( (r1+r2).^2 + a1.^2 );

1180            [K, E, PI] = ellipkepi( a4./(1-a2), a4 );

1182            a2_ind = ( a2 == 1 | isnan(a2));
1183            if all(a2_ind)% singularity at a2=1 (i.e., equal radii)
1184                PI_term(a2_ind)= 0;
1185            elseif all(~a2_ind)
1186                PI_term = (1-a1.^2./a3.^2).*PI;
1187            else % this branch just for completeness

```

```

1188     PI_term = zeros(size(a2));
1189     PI_term(~a2_ind)= (1-a1.^2/a3.^2).*PI;
1190 end
1192     f_z = a1.*a2.*a3.*( K - E./a2 - PI_term );
1194     f_z(abs(a1)<eps)=0; % singularity at a1=0 (i.e., coincident faces)
1196     C_d = C_d + (-1)^(ii+jj).*f_z;
1198 end
1200 end
1202     calc_out = J1*J2/(8*pi*1e-7)*C_d;
1204 end

forces_cyl_ecc_calc function calc_out = forces_cyl_calc(size1,size2,h_gap,J1,J2)
1210     r1 = size1(1);
1211     r2 = size2(1);
1213     z1 = -size1(2)/2;
1214     z2 = size1(2)/2;
1215     z3 = h_gap - size2(2)/2;
1216     z4 = h_gap + size2(2)/2;
1218     h = [z4-z2; z3-z2; z4-z1; z3-z1];
1220     fn = @(t)[xdir(t,r1,r2,h,e_displ), zdir(t,r1,r2,h,e_displ)];
1221     fn_int = integral(fn,0,pi,'ArrayValued',true,'AbsTol',1e-6);
1223     calc_out = -1e7*J1*J2*r1*r2*fn_int/4/pi/pi;
1225 function gx = xdir(t,r,R,h,p)
1227     X = sqrt(r^2+R^2-2*r*R*cos(t));
1228     hh = h.^2;
1229     ff = (p+X)^2+hh;
1230     gg = (p-X)^2+hh;
1231     f = sqrt(ff);
1232     g = sqrt(gg);
1233     m = 1-gg./ff; % equivalent to  $m = 4pX/f^2$ 
1235     [KK, EE] = ellipke(m);
1236     [F2, E2] = arrayfun(@elliptic12,asin(h./g),1-m);
1238     Ta = f.*EE;
1239     Tb = (p^2-X^2).*KK./f;
1240     Tc = sign(p-X)*h.*( F2.*(EE-KK)+ KK.*E2 - 1 );
1241     Td = -pi/2*h;
1243     T = cos(t)/p*(Ta+Tb+Tc+Td);
1244     gx = -T(1)+T(2)+T(3)-T(4);
1246 end
1248 function gz = zdir(t,r,R,h,p)
1250     XX = p^2+R^2-2*p*R*cos(t);

```

```

1251     rr = r.^2;
1252     X = sqrt(XX);
1253     hh = h.^2;
1254     ff = (r+X)^2+hh;
1255     gg = (r-X)^2+hh;
1256     f = sqrt(ff);
1257     g = sqrt(gg);
1258     m = 1-gg./ff;

1260     [KK, EE] = ellipke(m);
1261     [F2, E2] = arrayfun(@elliptic12,asin(h./g),1-m);

1263     Ta = +h.*f.*(EE-KK);
1264     Tb = -h.*KK.*(r-X)^2./f;
1265     Tc = abs(rr-XX).*( F2.*(EE-KK)+ KK.*E2 - 1 );
1266     Td = 4/pi.*min(rr,XX); % note  $r^2 + X^2 - |r^2 - X^2| = 2 \min(r^2, X^2)$ 

1268     T = (R-p.*cos(t))./(2.*r.*XX).*(Ta+Tb+Tc+Td);
1269     gz = -T(1)+T(2)+T(3)-T(4);

1271     end

1273 end

```

ellipkepi Complete elliptic integrals calculated with the arithmetic-geometric mean algorithms contained here: <http://dlmf.nist.gov/19.8>. Valid for $a \leq 1$ and $m \leq 1$.

