

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

Will Robertson

March 12, 2019

## **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

<b>I</b>	<b>User guide</b>	<b>3</b>
<b>1</b>	<b>Defining magnets and coils</b>	<b>3</b>
<b>2</b>	<b>Forces</b>	<b>4</b>
2.1	Forces between magnets . . . . .	4
2.2	Forces between multipole arrays of magnets . . . . .	5
<b>3</b>	<b>Meta-information</b>	<b>7</b>
<b>II</b>	<b>Typeset code / implementation</b>	<b>8</b>
<b>4</b>	<b>Magnets setup</b>	<b>8</b>
4.1	The magnetdefine() function . . . . .	8
4.1.1	The grade2magn() function . . . . .	10
4.1.2	The make_unit_vector() function . . . . .	10
<b>5</b>	<b>The magnetforces() function</b>	<b>11</b>
5.1	The single_magnet_cyl_force() function . . . . .	16
5.2	The single_magnet_ring_force() function . . . . .	16
5.3	The single_magnet_force() function . . . . .	17
5.4	The single_magnet_torque() function . . . . .	18
5.5	The single_magnet_stiffness() function . . . . .	18
5.6	The stiffnesses_calc_z_z() function . . . . .	19
5.7	The stiffnesses_calc_z_y() function . . . . .	20
5.7.1	Helpers . . . . .	20
5.7.2	The multiply_x_log_y() function . . . . .	20
5.7.3	The atan1() function . . . . .	21
5.8	The stiffnesses_calc_z_x() function . . . . .	21
5.9	The torques_calc_z_y() function . . . . .	21
5.10	The torques_calc_z_x() function . . . . .	21
5.11	The forces_magcyl_shell_calc() function . . . . .	22
<b>6</b>	<b>Magnet interactions</b>	<b>22</b>
6.1	The cuboid_force_z_z() function . . . . .	22
6.2	The cuboid_force_z_y() function . . . . .	24
6.3	The cuboid_force_z_x() function . . . . .	26
6.4	The cuboid_torque_z_z() function . . . . .	28
<b>7</b>	<b>Mathematical functions</b>	<b>34</b>
7.1	The ellipkepi() function . . . . .	34
<b>8</b>	<b>Magnet arrays</b>	<b>35</b>
8.1	The multipoleforces() function . . . . .	35

# 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.

#### 1.1 Cuboid magnets

For cuboid magnets, the following should be specified:

**'dim'** A  $(3 \times 1)$  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; either a  $(3 \times 1)$  vector in cartesian coordinates or a string such as '+x'.

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.

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

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

#### 1.2 Cylindrical and ring 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 or ring magnets.

Support for ring magnets is preliminary.

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.)

**'radius'** For cylindrical magnets, a  $(1 \times 1)$  element specifying the magnet radius. For ring magnets, a  $(2 \times 1)$  vector containing the inner and outer radius, resp.

**'dim'** A  $(2 \times 1)$  or  $(3 \times 1)$  vector containing, respectively, the magnet radius and length (for cylinders) or magnet radii and length (for rings).

**'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.

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

**'magdir'** A vector representing the direction of the magnetisation; either a  $(3 \times 1)$  vector in cartesian coordinates or a string such as '+x'.

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.

### 1.3 Coil—unfinished!

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  $(3 \times 1)$  vector containing, respectively, the inner coil radius, the outer coil radius, and the coil length.

**'turns'** A  $(2 \times 1)$  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` 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 respectively. The force<sup>1</sup> 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.

---

<sup>1</sup>From 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  $(3 \times 1)$  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');
```

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  $(3 \times 1)$  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  $90^\circ$ ).

**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.<sup>2</sup> 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.<sup>3</sup>

---

<sup>2</sup><http://www.apache.org/licenses/LICENSE-2.0>

<sup>3</sup><http://github.com/wspr/magcode/issues>

## Part II

# Typeset code / implementation

## 4 Magnets setup

### 4.1 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     warning('Magnets should always define their "type". E.g., {'type','cuboid'} for
    a cuboid magnet.')
20     if length(mag.dim)== 2
21         mag.type = 'cylinder';
22     else
23         mag.type = 'cuboid';
24     end
25 end

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

36 if ~isfield(mag,'lever')
37     mag.lever = [0; 0; 0];
38 else
39     ss = size(mag.lever);
40     if (ss(1)~=3)&& (ss(2)==3)
41         mag.lever = mag.lever.'; % attempt [3 M] shape
42     end
43 end

45 if isfield(mag,'magdir')
46     mag.magdir = make_unit_vector(mag,'magdir');
47 end
48 if isfield(mag,'dir')
49     mag.dir = make_unit_vector(mag,'dir');
50 end
51 if isfield(mag,'dim')
52     mag.dim = mag.dim(:);
53 end
```



```

55 if strcmp(mag.type,'cylinder')|| strcmp(mag.type,'ring')
56 % default to +Z magnetisation
57 if ~isfield(mag,'dir')
58     if ~isfield(mag,'magdir')
59         mag.dir = [0; 0; 1];
60         mag.magdir = [0; 0; 1];
61     else
62         mag.dir = mag.magdir;
63     end
64 else
65     if ~isfield(mag,'magdir')
66         mag.magdir = mag.dir;
67     end
68 end
69 end

71 switch mag.type
72 case 'cylinder'

74 % convert from current/turns to equiv magnetisation:
75     if ~isfield(mag,'magn')
76         if isfield(mag,'turns')&& isfield(mag,'current')
77             mag.magn = 4*pi*1e-7*mag.turns*mag.current/mag.dim(2);
78         end
79     end

81     if isfield(mag,'radius')&& isfield(mag,'height')
82         mag.dim = [mag.radius; mag.height];
83     end

85     if isfield(mag,'dim')
86         mag.volume = pi*mag.dim(1)^2*mag.dim(2);
87     end

89 case 'ring'

91     if isfield(mag,'radius')&& isfield(mag,'height')
92         mag.dim = [mag.radius(:); mag.height];
93     end

95     if isfield(mag,'dim')
96         if mag.dim(2)<= mag.dim(1)
97             error('Ring radii must be defined as [ri ro] with ro > ri.')
98         end
99         mag.volume = pi*(mag.dim(2)^2-mag.dim(1)^2)*mag.dim(3);
100     end

102 case 'cuboid'

104     if ~isfield(mag,'magdir')
105         warning('Magnet direction ("magdir")not specified; assuming +z.')
106         mag.magdir = [0; 0; 1];
107     else
108         mag.magdir = make_unit_vector(mag,'magdir');
109     end

111     if isfield(mag,'dim')

```

```

112     mag.volume = prod(mag.dim);
113 end
115 end
117 if isfield(mag,'magdir') && isfield(mag,'magn')
118     mag.magM = mag.magdir*mag.magn;
119     mag.dipolemoment = 1/(4*pi*1e-7)*mag.magM*mag.volume;
120 end
122 mag.fndefined = true;
124 end

```

#### 4.1.1 The grade2magn() function

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.

```

141 function magn = grade2magn(grade)
143 if isnumeric(grade)
144     magn = 2*sqrt(grade/100);
145 else
146     if strcmp(grade(1),'N')
147         grade = grade(2:end);
148     end
149     magn = 2*sqrt(str2double(grade)/100);
150 end
152 end