```

1281 function [k,e,PI] = ellipkepi(a,m)

1283     a0 = 1;
1284     g0 = sqrt(1-m);
1285     s0 = m;
1286     nn = 0;

1288     p0 = sqrt(1-a);
1289     Q0 = 1;
1290     Q1 = 1;
1291     QQ = Q0;

1293     while max(Q1(:))> eps

1295     % for Elliptic I
1296         a1 = (a0+g0)/2;
1297         g1 = sqrt(a0.*g0);

1299     % for Elliptic II
1300         nn = nn + 1;
1301         c1 = (a0-g0)/2;
1302         w1 = 2^nn*c1.^2;
1303         s0 = s0 + w1;

1305     % for Elliptic III
1306         rr = p0.^2+a0.*g0;
1307         p1 = rr./(2.*p0);
1308         Q1 = 0.5*Q0.*(p0.^2-a0.*g0)./rr;
1309         QQ = QQ+Q1;

```

```

1311     a0 = a1;
1312     g0 = g1;
1313     Q0 = Q1;
1314     p0 = p1;

1316     end

1318     k = pi./(2*a1);
1319     e = k.*(1-s0/2);
1320     PI = pi./(4.*a1).*(2+a./(1-a).*QQ);

1322     im = find(m == 1);
1323     if ~isempty(im)
1324         k(im) = inf;
1325         e(im) = ones(length(im),1);
1326         PI(im) = inf;
1327     end

1329     end

1332     function [F,E] = elliptic12(u,m)
1333     % ELLIPTIC12 evaluates the value of the Incomplete Elliptic Integrals
1334     % of the First, Second Kind.
1335     % GNU GENERAL PUBLIC LICENSE Version 2, June 1991
1336     % Copyright (C) 2007 by Moiseev Igor.

1338     % EDITED BY WSPR to optimise for numel(u)=numel(m)=1
1339     % TODO: re-investigate vectorising once the wrapper code is properly in place

1341     tol = eps; % making this 1e-6 say makes it slower??

1343     F = zeros(size(u)); E = F; Z = E;

1345     m(m<eps)= 0;

1347     I = uint32( find(m ~= 1 & m ~= 0));
1348     if ~isempty(I)
1349         signU = sign(u(I));

1351     % pre-allocate space and augment if needed
1352     chunk = 7;
1353     a = zeros(chunk,1);
1354     c = a;
1355     b = a;
1356     a(1,:)= 1;
1357     c(1,:)= sqrt(m);
1358     b(1,:)= sqrt(1-m);
1359     n = uint32( zeros(1,1));
1360     i = 1;
1361     while any(abs(c(i,:))> tol)% Arithmetic-Geometric Mean of A, B and C
1362         i = i + 1;
1363         if i > size(a,1)
1364             a = [a; zeros(2,1)];
1365             b = [b; zeros(2,1)];
1366             c = [c; zeros(2,1)];
1367         end
1368         a(i,:) = 0.5 * (a(i-1,:)+ b(i-1,:));

```

```

1369     b(i,:)= sqrt(a(i-1,:).* b(i-1,:));
1370     c(i,:)= 0.5 * (a(i-1,:)- b(i-1,:));
1371     in = uint32( find((abs(c(i,:))<= tol)& (abs(c(i-1,:))> tol)));
1372     if ~isempty(in)
1373         [mi,ni] = size(in);
1374         n(in) = ones(mi,ni)*(i-1);
1375     end
1376 end

1378 mmax = length(I);
1379 mn = double(max(n));
1380 phin = zeros(1,mmax); C = zeros(1,mmax);
1381 Cp = C; e = uint32(C); phin(:)= signU.*u(I);
1382 i = 0; c2 = c.^2;
1383 while i < mn % Descending Landen Transformation
1384     i = i + 1;
1385     in = uint32(find(n > i));
1386     if ~isempty(in)
1387         phin(in)= atan(b(i)./a(i).*tan(phin(in)))+ ...
1388             pi.*ceil(phin(in)/pi - 0.5)+ phin(in);
1389         e(in) = 2.^(i-1);
1390         C(in) = C(in) + double(e(in(1)))*c2(i);
1391         Cp(in)= Cp(in)+ c(i+1).*sin(phin(in));
1392     end
1393 end

1395 Ff = phin ./ (a(mn).*double(e)*2);
1396 F(I) = Ff.*signU; % Incomplete Ell. Int. of the First Kind
1397 E(I) = (Cp + (1 - 1/2*C).* Ff).*signU; % Incomplete Ell. Int. of the Second Kind
1398 end

1400 % Special cases: m == 0, 1
1401 m0 = find(m == 0);
1402 if ~isempty(m0), F(m0)= u(m0); E(m0)= u(m0); end

1404 m1 = find(m == 1);
1405 um1 = abs(u(m1));
1406 if ~isempty(m1)
1407     N = floor( (um1+pi/2)/pi );
1408     M = find(um1 < pi/2);

1410     F(m1(M))= log(tan(pi/4 + u(m1(M))/2));
1411     F(m1(um1 >= pi/2))= Inf.*sign(u(m1(um1 >= pi/2)));

1413     E(m1) = ((-1).^N .* sin(um1)+ 2*N).*sign(u(m1));
1414 end
1415 end

```

```

forces_magcyl_shell_calc9 function Fz = forces_magcyl_shell_calc(magsize,coilsize,displ
,Jmag,Nrz,I)
1421     Jcoil = 4*pi*1e-7*Nrz(2)*I/coil.dim(3);
1423     shell_forces = nan([length(displ)Nrz(1)]);
1425     for rr = 1:Nrz(1)
1427         this_radius = coilsize(1)+(rr-1)/(Nrz(1)-1)*(coilsize(2)-coilsize(1));
1428         shell_size = [this_radius, coilsize(3)];
1430         shell_forces(:,rr)= forces_cyl_calc(magsize,shell_size,displ,Jmag,Jcoil);
1432     end
1434     Fz = sum(shell_forces,2);
1436 end

```

5.2 Helpers

The equations contain two singularities. Specifically, the equations contain terms of the form $x \log(y)$, which becomes NaN when both x and y are zero since $\log(0)$ is negative infinity.

multiply_x_log_y This function computes $x \log(y)$, special-casing the singularity to output zero, instead. (This is indeed the value of the limit.)

```

1447 function out = multiply_x_log_y(x,y)
1448     out = x.*log(y);
1449     out(~isfinite(out))=0;
1450 end

```

atan1 We're using **atan** instead of **atan2** (otherwise the wrong results are calculated — I guess I don't totally understand that), which becomes a problem when trying to compute **atan(0/0)** since 0/0 is NaN.

```

1457 function out = atan1(x,y)
1458     out = zeros(size(x));
1459     ind = x~=0 & y~=0;
1460     out(ind)= atan(x(ind)./y(ind));
1461 end
1464 end

```

grade2magn Magnet 'strength' can be specified using either **magn** or **grade**. In the latter case, this should be a string such as 'N42', from which the **magn** is automatically calculated using the equation

$$B_r = 2\sqrt{\mu_0[BH]_{\max}}$$

where $[BH]_{\max}$ is the numeric value given in the grade in MG Oe. I.e., an N42 magnet has $[BH]_{\max} = 42$ MG Oe. Since $1 \text{ MG Oe} = 100/(4\pi) \text{ kJ/m}^3$, the calculation simplifies to

$$B_r = 2\sqrt{N/100}$$

where N is the numeric grade in MG Oe. Easy.

6 The multipoleforces function

```
1474 function [varargout] = multipoleforces(fixed_array, float_array, displ, varargin)
1476 debug_disp = @(str)disp([]);
1477 calc_force_bool = false;
1478 calc_stiffness_bool = false;
1479 calc_torque_bool = false;
```

Undefined calculation flags for the three directions:

```
1482 calc_xyz = [-1; -1; -1];
1484 for ii = 1:length(varargin)
1485     switch varargin{ii}
1486         case 'debug',    debug_disp = @(str)disp(str);
1487         case 'force',    calc_force_bool    = true;
1488         case 'stiffness', calc_stiffness_bool = true;
1489         case 'torque',    calc_torque_bool   = true;
1490         case 'x', calc_xyz(1)= 1;
1491         case 'y', calc_xyz(2)= 1;
1492         case 'z', calc_xyz(3)= 1;
1493         otherwise
1494             error(['Unknown calculation option ''',varargin{ii},'''])
1495         end
1496 end
```

If none of 'x', 'y', 'z' are specified, calculate all.