```

#### 4.1.2 The make\_unit\_vector() function

```

156 function vec = make_unit_vector(mag,vecname)

```

Magnetisation directions are specified in cartesian coordinates. Although they should be unit vectors, we don’t assume they are.

```

160 if ~isfield(mag,vecname)
161     vec = [0;0;0];
162     return
163 end
165 vec_in = mag.(vecname);
167 if isnumeric(vec_in)

```

```

169     if numel(vec_in)~= 3
170         error(['"',vecname,'" has wrong number of elements (should be 3x1 vector or string
input like ''+x''.')])
171     end
172     norm_vec_in = norm(vec_in);
173     if norm_vec_in < eps
174         norm_vec_in = 1; % to avoid 0/0
175     end
176     vec = vec_in(:)/norm_vec_in;
177 elseif ischar(vec_in)
178
179     switch vec_in
180     case 'x'; vec = [1;0;0];
181     case 'y'; vec = [0;1;0];
182     case 'z'; vec = [0;0;1];
183     case '+x'; vec = [1;0;0];
184     case '+y'; vec = [0;1;0];
185     case '+z'; vec = [0;0;1];
186     case '-x'; vec = [-1; 0; 0];
187     case '-y'; vec = [ 0;-1; 0];
188     case '-z'; vec = [ 0; 0;-1];
189     otherwise, error('Vector string %s not understood.',vec);
190     end
191
192 else
193     error('Strange input (this shouldn't happen)')
194 end
195
196 end

```

## 5 The magnetforces() function

```

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

439
440
441
442 magconst = 1/(4*pi*(4*pi*1e-7));
443
444 [index_i, index_j, index_k, index_l, index_p, index_q] = ndgrid([0 1]);
445
446 index_sum = (-1).^(index_i+index_j+index_k+index_l+index_p+index_q);

```

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`.

```

454 calc_force_bool    = false;
455 calc_stiffness_bool = false;
456 calc_torque_bool   = false;

457 for iii = 1:length(varargin)
458     switch varargin{iii}

```

```

460     case 'force',    calc_force_bool    = true;
461     case 'stiffness', calc_stiffness_bool = true;
462     case 'torque',    calc_torque_bool   = true;
463     case 'x', warning("Options 'x','y','z'are no longer supported.");
464     case 'y', warning("Options 'x','y','z'are no longer supported.");
465     case 'z', warning("Options 'x','y','z'are no longer supported.");
466     otherwise
467         error(['Unknown calculation option ',varargin{iii},'''])
468     end
469 end

471 if ~calc_force_bool && ~calc_stiffness_bool && ~calc_torque_bool
472     varargin{end+1} = 'force';
473     calc_force_bool = true;
474 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.

```

481 if size(displ,1)== 3
482     % all good
483 elseif size(displ,2)== 3
484     displ = transpose(displ);
485 else
486     error(['Displacements matrix should be of size (3, D)',...
487         'where D is the number of displacements.'])
488 end

490 Ndispl = size(displ,2);

492 if calc_force_bool
493     forces_out = nan([3 Ndispl]);
494 end

496 if calc_stiffness_bool
497     stiffnesses_out = nan([3 Ndispl]);
498 end

500 if calc_torque_bool
501     torques_out = nan([3 Ndispl]);
502 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.)

```

516 if ~isfield(magnet_fixed,'fdefined')
517     magnet_fixed = magnetdefine(magnet_fixed);
518 end
519 if ~isfield(magnet_float,'fdefined')
520     magnet_float = magnetdefine(magnet_float);
521 end

```

```

525 if strcmp(magnet_fixed.type, 'coil')
527     if ~strcmp(magnet_float.type, 'cylinder')
528         error('Coil/magnet forces can only be calculated for cylindrical magnets.')
529     end
531     coil = magnet_fixed;
532     magnet = magnet_float;
533     magtype = 'coil';
534     coil_sign = +1;
536 end
538 if strcmp(magnet_float.type, 'coil')
540     if ~strcmp(magnet_fixed.type, 'cylinder')
541         error('Coil/magnet forces can only be calculated for cylindrical magnets.')
542     end
544     coil = magnet_float;
545     magnet = magnet_fixed;
546     magtype = 'coil';
547     coil_sign = -1;
549 end
552 if ~strcmp(magnet_fixed.type, magnet_float.type)
553     error('Magnets must be of same type (cuboid/cuboid or cylinder/cylinder)')
554 end
555 magtype = magnet_fixed.type;
560 switch magtype
561     case 'cuboid'
563         size1 = magnet_fixed.dim(:)/2;
564         size2 = magnet_float.dim(:)/2;
566         swap_x_y = @(vec)vec([2 1 3],:);
567         swap_x_z = @(vec)vec([3 2 1],:);
568         swap_y_z = @(vec)vec([1 3 2],:);
570         rotate_z_to_x = @(vec)[ vec(3,:); vec(2,:); -vec(1,:) ] ; % Ry( 90)
571         rotate_x_to_z = @(vec)[ -vec(3,:); vec(2,:); vec(1,:) ] ; % Ry(-90)
573         rotate_y_to_z = @(vec)[ vec(1,:); -vec(3,:); vec(2,:) ] ; % Rx( 90)
574         rotate_z_to_y = @(vec)[ vec(1,:); vec(3,:); -vec(2,:) ] ; % Rx(-90)
576         rotate_x_to_y = @(vec)[ -vec(2,:); vec(1,:); vec(3,:) ] ; % Rz( 90)
577         rotate_y_to_x = @(vec)[ vec(2,:); -vec(1,:); vec(3,:) ] ; % Rz(-90)
579         size1_x = swap_x_z(size1);
580         size2_x = swap_x_z(size2);
581         J1_x = rotate_x_to_z(magnet_fixed.magM);
582         J2_x = rotate_x_to_z(magnet_float.magM);
584         size1_y = swap_y_z(size1);
585         size2_y = swap_y_z(size2);
586         J1_y = rotate_y_to_z(magnet_fixed.magM);

```

```

587     J2_y    = rotate_y_to_z(magnet_float.magM);
588
589     if calc_force_bool
590         for iii = 1:Ndispl
591             forces_out(:,iii)= single_magnet_force(displ(:,iii));
592         end
593     end
594
595     if calc_stiffness_bool
596         for iii = 1:Ndispl
597             stiffnesses_out(:,iii)= single_magnet_stiffness(displ(:,iii));
598         end
599     end
600
601     if calc_torque_bool
602         torques_out = single_magnet_torque(displ,magnet_float.lever);
603     end
604
605     case 'cylinder'
606
607         if any(abs(magnet_fixed.dir)~= abs(magnet_float.dir))
608             error('Cylindrical magnets must be oriented in the same direction')
609         end
610         if any(abs(magnet_fixed.magdir)~= abs(magnet_float.magdir))
611             error('Cylindrical magnets must be oriented in the same direction')
612         end
613         if any(abs(magnet_fixed.dir)~= abs(magnet_fixed.magdir))
614             error('Cylindrical magnets must be magnetised in the same direction as their orientation
615 ')
616         end
617         if any(abs(magnet_float.dir)~= abs(magnet_float.magdir))
618             error('Cylindrical magnets must be magnetised in the same direction as their orientation
619 ')
620         end
621
622         cyldir    = find(magnet_float.magdir ~= 0);
623         cylnotdir = find(magnet_float.magdir == 0);
624         if length(cyldir)~= 1
625             error('Cylindrical magnets must be aligned in one of the x, y or z directions')
626         end
627
628         if calc_force_bool
629             for iii = 1:Ndispl
630                 forces_out(:,iii)= single_magnet_cyl_force(displ(:,iii));
631             end
632         end
633
634         if calc_stiffness_bool
635             error('Stiffness cannot be calculated for cylindrical magnets yet.')
636         end
637
638         if calc_torque_bool
639             error('Torques cannot be calculated for cylindrical magnets yet.')
640         end
641
642     case 'ring'
643
644         cyldir    = find(magnet_float.magdir ~= 0);

```

```

643     cylnotdir = find(magnet_float.magdir == 0);
644     if length(cyldir)~= 1
645         error('Cylindrical magnets must be aligned in one of the x, y or z directions')
646     end
647
648     if calc_force_bool
649         for iii = 1:Ndispl
650             forces_out(:,iii)= single_magnet_ring_force(displ(:,iii));
651         end
652     end
653
654     case 'coil'
655
656         warning('Code for coils in Matlab has never been completed :( See the Mathematica
code for more details')!
657         for iii = 1:Ndispl
658             forces_out(:,iii)= coil_sign*coil.dir*...
659                 forces_magcyl_shell_calc(...
660                     magnet.dim, ...
661                     coil.dim, ...
662                     squeeze(displ(cyldir,:)), ...
663                     magnet.magM(cyldir), ...
664                     coil.current, ...
665                     coil.turns);
666         end
667     end
668 end
669 end

```

After all of the calculations have occurred, they're placed back into `varargout`. Outputs are ordered in the same order as the inputs are specified, which makes the code a bit uglier but is presumably a bit nicer for the user and/or just a bit more flexible.

```

682 argcount = 0;
683
684 for iii = 1:length(varargin)
685     switch varargin{iii}
686         case 'force',    argcount = argcount+1;
687         case 'stiffness', argcount = argcount+1;
688         case 'torque',   argcount = argcount+1;
689     end
690 end
691
692 varargout = cell(argcount,1);
693
694 argcount = 0;
695
696 for iii = 1:length(varargin)
697     switch varargin{iii}
698         case 'force',    argcount = argcount+1; varargout{argcount} = forces_out;
699         case 'stiffness', argcount = argcount+1; varargout{argcount} = stiffnesses_out;
700         case 'torque',   argcount = argcount+1; varargout{argcount} = torques_out;
701     end
702 end

```

That is the end of the main function.

## 5.1 The single\_magnet\_cyl\_force() function

```
711 function forces_out = single_magnet_cyl_force(displ)
713     forces_out = nan(size(displ));
715     ecc = sqrt(sum(displ(cylnotdir).^2));
717     if ecc < eps
718         magdir = [0;0;0];
719         magdir(cyldir)= 1;
720         forces_out = magdir*cylinder_force_coaxial(magnet_fixed.magM(cyldir), magnet_float
721 .magM(cyldir), magnet_fixed.dim(1), magnet_float.dim(1), magnet_fixed.dim(2), magnet_float
722 .dim(2), displ(cyldir)).';
723     else
724         ecc_forces = cylinder_force_eccentric(magnet_fixed.dim, magnet_float.dim, displ(
725 cyldir), ecc, magnet_fixed.magM(cyldir), magnet_float.magM(cyldir)).';
726         forces_out(cyldir)= ecc_forces(2);
727         forces_out(cylnotdir(1))= displ(cylnotdir(1))/ecc*ecc_forces(1);
728         forces_out(cylnotdir(2))= displ(cylnotdir(2))/ecc*ecc_forces(1);
729 % Need to check this division into components is correct...
730     end
731 end
```

## 5.2 The single\_magnet\_ring\_force() function

```
733 function forces_out = single_magnet_ring_force(displ)
735     forces_out = nan(size(displ));
737     ecc = sqrt(sum(displ(cylnotdir).^2));
739     if ecc < eps
740         magdir = [0;0;0];
741         magdir(cyldir)= 1;
742         forces11 = magdir*cylinder_force_coaxial(-magnet_fixed.magM(cyldir), -magnet_float
743 .magM(cyldir), magnet_fixed.dim(1), magnet_float.dim(1), magnet_fixed.dim(3), magnet_float
744 .dim(3), displ(cyldir)).';
745         forces12 = magdir*cylinder_force_coaxial(-magnet_fixed.magM(cyldir), +magnet_float
746 .magM(cyldir), magnet_fixed.dim(1), magnet_float.dim(2), magnet_fixed.dim(3), magnet_float
747 .dim(3), displ(cyldir)).';
748         forces21 = magdir*cylinder_force_coaxial(+magnet_fixed.magM(cyldir), -magnet_float
749 .magM(cyldir), magnet_fixed.dim(2), magnet_float.dim(1), magnet_fixed.dim(3), magnet_float
750 .dim(3), displ(cyldir)).';
751         forces22 = magdir*cylinder_force_coaxial(+magnet_fixed.magM(cyldir), +magnet_float
752 .magM(cyldir), magnet_fixed.dim(2), magnet_float.dim(2), magnet_fixed.dim(3), magnet_float
753 .dim(3), displ(cyldir)).';
754         forces_out = forces11 + forces12 + forces21 + forces22;
755     else
756         ecc_forces = cylinder_force_eccentric(magnet_fixed.dim, magnet_float.dim, displ(
757 cyldir), ecc, magnet_fixed.magM(cyldir), magnet_float.magM(cyldir)).';
758         forces_out(cyldir)= ecc_forces(2);
759         forces_out(cylnotdir(1))= displ(cylnotdir(1))/ecc*ecc_forces(1);
```



```

751     forces_out(cylnotdir(2))= displ(cylnotdir(2))/ecc*ecc_forces(1);
752 % Need to check this division into components is correct...
753     end
754
755 end

```

### 5.3 The single\_magnet\_force() function

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.

```

767 function force_out = single_magnet_force(displ)
768
769     force_components = nan([9 3]);
770
771     d_x = rotate_x_to_z(displ);
772     d_y = rotate_y_to_z(displ);
773
774     force_components(1,:)= ...
775         rotate_z_to_x( cuboid_force_z_z(size1_x,size2_x,d_x,J1_x,J2_x));
776
777     force_components(2,:)= ...
778         rotate_z_to_x( cuboid_force_z_y(size1_x,size2_x,d_x,J1_x,J2_x));
779
780     force_components(3,:)= ...
781         rotate_z_to_x( cuboid_force_z_x(size1_x,size2_x,d_x,J1_x,J2_x));
782
783     force_components(4,:)= ...
784         rotate_z_to_y( cuboid_force_z_x(size1_y,size2_y,d_y,J1_y,J2_y));
785
786     force_components(5,:)= ...
787         rotate_z_to_y( cuboid_force_z_z(size1_y,size2_y,d_y,J1_y,J2_y));
788
789     force_components(6,:)= ...
790         rotate_z_to_y( cuboid_force_z_y(size1_y,size2_y,d_y,J1_y,J2_y));
791
792     force_components(9,:)= cuboid_force_z_z( size1,size2,displ,magnet_fixed.magM,magnet_float
793         .magM );
794
795     force_components(8,:)= cuboid_force_z_y( size1,size2,displ,magnet_fixed.magM,magnet_float
796         .magM );
797
798     force_components(7,:)= cuboid_force_z_x( size1,size2,displ,magnet_fixed.magM,magnet_float
799         .magM );
800
801     force_out = sum(force_components);
802
803 end