```
1499 if all( calc_xyz == -1 )
1500     calc_xyz = [1; 1; 1];
1501 end
1503 calc_xyz( calc_xyz == -1 )= 0;
1505 if ~calc_force_bool && ~calc_stiffness_bool && ~calc_torque_bool
1506     varargin{end+1} = 'force';
1507     calc_force_bool = true;
1508 end
1511 if size(displ,1)== 3
1512     % all good
1513 elseif size(displ,2)== 3
1514     displ = transpose(displ);
1515 else
1516     error(['Displacements matrix should be of size (3, D)',...
1517         'where D is the number of displacements.'])
1518 end
1520 Ndispl = size(displ,2);
1522 if calc_force_bool
1523     forces_out = nan([3 Ndispl]);
1524 end
1526 if calc_stiffness_bool
1527     stiffnesses_out = nan([3 Ndispl]);
```

```

1528 end

1530 if calc_torque_bool
1531     torques_out = nan([3 Ndispl]);
1532 end

1535 part = @(x,y)x(y);

1537 fixed_array = complete_array_from_input(fixed_array);
1538 float_array = complete_array_from_input(float_array);

1540 if calc_force_bool
1541     array_forces = nan([3 Ndispl fixed_array.total float_array.total]);
1542 end

1544 if calc_stiffness_bool
1545     array_stiffnesses = nan([3 Ndispl fixed_array.total float_array.total]);
1546 end

1548 displ_from_array_corners = displ ...
1549 + repmat(fixed_array.size/2,[1 Ndispl])...
1550 - repmat(float_array.size/2,[1 Ndispl]);

1553 for ii = 1:fixed_array.total

1555     fixed_magnet = struct(...
1556         'dim',    fixed_array.dim(ii,:), ...
1557         'magn',   fixed_array.magn(ii), ...
1558         'magdir', fixed_array.magdir(ii,:)...
1559     );

1561     for jj = 1:float_array.total

1563         float_magnet = struct(...
1564             'dim',    float_array.dim(jj,:), ...
1565             'magn',   float_array.magn(jj), ...
1566             'magdir', float_array.magdir(jj,:)...
1567         );

1569         mag_displ = displ_from_array_corners ...
1570             - repmat(fixed_array.magloc(ii,:)',[1 Ndispl])...
1571             + repmat(float_array.magloc(jj,:)',[1 Ndispl]);

1573         if calc_force_bool && ~calc_stiffness_bool
1574             array_forces(:, :, ii, jj) = ...
1575                 magnetforces(fixed_magnet, float_magnet, mag_displ, varargin{:});
1576         elseif calc_stiffness_bool && ~calc_force_bool
1577             array_stiffnesses(:, :, ii, jj) = ...
1578                 magnetforces(fixed_magnet, float_magnet, mag_displ, varargin{:});
1579         else
1580             [array_forces(:, :, ii, jj) array_stiffnesses(:, :, ii, jj)] = ...
1581                 magnetforces(fixed_magnet, float_magnet, mag_displ, varargin{:});
1582         end

1584     end
1585 end

1587 if calc_force_bool

```

```

1588     forces_out = sum(sum(array_forces,4),3);
1589 end

1591 if calc_stiffness_bool
1592     stiffnesses_out = sum(sum(array_stiffnesses,4),3);
1593 end

1596 varargout = {};

1598 for ii = 1:length(varargin)
1599     switch varargin{ii}
1600         case 'force'
1601             varargout{end+1} = forces_out;

1603         case 'stiffness'
1604             varargout{end+1} = stiffnesses_out;

1606         case 'torque'
1607             varargout{end+1} = torques_out;
1608     end
1609 end

1615 function array = complete_array_from_input(array)

1617 if ~isfield(array,'type')
1618     array.type = 'generic';
1619 end

1622 if ~isfield(array,'face')
1623     array.face = 'undefined';
1624 end

1626 linear_index = 0;
1627 planar_index = [0 0];

1629 switch array.type
1630     case 'generic'
1631     case 'linear',           linear_index = 1;
1632     case 'linear-quasi',     linear_index = 1;
1633     case 'planar',          planar_index = [1 2];
1634     case 'quasi-halbach',    planar_index = [1 2];
1635     case 'patchwork',        planar_index = [1 2];
1636     otherwise
1637         error(['Unknown array type ''',array.type,','.'])
1638 end

1640 if ~isequal(array.type,'generic')
1641     if linear_index == 1
1642         if ~isfield(array,'align')
1643             array.align = 'x';
1644         end
1645         switch array.align
1646             case 'x', linear_index = 1;
1647             case 'y', linear_index = 2;
1648             case 'z', linear_index = 3;

```