```

## 5.4 The single\_magnet\_torque() function

For the magnetforces code we always assume the first magnet is fixed. But the Janssen code assumes the torque is calculated on the first magnet and defines the lever arm for that first magnet. Therefore we need to flip the definitions a bit.

```
812 function torques_out = single_magnet_torque(displ,lever)
814     torque_components = nan([size(displ)9]);
816     d_x = rotate_x_to_z(displ);
817     d_y = rotate_y_to_z(displ);
818     d_z = displ;
820     l_x = rotate_x_to_z(lever);
821     l_y = rotate_y_to_z(lever);
822     l_z = lever;
824     torque_components(:, :, 9) = cuboid_torque_z_z( size1, size2, d_z, l_z, magnet_fixed.magM
, magnet_float.magM );
826     torque_components(:, :, 8) = cuboid_torque_z_y( size1, size2, d_z, l_z, magnet_fixed.magM
, magnet_float.magM );
828     torque_components(:, :, 7) = torques_calc_z_x( size1, size2, d_z, l_z, magnet_fixed.magM
, magnet_float.magM );
830     torque_components(:, :, 1) = ...
831         rotate_z_to_x( cuboid_torque_z_z(size1_x, size2_x, d_x, l_x, J1_x, J2_x));
833     torque_components(:, :, 2) = ...
834         rotate_z_to_x( cuboid_torque_z_y(size1_x, size2_x, d_x, l_x, J1_x, J2_x));
836     torque_components(:, :, 3) = ...
837         rotate_z_to_x( torques_calc_z_x(size1_x, size2_x, d_x, l_x, J1_x, J2_x));
839     torque_components(:, :, 4) = ...
840         rotate_z_to_y( torques_calc_z_x(size1_y, size2_y, d_y, l_y, J1_y, J2_y));
842     torque_components(:, :, 5) = ...
843         rotate_z_to_y( cuboid_torque_z_z(size1_y, size2_y, d_y, l_y, J1_y, J2_y));
845     torque_components(:, :, 6) = ...
846         rotate_z_to_y( cuboid_torque_z_y(size1_y, size2_y, d_y, l_y, J1_y, J2_y));
848     torques_out = sum(torque_components, 3);
849 end
```

## 5.5 The single\_magnet\_stiffness() function

```
856 function stiffness_out = single_magnet_stiffness(displ)
858     stiffness_components = nan([9 3]);
860     d_x = rotate_x_to_z(displ);
861     d_y = rotate_y_to_z(displ);
863     stiffness_components(7, :) = ...
```

```

864     stiffnesses_calc_z_x( size1,size2,displ,magnet_fixed.magM,magnet_float.magM );
866     stiffness_components(8,:)= ...
867     stiffnesses_calc_z_y( size1,size2,displ,magnet_fixed.magM,magnet_float.magM );
869     stiffness_components(9,:)= ...
870     stiffnesses_calc_z_z( size1,size2,displ,magnet_fixed.magM,magnet_float.magM );
872     stiffness_components(1,:)= ...
873     swap_x_z( stiffnesses_calc_z_z( size1_x,size2_x,d_x,J1_x,J2_x ));
875     stiffness_components(2,:)= ...
876     swap_x_z( stiffnesses_calc_z_y( size1_x,size2_x,d_x,J1_x,J2_x ));
878     stiffness_components(3,:)= ...
879     swap_x_z( stiffnesses_calc_z_x( size1_x,size2_x,d_x,J1_x,J2_x ));
881     stiffness_components(4,:)= ...
882     swap_y_z( stiffnesses_calc_z_x( size1_y,size2_y,d_y,J1_y,J2_y ));
884     stiffness_components(5,:)= ...
885     swap_y_z( stiffnesses_calc_z_z( size1_y,size2_y,d_y,J1_y,J2_y ));
887     stiffness_components(6,:)= ...
888     swap_y_z( stiffnesses_calc_z_y( size1_y,size2_y,d_y,J1_y,J2_y ));
890     stiffness_out = sum(stiffness_components);
891 end

```

## 5.6 The stiffnesses\_calc\_z\_z() function

```

901 function calc_out = stiffnesses_calc_z_z(size1,size2,offset,J1,J2)
903     J1 = J1(3);
904     J2 = J2(3);
906
907     if (J1==0 || J2==0)
908         calc_out = [0; 0; 0];
909         return;
910     end
912     u = offset(1)+ size2(1)*(-1).^index_j - size1(1)*(-1).^index_i;
913     v = offset(2)+ size2(2)*(-1).^index_l - size1(2)*(-1).^index_k;
914     w = offset(3)+ size2(3)*(-1).^index_q - size1(3)*(-1).^index_p;
915     r = sqrt(u.^2+v.^2+w.^2);
917     component_x = - r - (u.^2 .*v)./(u.^2+w.^2)- v.*log(r-v);
919     component_y = - r - (v.^2 .*u)./(v.^2+w.^2)- u.*log(r-u);
921     component_z = - component_x - component_y;
923     component_x = index_sum.*component_x;
924     component_y = index_sum.*component_y;
925     component_z = index_sum.*component_z;

```

```

927     calc_out = J1*J2*magconst .* ...
928         [ sum(component_x(:));
929           sum(component_y(:));
930           sum(component_z(:))] ;
932 end

```

## 5.7 The stiffnesses\_calc\_z\_y() function

```

938 function calc_out = stiffnesses_calc_z_y(size1,size2,offset,J1,J2)
940     J1 = J1(3);
941     J2 = J2(2);
944     if (J1==0 || J2==0)
945         calc_out = [0; 0; 0];
946         return;
947     end
949     u = offset(1)+ size2(1)*(-1).^index_j - size1(1)*(-1).^index_i;
950     v = offset(2)+ size2(2)*(-1).^index_l - size1(2)*(-1).^index_k;
951     w = offset(3)+ size2(3)*(-1).^index_q - size1(3)*(-1).^index_p;
952     r = sqrt(u.^2+v.^2+w.^2);
954     component_x = ((u.^2 .*v)./(u.^2 + v.^2))+ (u.^2 .*w)./(u.^2 + w.^2)...
955         - u.*atan1(v.*w,r.*u)+ multiply_x_log_y( w , r + v )+ ...
956         + multiply_x_log_y( v , r + w );
957     component_y = - v/2 + (u.^2 .*v)./(u.^2 + v.^2)- (u.*v.*w)./(v.^2 + w.^2)...
958         - u.*atan1(u.*w,r.*v)- multiply_x_log_y( v , r + w );
959     component_z = - component_x - component_y;
961     component_x = index_sum.*component_x;
962     component_y = index_sum.*component_y;
963     component_z = index_sum.*component_z;
965     calc_out = J1*J2*magconst .* ...
966         [ sum(component_x(:));
967           sum(component_y(:));
968           sum(component_z(:))] ;

```

### 5.7.1 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.

### 5.7.2 The multiply\_x\_log\_y() function

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

```

980 function out = multiply_x_log_y(x,y)
981     out = x.*log(y);
982     out(~isfinite(out))=0;
983 end

```

### 5.7.3 The atan1() function

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.

```
990     function out = atan1(x,y)
991         out = zeros(size(x));
992         ind = x~=0 & y~=0;
993         out(ind)= atan(x(ind)./y(ind));
994     end
997 end
```

## 5.8 The stiffnesses\_calc\_z\_x() function

```
1003 function calc_out = stiffnesses_calc_z_x(size1,size2,offset,J1,J2)
1005     stiffnesses_xyz = stiffnesses_calc_z_y(...
1006         swap_x_y(size1), swap_x_y(size2), rotate_x_to_y(offset),...
1007         J1, rotate_x_to_y(J2));
1009     calc_out = swap_x_y(stiffnesses_xyz);
1011 end
```

## 5.9 The torques\_calc\_z\_y() function

```
1016 function calc_out = torques_calc_z_y(size1,size2,offset,lever,J1,J2)
1018     if J1(3)~=0 && J2(2)~=0
1019         error('Torques cannot be calculated for orthogonal magnets yet.')
1020     end
1022     calc_out = 0*offset;
1024 end
```

## 5.10 The torques\_calc\_z\_x() function

```
1028 function calc_out = torques_calc_z_x(size1,size2,offset,lever,J1,J2)
1030     if J1(3)~=0 && J2(1)~=0
1031         error('Torques cannot be calculated for orthogonal magnets yet.')
1032     end
1034     calc_out = 0*offset;
1036 end
```

### 5.11 The forces\_magcyl\_shell\_calc() function

```
1041 function Fz = forces_magcyl_shell_calc(magsize,coilsize,displ,Jmag,Nrz,I)
1043     Jcoil = 4*pi*1e-7*Nrz(2)*I/coil.dim(3);
1045     shell_forces = nan([length(displ)Nrz(1)]);
1047     for rr = 1:Nrz(1)
1049         this_radius = coilsize(1)+(rr-1)/(Nrz(1)-1)*(coilsize(2)-coilsize(1));
1050         shell_size = [this_radius, coilsize(3)];
1052         shell_forces(:,rr)= cylinder_force_coaxial(magsize,shell_size,displ,Jmag,Jcoil);

1054     end
1056     Fz = sum(shell_forces,2);
1058 end

1062 end
```

## 6 Magnet interactions

The functions described in this section are translations of specific cases from the literature. They have been written to be somewhat self-contained from the main code so they can be used directly for translation into other programming languages, or in applications where speed is important (such as for dynamic simulations).

### 6.1 The cuboid\_force\_z\_z() function

The expressions here follow directly from [akoun1984](#).

Inputs:	size1=(a,b,c)	the half dimensions of the fixed magnet
	size2=(A,B,C)	the half dimensions of the floating magnet
	offset=(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

```
1086 function forces_xyz = cuboid_force_z_z(size1,size2,offset,J1,J2)
1088 magconst = 1/(4*pi*(4*pi*1e-7));
1090 J1 = J1(3);
1091 J2 = J2(3);
1093 if ( abs(J1)<eps || abs(J2)<eps )
1094     forces_xyz = [0; 0; 0];
1095     return;
1096 end
1098 component_x = 0;
1099 component_y = 0;
```

```

1100 component_z = 0;
1102 for ii = [1 -1]
1103     for jj = [1 -1]
1104         for kk = [1 -1]
1105             for ll = [1 -1]
1106                 for pp = [1 -1]
1107                     for qq = [1 -1]
1109                         u = offset(1)+ size2(1)*jj - size1(1)*ii;
1110                         v = offset(2)+ size2(2)*ll - size1(2)*kk;
1111                         w = offset(3)+ size2(3)*qq - size1(3)*pp;
1112                         r = sqrt(u.^2+v.^2+w.^2);
1114                         if w == 0
1115                             atan_term = 0;
1116                         else
1117                             atan_term = atan(u.*v./(r.*w));
1118                         end
1119                         if abs(r-u)< eps
1120                             log_ru = 0;
1121                         else
1122                             log_ru = log(r-u);
1123                         end
1124                         if abs(r-v)< eps
1125                             log_rv = 0;
1126                         else
1127                             log_rv = log(r-v);
1128                         end
1130                         cx = ...
1131                             + 0.5*(v.^2-w.^2).*log_ru ...
1132                             + u.*v.*log_rv ...
1133                             + v.*w.*atan_term...
1134                             + 0.5*r.*u;
1136                         cy = ...
1137                             + 0.5*(u.^2-w.^2).*log_rv ...
1138                             + u.*v.*log_ru ...
1139                             + u.*w.*atan_term ...
1140                             + 0.5*r.*v;
1142                         cz = ...
1143                             - u.*w.*log_ru ...
1144                             - v.*w.*log_rv ...
1145                             + u.*v.*atan_term ...
1146                             - r.*w;
1148                         ind_sum = ii*jj*kk*ll*pp*qq;
1149                         component_x = component_x + ind_sum.*cx;
1150                         component_y = component_y + ind_sum.*cy;
1151                         component_z = component_z + ind_sum.*cz;
1153                     end
1154                 end
1155             end

```

```

1156     end
1157 end
1158 end

1160 forces_xyz = J1*J2*magconst.*[component_x; component_y; component_z];

1162 end

```

## 6.2 The cuboid\_force\_z\_y() function

Orthogonal magnets forces given by **yonnet2009-ldia**. 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.

Inputs:	size1=(a,b,c)	the half dimensions of the fixed magnet
	size2=(A,B,C)	the half dimensions of the floating magnet
	offset=(dx,dy,dz)	distance between magnet centres
	(J1,J2)	magnetisation vectors of the magnets
Outputs:	forces_xyz=(Fx,Fy,Fz)	Forces of the second magnet

```

1191 function forces_xyz = cuboid_force_z_y(size1,size2,offset,J1,J2)

1193     J1 = J1(3);
1194     J2 = J2(2);

1196     if ( abs(J1)<eps || abs(J2)<eps )
1197         forces_xyz = [0; 0; 0];
1198         return;
1199     end

1201     component_x = 0;
1202     component_y = 0;
1203     component_z = 0;

1205     for ii = [1 -1]
1206         for jj = [1 -1]
1207             for kk = [1 -1]
1208                 for ll = [1 -1]
1209                     for pp = [1 -1]
1210                         for qq = [1 -1]

1212                             ind_sum = ii*jj*kk*ll*pp*qq;

1214                             u = offset(1)+ size2(1)*jj - size1(1)*ii;
1215                             v = offset(2)+ size2(2)*ll - size1(2)*kk;
1216                             w = offset(3)+ size2(3)*qq - size1(3)*pp;
1217                             r = sqrt(u.^2+v.^2+w.^2);

1219                             if u == 0
1220                                 atan_term_u = 0;
1221                             else
1222                                 atan_term_u = atan(v.*w./(r.*u));
1223                             end