```

1649     otherwise
1650         error('Alignment for linear array must be 'x', 'y', or 'z'.')
1651     end
1652 else
1653     if ~isfield(array,'align')
1654         array.align = 'xy';
1655     end
1656     switch array.align
1657         case 'xy', planar_index = [1 2];
1658         case 'yz', planar_index = [2 3];
1659         case 'xz', planar_index = [1 3];
1660     otherwise
1661         error('Alignment for planar array must be 'xy', 'yz', or 'xz'.')
1662     end
1663 end
1664 end

1666 switch array.face
1667     case {'+x','-x'}, facing_index = 1;
1668     case {'+y','-y'}, facing_index = 2;
1669     case {'up','down'}, facing_index = 3;
1670     case {'+z','-z'}, facing_index = 3;
1671     case 'undefined', facing_index = 0;
1672 end

1674 if linear_index ~= 0
1675     if linear_index == facing_index
1676         error('Arrays cannot face into their alignment direction.')
1677     end
1678 elseif ~isequal( planar_index, [0 0] )
1679     if any( planar_index == facing_index )
1680         error('Planar-type arrays can only face into their orthogonal direction')
1681     end
1682 end

1685 switch array.type
1686     case 'linear'

1688 array = extrapolate_variables(array);

1690 array.mcount = ones(1,3);
1691 array.mcount(linear_index)= array.Nmag;

1693     case 'linear-quasi'

1696 if isfield(array,'ratio')&& isfield(array,'mlength')
1697     error('Cannot specify both 'ratio'and 'mlength'.')
1698 elseif ~isfield(array,'ratio')&& ~isfield(array,'mlength')
1699     error('Must specify either 'ratio'or 'mlength'.')
1700 end

1703 array.Nmag_per_wave = 4;
1704 array.magdir_rotate = 90;

1706 if isfield(array,'Nwaves')

```

```

1707     array.Nmag = array.Nmag_per_wave*array.Nwaves+1;
1708 else
1709     error('''Nwaves''must be specified.')
1710 end

1712 if isfield(array,'mlength')
1713     if numel(array.mlength)~=2
1714         error('''mlength''must have length two for linear-quasi arrays.')
1715     end
1716     array.ratio = array.mlength(2)/array.mlength(1);
1717 else
1718     if isfield(array,'length')
1719         array.mlength(1)= 2*array.length/(array.Nmag*(1+array.ratio)+1-array.ratio);
1720         array.mlength(2)= array.mlength(1)*array.ratio;
1721     else
1722         error('''length''must be specified.')
1723     end
1724 end

1726 array.mcount = ones(1,3);
1727 array.mcount(linear_index)= array.Nmag;

1729 array.msize = nan([array.mcount 3]);

1731 [sindex_x sindex_y sindex_z] = ...
1732     meshgrid(1:array.mcount(1), 1:array.mcount(2), 1:array.mcount(3));

1736 all_indices = [1 1 1];
1737 all_indices(linear_index)= 0;
1738 all_indices(facing_index)= 0;
1739 width_index = find(all_indices);

1741 for ii = 1:array.Nmag
1742     array.msize(sindex_x(ii),sindex_y(ii),sindex_z(ii),linear_index)= ...
1743         array.mlength(mod(ii-1,2)+1);
1744     array.msize(sindex_x(ii),sindex_y(ii),sindex_z(ii),facing_index)= ...
1745         array.height;
1746     array.msize(sindex_x(ii),sindex_y(ii),sindex_z(ii),width_index)= ...
1747         array.width;
1748 end

1751 case 'planar'

1753 if isfield(array,'length')
1754     if length(array.length)== 1
1755         if isfield(array,'width')
1756             array.length = [ array.length array.width ];
1757         else
1758             array.length = [ array.length array.length ];
1759         end
1760     end
1761 end

1763 if isfield(array,'mlength')
1764     if length(array.mlength)== 1

```

```

1765     if isfield(array,mwidth)
1766         array.mlength = [ array.mlength array.mwidth ];
1767     else
1768         array.mlength = [ array.mlength array.mlength ];
1769     end
1770 end
1771 end

1773 var_names = {'length','mlength','wavelength','Nwaves',...
1774             'Nmag','Nmag_per_wave','magdir_rotate'};

1776 tmp_array1 = struct();
1777 tmp_array2 = struct();
1778 var_index = zeros(size(var_names));