```



```

1224     if v == 0
1225         atan_term_v = 0;
1226     else
1227         atan_term_v = atan(u.*w./(r.*v));
1228     end
1229     if w == 0
1230         atan_term_w = 0;
1231     else
1232         atan_term_w = atan(u.*v./(r.*w));
1233     end
1234
1235     if abs(r-u)< eps
1236         log_ru = 0;
1237     else
1238         log_ru = log(r-u);
1239     end
1240     if abs(r+w)< eps
1241         log_rw = 0;
1242     else
1243         log_rw = log(r+w);
1244     end
1245     if abs(r+v)< eps
1246         log_rv = 0;
1247     else
1248         log_rv = log(r+v);
1249     end
1250
1251     cx = ...
1252         + v.*w.*log_ru ...
1253         - v.*u.*log_rw ...
1254         - u.*w.*log_rv ...
1255         + 0.5*u.^2.*atan_term_u ...
1256         + 0.5*v.^2.*atan_term_v ...
1257         + 0.5*w.^2.*atan_term_w;
1258
1259     cy = ...
1260         - 0.5*(u.^2-v.^2).*log_rw ...
1261         + u.*w.*log_ru ...
1262         + u.*v.*atan_term_v ...
1263         + 0.5*w.*r;
1264
1265     cz = ...
1266         - 0.5*(u.^2-w.^2).*log_rv ...
1267         + u.*v.*log_ru ...
1268         + u.*w.*atan_term_w ...
1269         + 0.5*v.* r;
1270
1271     component_x = component_x + ind_sum.*cx;
1272     component_y = component_y + ind_sum.*cy;
1273     component_z = component_z + ind_sum.*cz;
1274
1275     end
1276 end
1277 end
1278 end

```

```

1279     end
1280 end

1282     forces_xyz = J1*J2/(4*pi*(4*pi*1e-7))*[ component_x; component_y; component_z ];

1284 end

```

### 6.3 The cuboid\_force\_z\_x() function

This is a translation of cuboid\_force\_z\_y into a rotated coordinate system.

Inputs:	size1=(a,b,c)	the half dimensions of the fixed magnet
	size2=(A,B,C)	the half dimensions of the floating magnet
	offset=(dx,dy,dz)	distance between magnet centres
	(J1,J2)	magnetisation vectors of the magnets
Outputs:	forces_xyz=(Fx,Fy,Fz)	Forces of the second magnet

```

1310 function forces_xyz = cuboid_force_z_x(size1,size2,offset,J1,J2)

1312 J1 = J1(3);
1313 J2 = J2(1);

1315 if ( abs(J1)<eps || abs(J2)<eps )
1316     forces_xyz = [0; 0; 0];
1317     return;
1318 end

1320 component_x = 0;
1321 component_y = 0;
1322 component_z = 0;

1324 for ii = [1 -1]
1325     for jj = [1 -1]
1326         for kk = [1 -1]
1327             for ll = [1 -1]
1328                 for pp = [1 -1]
1329                     for qq = [1 -1]

1331                         ind_sum = ii*jj*kk*ll*pp*qq;

1333                         u = -offset(2)- size2(2)*jj + size1(2)*ii;
1334                         v = offset(1)+ size2(1)*ll - size1(1)*kk;
1335                         w = offset(3)+ size2(3)*qq - size1(3)*pp;
1336                         r = sqrt(u.^2+v.^2+w.^2);

1338                         if u == 0
1339                             atan_term_u = 0;
1340                         else
1341                             atan_term_u = atan(v.*w./(r.*u));
1342                         end
1343                         if v == 0
1344                             atan_term_v = 0;
1345                         else

```

```

1346         atan_term_v = atan(u.*w./(r.*v));
1347     end
1348     if w == 0
1349         atan_term_w = 0;
1350     else
1351         atan_term_w = atan(u.*v./(r.*w));
1352     end
1353
1354     if abs(r-u)< eps
1355         log_ru = 0;
1356     else
1357         log_ru = log(r-u);
1358     end
1359     if abs(r+w)< eps
1360         log_rw = 0;
1361     else
1362         log_rw = log(r+w);
1363     end
1364     if abs(r+v)< eps
1365         log_rv = 0;
1366     else
1367         log_rv = log(r+v);
1368     end
1369
1370     cx = ...
1371         + v.*w.*log_ru ...
1372         - v.*u.*log_rw ...
1373         - u.*w.*log_rv ...
1374         + 0.5*u.^2.*atan_term_u ...
1375         + 0.5*v.^2.*atan_term_v ...
1376         + 0.5*w.^2.*atan_term_w;
1377
1378     cy = ...
1379         - 0.5*(u.^2-v.^2).*log_rw ...
1380         + u.*w.*log_ru ...
1381         + u.*v.*atan_term_v ...
1382         + 0.5*w.*r;
1383
1384     cz = ...
1385         - 0.5*(u.^2-w.^2).*log_rv ...
1386         + u.*v.*log_ru ...
1387         + u.*w.*atan_term_w ...
1388         + 0.5*v.* r;
1389
1390     component_x = component_x + ind_sum.*cx;
1391     component_y = component_y + ind_sum.*cy;
1392     component_z = component_z + ind_sum.*cz;
1393
1394     end
1395     end
1396     end
1397     end
1398     end
1399 end

```

```

1401 forces_xyz = J1*J2/(4*pi*(4*pi*1e-7))*[ component_y; -component_x; component_z ];
1403 end

```

## 6.4 The cuboid\_torque\_z\_z() function

The expressions here follow directly from **janssen2010-ietm**. 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.

Note that despite this verification this code produces results which are inconsistent with the graph in the **janssen2010-ietm** paper. This appears to have been an oversight in the publication.

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

```

1441 function torque_zz = cuboid_torque_z_z(size1,size2,offset,lever,J1,J2)
1443 br1 = J1(3);
1444 br2 = J2(3);
1446 if br1==0 || br2==0
1447     torque_zz = 0*offset;
1448     return
1449 end
1451 Txyz = zeros([3, size(offset,2)]);
1453 for ii=[0,1]
1454     for jj=[0,1]
1455         for kk=[0,1]
1456             for ll=[0,1]
1457                 for mm=[0,1]
1458                     for nn=[0,1]
1460                         Cu = (-1)^ii.*size1(1)- offset(1,:)- lever(1,:);
1461                         Cv = (-1)^kk.*size1(2)- offset(2,:)- lever(2,:);
1462                         Cw = (-1)^mm.*size1(3)- offset(3,:)- lever(3,:);
1464                         u = offset(1,:)- (-1)^ii.*size1(1)+ (-1)^jj.*size2(1);
1465                         v = offset(2,:)- (-1)^kk.*size1(2)+ (-1)^ll.*size2(2);
1466                         w = offset(3,:)- (-1)^mm.*size1(3)+ (-1)^nn.*size2(3);
1468                         Cuu = 2*Cu + u;
1469                         Cvv = 2*Cv + v;
1470                         Cww = 2*Cw + w;
1472                         u2 = u.^2;
1473                         v2 = v.^2;