1780 for iii = 1:length(var_names)
1781     if isfield(array,var_names(iii))
1782         tmp_array1.(var_names{iii})= array.(var_names{iii})(1);
1783         tmp_array2.(var_names{iii})= array.(var_names{iii})(end);
1784     else
1785         var_index(iii)= 1;
1786     end
1787 end

1789 tmp_array1 = extrapolate_variables(tmp_array1);
1790 tmp_array2 = extrapolate_variables(tmp_array2);

1792 for iii = find(var_index)
1793     array.(var_names{iii})= [tmp_array1.(var_names{iii})tmp_array2.(var_names{iii})];
1794 end

1796 array.width = array.length(2);
1797 array.length = array.length(1);

1799 array.mwidth = array.mlength(2);
1800 array.mlength = array.mlength(1);

1802 array.mcount = ones(1,3);
1803 array.mcount(planar_index)= array.Nmag;

1805 case 'quasi-halbach'

1807 if isfield(array,'mcount')
1808     if numel(array.mcount)~=3
1809         error(''mcount''must always have three elements.')
1810     end
1811 elseif isfield(array,'Nwaves')
1812     if numel(array.Nwaves)> 2
1813         error(''Nwaves''must have one or two elements only.')
1814     end
1815     array.mcount(facing_index)= 1;
1816     array.mcount(planar_index)= 4*array.Nwaves+1;
1817 elseif isfield(array,'Nmag')
1818     if numel(array.Nmag)> 2
1819         error(''Nmag''must have one or two elements only.')
1820     end
1821     array.mcount(facing_index)= 1;

```

```

1822     array.mcount(planar_index)= array.Nmag;
1823 else
1824     error('Must specify the number of magnets (''mcount''or ''Nmag'')or wavelengths (''
1825     Nwaves'')')
1826 end

1827 case 'patchwork'

1828 if isfield(array,'mcount')
1829     if numel(array.mcount)~=3
1830         error('''mcount''must always have three elements.')
1831     end
1832 elseif isfield(array,'Nmag')
1833     if numel(array.Nmag)> 2
1834         error('''Nmag''must have one or two elements only.')
1835     end
1836     array.mcount(facing_index)= 1;
1837     array.mcount(planar_index)= array.Nmag;
1838 else
1839     error('Must specify the number of magnets (''mcount''or ''Nmag'')')
1840 end
1841 end

1842 array.total = prod(array.mcount);

1843 if ~isfield(array,'msize')
1844     array.msize = [NaN NaN NaN];
1845     if linear_index ~=0
1846         array.msize(linear_index)= array.mlength;
1847         array.msize(facing_index)= array.height;
1848         array.msize(isnan(array.msize))= array.width;
1849     elseif ~isequal( planar_index, [0 0] )
1850         array.msize(planar_index)= [array.mlength array.mwidth];
1851         array.msize(facing_index)= array.height;
1852     else
1853         error('The array property ''msize''is not defined and I have no way to infer it.'
1854         )
1855     end
1856 elseif numel(array.msize)== 1
1857     array.msize = repmat(array.msize,[3 1]);
1858 end

1859 if numel(array.msize)== 3
1860     array.msize_array = ...
1861     repmat(reshape(array.msize,[1 1 1 3]), array.mcount);
1862 else
1863     if isequal([array.mcount 3],size(array.msize))
1864         array.msize_array = array.msize;
1865     else
1866         error('Magnet size ''msize''must have three elements (or one element for a cube magnet
1867         ).')
1868     end
1869 end
1870 end

```

```

1874 array.dim = reshape(array.msize_array, [array.total 3]);
1876 if ~isfield(array, 'mgap')
1877     array.mgap = [0; 0; 0];
1878 elseif length(array.mgap) == 1
1879     array.mgap = repmat(array.mgap, [3 1]);
1880 end

1884 if ~isfield(array, 'magn')
1885     if isfield(array, 'grade')
1886         array.magn = grade2magn(array.grade);
1887     else
1888         array.magn = 1;
1889     end
1890 end

1892 if length(array.magn) == 1
1893     array.magn = repmat(array.magn, [array.total 1]);
1894 else
1895     error('Magnetisation magnitude ''magn'' must be a single value.')
1896 end

1900 if ~isfield(array, 'magdir_fn')
1902     if ~isfield(array, 'face')
1903         array.face = '+z';
1904     end

1906     switch array.face
1907     case {'up', '+z', '+y', '+x'}, magdir_rotate_sign = 1;
1908     case {'down', '-z', '-y', '-x'}, magdir_rotate_sign = -1;
1909     end

1911     if ~isfield(array, 'magdir_first')
1912         array.magdir_first = magdir_rotate_sign*90;
1913     end

1915     magdir_fn_comp{1} = @(ii,jj,kk)0;
1916     magdir_fn_comp{2} = @(ii,jj,kk)0;
1917     magdir_fn_comp{3} = @(ii,jj,kk)0;

1919     switch array.type
1920     case 'linear'
1921         magdir_theta = @(nn)...
1922             array.magdir_first+magdir_rotate_sign*array.magdir_rotate*(nn-1);

1924         magdir_fn_comp{linear_index} = @(ii,jj,kk)...
1925             cosd(magdir_theta(part([ii,jj,kk], linear_index)));

1927         magdir_fn_comp{facing_index} = @(ii,jj,kk)...
1928             sind(magdir_theta(part([ii,jj,kk], linear_index)));

1930     case 'linear-quasi'

1932         magdir_theta = @(nn)...
1933             array.magdir_first+magdir_rotate_sign*90*(nn-1);

```



```

1935     magdir_fn_comp{linear_index} = @(ii,jj,kk)...
1936         cosd(magdir_theta(part([ii,jj,kk],linear_index))));
1937
1938     magdir_fn_comp{facing_index} = @(ii,jj,kk)...
1939         sind(magdir_theta(part([ii,jj,kk],linear_index))));
1940
1941     case 'planar'
1942
1943         magdir_theta = @(nn)...
1944             array.magdir_first(1)+magdir_rotate_sign*array.magdir_rotate(1)*(nn-1);
1945
1946         magdir_phi = @(nn)...
1947             array.magdir_first(end)+magdir_rotate_sign*array.magdir_rotate(end)*(nn-1);
1948
1949         magdir_fn_comp{planar_index(1)} = @(ii,jj,kk)...
1950             cosd(magdir_theta(part([ii,jj,kk],planar_index(2)))));
1951
1952         magdir_fn_comp{planar_index(2)} = @(ii,jj,kk)...
1953             cosd(magdir_phi(part([ii,jj,kk],planar_index(1)))));
1954
1955         magdir_fn_comp{facing_index} = @(ii,jj,kk)...
1956             sind(magdir_theta(part([ii,jj,kk],planar_index(1))))...
1957             + sind(magdir_phi(part([ii,jj,kk],planar_index(2)))));
1958
1959     case 'patchwork'
1960
1961         magdir_fn_comp{planar_index(1)} = @(ii,jj,kk)0;
1962
1963         magdir_fn_comp{planar_index(2)} = @(ii,jj,kk)0;
1964
1965         magdir_fn_comp{facing_index} = @(ii,jj,kk)...
1966             magdir_rotate_sign*(-1)^( ...
1967                 part([ii,jj,kk],planar_index(1))...
1968                 + part([ii,jj,kk],planar_index(2))...
1969                 + 1 ...
1970             );
1971
1972     case 'quasi-halbach'
1973
1974         magdir_fn_comp{planar_index(1)} = @(ii,jj,kk)...
1975             sind(90*part([ii,jj,kk],planar_index(1)))...
1976             * cosd(90*part([ii,jj,kk],planar_index(2)));
1977
1978         magdir_fn_comp{planar_index(2)} = @(ii,jj,kk)...
1979             cosd(90*part([ii,jj,kk],planar_index(1)))...
1980             * sind(90*part([ii,jj,kk],planar_index(2)));
1981
1982         magdir_fn_comp{facing_index} = @(ii,jj,kk)...
1983             magdir_rotate_sign ...
1984             * sind(90*part([ii,jj,kk],planar_index(1)))...
1985             * sind(90*part([ii,jj,kk],planar_index(2)));
1986
1987     otherwise
1988         error('Array property ''magdir_fn''not defined and I have no way to infer it.')
1989     end
1990
1991     array.magdir_fn = @(ii,jj,kk)...
1992         [ magdir_fn_comp{1}(ii,jj,kk)...
1993           magdir_fn_comp{2}(ii,jj,kk)...
1994           magdir_fn_comp{3}(ii,jj,kk)];

```

```

1996 end

2002 array.magloc = nan([array.total 3]);
2003 array.magdir = array.magloc;
2004 arrat.magloc_array = nan([array.mcount(1)array.mcount(2)array.mcount(3)3]);