```

```

1474     w2 = w.^2;
1475     s2 = u2+v2+w2;
1476     s = sqrt(s2);

1478 % find indexes where cuboid faces align
1479     a = (u2<eps)& (v2<eps);
1480     b = (u2<eps)& (w2<eps);
1481     c = (v2<eps)& (w2<eps);

1483 % and all those that do not
1484     d = ~a & ~b & ~c;

1486     Ex = nan(1,size(offset,2));
1487     Ey = nan(1,size(offset,2));
1488     Ez = nan(1,size(offset,2));

1490     if any(a)
1491         Ex(a) = 1/8*w(a).*(-w2(a)-2*Cw(a).*w(a)-8*Cv(a).*abs(w(a))+w(a).*Cww(a).*
log(w2(a)));
1492         Ey(a) = 1/8*w(a).*(+w2(a)+2*Cw(a).*w(a)+8*Cu(a).*abs(w(a))-w(a).*Cww(a).*
log(w2(a)));
1493         Ez(a) = 1/4*(Cu(a)-Cv(a))*w2(a).*log(w2(a));
1494     end

1496     if any(b)
1497         Ex(b) = -1/4*Cw(b).*v(b).*(v(b)+2*abs(v(b)));
1498         Ey(b) = -1/4*Cw(b).*v2(b).*(log(v2(b))-1);
1499         Ez(b) = 1/72*v(b).*(2*v2(b)+36*Cu(b).*abs(v(b))+9*v(b).*Cvv(b).*log(v2(b)
)));
1500     end

1502     if any(c)
1503         Ex(c) = 1/4*Cw(c).*u2(c).*(log(u2(c))-1);
1504         Ey(c) = 1/4*Cw(c).*(u2(c)+2*abs(u(c)).*u(c));
1505         Ez(c) = -1/72*u(c).*(2*u2(c)+36*Cv(c).*abs(u(c))+9*u(c).*Cuu(c).*log(u2(c)
)));
1506     end

1508     if any(d)
1510         Ex(d) = 1/8.*( ...
1511             - Cww(d).*s2(d)+ ...
1512             - 2.*s(d).*(v(d).*Cww(d)+2.*Cvv(d).*w(d))+ ...
1513             ...
1514             + 2.*Cww(d).*(s2(d)-v2(d)).*log(s(d)+v(d))+ ...
1515             - 8.*u(d).*v(d).*(Cw(d)+w(d)).*log(s(d)-u(d))+ ...
1516             ...
1517             + 8.*Cv(d).*u(d).*w(d).*acoth(s(d)./u(d))+ ...
1518             + 4.*w(d).*(v(d).*Cvv(d)-w(d).*Cww(d)).*acoth(s(d)./v(d))+ ...
1519             + 4.*u(d).*(v(d).*Cvv(d)-w(d).*Cww(d)).*atan(u(d).*v(d)./(w(d).*s(d)))+
...
1520             0);

1522         Ey(d) = 1/8.*( ...
1523             + Cww(d).*s2(d)+ ...
1524             + 2.*s(d).*(u(d).*Cww(d)+2.*Cuu(d).*w(d)).*...

```

```

1525     ...
1526     - 2.*Cww(d).*(s2(d)-u2(d)).*log(s(d)+u(d))+ ...
1527     + 8.*u(d).*v(d).*(Cw(d)+w(d)).*(log(s(d)-v(d))-1)+ ...
1528     ...
1529     - 8.*Cu(d).*v(d).*w(d).*acoth(s(d)./v(d))+ ...
1530     - 4.*w(d).*(u(d).*Cuu(d)-w(d).*Cww(d)).*acoth(s(d)./u(d))+ ...
1531     - 4.*v(d).*(u(d).*Cuu(d)-w(d).*Cww(d)).*atan(u(d).*v(d)./(w(d).*s(d)))+
    ...
1532     ...
1533     + 8.*v(d).*w(d).*(Cw(d)+w(d)).*atan(u(d)./w(d))+ ...
1534     0);
1535
1536     Ez(d) = 1/36.*( ...
1537     - u(d).^3 + ...
1538     + v(d).^3 + ...
1539     + 6.*w2(d).*(v(d)-u(d))+ ...
1540     + 18.*s(d).*(Cu(d).*v(d)-u(d).*Cv(d))+ ...
1541     ...
1542     + 18.*u(d).*v(d).*( Cuu(d).*(log(s(d)-u(d))-1)- Cv(v(d).*(log(s(d)-v(d))
-1))+ ...
1543     + 9.*Cvv(d).*(v2(d)-w2(d)).*log(s(d)+u(d))+ ...
1544     - 9.*Cuu(d).*(u2(d)-w2(d)).*log(s(d)+v(d)) ...
1545     ...
1546     + 6.*w(d).*(w2(d)-3.*v(d).*Cvv(d)).*atan(u(d)./w(d))+ ...
1547     - 6.*w(d).*(w2(d)-3.*u(d).*Cuu(d)).*atan(v(d)./w(d))+ ...
1548     - 18.*w(d).*(Cvv(d).*v(d)-u(d).*Cuu(d)).*atan(u(d).*v(d)./(w(d).*s(d)))
+ ...
1549     0);
1550
1551     end
1552
1553     Txyz = Txyz + (-1)^(ii+jj+kk+ll+mm+nn)*[Ex; Ey; Ez];
1554
1555     end
1556     end
1557     end
1558     end
1559     end
1560     end
1561
1562     torque_zz = Txyz.*br1*br2/(16*pi^2*1e-7);
1563
1564     end

```

```

1573 function torque_zy = cuboid_torque_z_y(size1, size2, offset, lever, J1, J2)

```

cuboid\_torque\_z\_y calculates the torque on a cuboid magnet in the presence of another cuboid magnet, using theory described in Janssen 2011 this code assumes magnet 1 is magnetised along the z-axis and magnet 2 is magnetised along the y-axis Inputs size1 = [a1; b1; c1] - half-dimensions of magnet 1 in x, y and z directions size2 = [a2; b2; c2] - half-dimensions of magnet 2 in x, y and z directions offset = [alpha; beta; gamma] - vector from centre of magnet 1 to centre of magnet 2 lever = [delta; epsilon; zeta] - vector from centre of magnet 1 to torque reference point J1 - flux density of magnet 1 J2 - flux density of magnet 2 Outputs torque\_zy = [Tx; Ty; Tz] - torques on magnet 2 in x, y and z directions 6/12 Sean McGowan a1705690

```

    remanent flux density
1595 bzt1 = J1(3);
1596 bzt2 = J2(2);

    if the remanent flux densities along z axis for magnet 1 and y axis for magnet 2 are 0, torque will
    be 0
1600 if abs(bzt1)< eps || abs(bzt2)< eps
1601     torque_zy = zeros(size(offset));
1602     return
1603 end

    preallocate sums as rows of zero with the same length as the offset array
1606 Tx = zeros([1, size(offset,2)]);
1607 Ty = Tx;
1608 Tz = Tx;

    calculate sums as described in Janssen, 2011
1611 for ii = 0:1
1612     for jj = 0:1
1613         for kk = 0:1
1614             for ll = 0:1
1615                 for mm = 0:1
1616                     for nn = 0:1

1618                         Ex = nan(size(Tx));
1619                         Ey = Ex;
1620                         Ez = Ex;

1622                         Cu = ((-1)^ii).*(size1(1))-offset(1,:)-lever(1,:);
1623                         Cv = ((-1)^kk).*(size1(2))-offset(2,:)-lever(2,:);
1624                         Cw = ((-1)^mm).*(size1(3))-offset(3,:)-lever(3,:);

1626                         u = offset(1,:)-((-1)^ii).*(size1(1))+((-1)^jj).*(size2(1));
1627                         v = offset(2,:)-((-1)^kk).*(size1(2))+((-1)^ll).*(size2(2));
1628                         w = offset(3,:)-((-1)^mm).*(size1(3))+((-1)^nn).*(size2(3));

1630                         u2 = u.^2;
1631                         v2 = v.^2;
1632                         w2 = w.^2;
1633                         r2 = u2+v2+w2;
1634                         r = sqrt(r2);

1636 % find indexes where cuboid magnets align
1637     a = (u2<eps)& (v2<eps);
1638     b = (u2<eps)& (w2<eps);
1639     c = (v2<eps)& (w2<eps);

1641 % find indexes where cuboid magnets do not align
1642     d = ~a & ~b & ~c;

1644 % if magnets are aligned in any two directions, use the following limit
1645 % expressions to calculate the sums

1647     if any(a)
1648         Ex(a) = -1/24.*w2(a).*(...