2006 nn = 0;
2007 for iii = 1:array.mcount(1)
2008     for jjj = 1:array.mcount(2)
2009         for kkk = 1:array.mcount(3)
2010             nn = nn + 1;
2011             array.magdir(nn,:)= array.magdir_fn(iii,jjj,kkk);
2012         end
2013     end
2014 end

2016 magsep_x = zeros(size(array.mcount(1)));
2017 magsep_y = zeros(size(array.mcount(2)));
2018 magsep_z = zeros(size(array.mcount(3)));

2020 magsep_x(1)= array.msize_array(1,1,1,1)/2;
2021 magsep_y(1)= array.msize_array(1,1,1,2)/2;
2022 magsep_z(1)= array.msize_array(1,1,1,3)/2;

2024 for iii = 2:array.mcount(1)
2025     magsep_x(iii)= array.msize_array(iii-1,1,1,1)/2 ...
2026         + array.msize_array(iii ,1,1,1)/2 ;
2027 end
2028 for jjj = 2:array.mcount(2)
2029     magsep_y(jjj)= array.msize_array(1,jjj-1,1,2)/2 ...
2030         + array.msize_array(1,jjj ,1,2)/2 ;
2031 end
2032 for kkk = 2:array.mcount(3)
2033     magsep_z(kkk)= array.msize_array(1,1,kkk-1,3)/2 ...
2034         + array.msize_array(1,1,kkk ,3)/2 ;
2035 end

2037 magloc_x = cumsum(magsep_x);
2038 magloc_y = cumsum(magsep_y);
2039 magloc_z = cumsum(magsep_z);

2041 for iii = 1:array.mcount(1)
2042     for jjj = 1:array.mcount(2)
2043         for kkk = 1:array.mcount(3)
2044             array.magloc_array(iii,jjj,kkk,:)= ...
2045                 [magloc_x(iii); magloc_y(jjj); magloc_z(kkk)] ...
2046                 + [iii-1; jjj-1; kkk-1].*array.mgap;
2047         end
2048     end
2049 end
2050 array.magloc = reshape(array.magloc_array,[array.total 3]);

2052 array.size = squeeze( array.magloc_array(end,end,end,:)...
2053     - array.magloc_array(1,1,1,:)...

```

```

2054         + array.msize_array(1,1,1,:)/2 ...
2055         + array.msize_array(end,end,end,:)/2 );

2057 debug_disp('Magnetisation directions')
2058 debug_disp(array.magdir)

2060 debug_disp('Magnet locations:')
2061 debug_disp(array.magloc)

2064 end

2068 function array_out = extrapolate_variables(array)
2070 var_names = {'wavelength','length','Nwaves','mlength',...
2071             'Nmag','Nmag_per_wave','magdir_rotate'};

2073 if isfield(array,'Nwaves')
2074     mcount_extra = 1;
2075 else
2076     mcount_extra = 0;
2077 end

2079 if isfield(array,'mlength')
2080     mlength_adjust = false;
2081 else
2082     mlength_adjust = true;
2083 end

2085 variables = nan([7 1]);

2087 for iii = 1:length(var_names);
2088     if isfield(array,var_names(iii))
2089         variables(iii)= array.(var_names{iii});
2090     end
2091 end

2093 var_matrix = ...
2094     [1, 0, 0, -1, 0, -1, 0;
2095      0, 1, 0, -1, -1, 0, 0;
2096      0, 0, 1, 0, -1, 1, 0;
2097      0, 0, 0, 0, 0, 1, 1];

2099 var_results = [0 0 0 log(360)]';
2100 variables = log(variables);

2102 idx = ~isnan(variables);
2103 var_known = var_matrix(:,idx)*variables(idx);
2104 var_calc = var_matrix(:,~idx)\(var_results-var_known);
2105 variables(~idx)= var_calc;
2106 variables = exp(variables);

2108 for iii = 1:length(var_names);
2109     array.(var_names{iii})= variables(iii);
2110 end

2112 array.Nmag = round(array.Nmag)+ mcount_extra;
2113 array.Nmag_per_wave = round(array.Nmag_per_wave);

```

```
2115 if mlength_adjust
2116     array.mlength = array.mlength * (array.Nmag-mcount_extra)/array.Nmag;
2117 end
2119 array_out = array;
2121 end
2125 end
```