```

```

1649         + 6.*Cw(a).*(1+2.*sign(w(a)))+ ...
1650         + 8.*abs(w(a))...
1651         );
1652     Ey(a) = pi/4.*w2(a).*sign(w(a)).*Cw(a);
1653     Ez(a) = 1/36.*w2(a).*(...
1654         + w(a) ...
1655         - 1.5.*w(a).*log(w2(a))...
1656         + 9.*(2.*Cu(a)-pi*Cv(a)).*sign(w(a))...
1657         );
1658     end
1659
1660     if any(b)
1661         Ex(b) = 1/12.*v2(b).*(...
1662             + 3.*Cw(b).*(log(v2(b))-1)+ ...
1663             + 2.*sign(v(b)).*(3.*Cv(b)+v(b))...
1664             );
1665
1666         Janssen (wrong?) Ey(b) = 1/24.*v2(b).*(... + v(b).*( log(v2(b)) - 3 ) ... - 12.*Cu(b).*sign(v(b))
1667         ... );
1668
1669         Ey(b) = 1/24*(...
1670             + 3*v(b).^3 ...
1671             - r(b).*v(b).*( 12*Cu(b)+10*u(b)+3*r(b))...
1672             + 2*v(b).*log(r(b)+u(b)).*( 3.*u(b).*(2*Cu(b)+u(b))+ v2(b)- 3*w(b).*(4.*
1673             Cw(b)+w(b))...
1674             + 2*v(b).*log(r(b)-u(b)).*( -3.*u(b).*(2*Cu(b)+u(b))+ v2(b))...
1675             );
1676
1677         Ez(b) = 1/4.*Cu(b).*v2(b).*log(v2(b));
1678     end
1679
1680     if any(c)
1681
1682         Janssen (wrong?) Ex(c) = 1/12.*u2(c).*(... + 2.*u(c).*(sign(u(c))-1) ... + 3.*Cw(c).*(log(u2(c))-
1683         1) ... );
1684
1685         Ex(c) = ...
1686             + 1/12.*r(c).*( 2*r2(c)- 3.*Cw(c).*r(c)+ 6.*Cv(c).*v(c)- 6.*Cw(c).*w(c)
1687             - 6.*w2(c))...
1688             + 1/2.*Cw(c).*(u2(c).*log(r(c)+w(c))+ v2(c).*log(r(c)-w(c)))...
1689             ;
1690         Ey(c) = 1/36.*u(c).^3.*(3*log(u2(c))- 5);
1691         Ez(c) = -1/12.*( 3*Cu(c)+ 2*u(c)).*u2(c).*log(u2(c));
1692     end
1693
1694     % if magnets are not aligned in two directions, use the following
1695     % expressions to calculate the sums
1696
1697     if any(d)
1698         Ex(d) = (...
1699             +1/12*r(d).*( 2*r2(d)+ 6.*Cv(d).*v(d)- 6.*w2(d)- 6*Cw(d).*w(d)- 3*Cw(
1700             d).*r(d))...
1701             ...
1702             -1/2*log(r(d)-u(d)).*u(d).*(2.*w(d).*Cw(d)+r2(d)-u2(d))...
1703             +1/2*log(r(d)-w(d)).*Cw(d).*(r2(d)-w2(d))...
1704             ...
1705             - acoth(r(d)./u(d)).*u(d).*v(d).*(Cv(d)+v(d))...

```



```

1705     -1/2*acoth(r(d)./v(d)).*(Cv(d)+v(d)).*(u2(d)-w2(d))...
1706     + acoth(r(d)./w(d)).*Cw(d).*u2(d)...
1707     ...
1708     +u(d).*w(d).*Cv(d).*atan(u(d).*v(d)./(w(d).*r(d)))...
1709     -u(d).*v(d).*Cw(d).*atan(u(d).*w(d)./(v(d).*r(d)))...
1710     ...
1711     +u(d).*v(d).*w(d).*atan(u(d).*v(d)./(w(d).*r(d)))...
1712     -u(d).*v(d).*Cw(d).*atan(w(d)./r(d))...
1713 );
1714 Ey(d) = 1/72*(...
1715     - 10*u(d).^3 ...
1716     + 9*v(d).^3 ...
1717     - 6*u(d).*w(d).*(18*Cw(d)+7*w(d))...
1718     - 3*r(d).*v(d).*( 12*Cu(d)+10*u(d)+3*r(d))...
1719     ...
1720     - 72*u(d).*v(d).*Cw(d).*log(r(d)+w(d))...
1721     - 72*u(d).*v(d).*Cw(d)...
1722     ...
1723     + 6*v(d).*log(r(d)+u(d)).*( 3.*u(d).*(2*Cu(d)+u(d))+ v2(d)- 3*w(d).*(4.*
Cw(d)+w(d)))...
1724     + 6*v(d).*log(r(d)-u(d)).*( -3.*u(d).*(2*Cu(d)+u(d))+ v2(d))...
1725     ...
1726     + 6*log(r(d)+v(d)).*( 2.*u(d).^3 + 3*Cu(d).*(u2(d)-w2(d)
))...
1727     + 18*log(r(d)-v(d)).*( 2.*u(d).*w(d).*(2*Cw(d)+w(d))- Cu(d).*(u2(d)-w2(
d)))...
1728     ...
1729     + 24*w2(d).*(3.*Cw(d)+w(d)).*atan(u(d)./w(d))...
1730     + 36*Cw(d).*(u2(d)+w2(d)).*atan(w(d)./u(d))...
1731     + 3*v(d).^3.*Cw(d).*atan(u(d).*w(d)./(v(d).*r(d)))...
1732     + 12*w(d).*( w2(d)+3.*Cw(d).*w(d)- 3.*u(d).*(2.*Cu(d)+u(d))).*atan(u(d)
.*v(d)./(w(d).*r(d)))...
1733     + 36*u2(d).*Cw(d).*atan((v(d).*w(d))./(u(d).*r(d)))...
1734 );
1735 Ez(d) = 1/12*(...
1736     + 1/3*w(d).^3 ...
1737     + 2.*w(d).*v2(d)...
1738     + r(d).*w(d).*(6.*Cu(d)+u(d))...
1739     + 6.*u(d).*w(d).*(2.*Cu(d)+u(d))...
1740     + 6.*u(d).*w(d).*(2.*Cu(d)+u(d)).*log(r(d)-u(d))...
1741     - w(d).*( 3.*v2(d)+ w2(d)).*log(r(d)+u(d))...
1742     - 2.*( 2.*u(d).^3 + 3.*Cu(d).*(u2(d)-v2(d))).*log(r(d)+w(d))...
1743     - 12.*(Cv(d)+v(d)).*(...
1744     - u(d).*v(d).*log(r(d)+w(d))...
1745     + u(d).*w(d).*log(r(d)-v(d))...
1746     - v(d).*w(d).*log(r(d)+u(d))...
1747     )...
1748     - 2.*v(d).*(v2(d)-3.*u(d).*(2.*Cu(d)+u(d))).*atan(w(d)./r(d))...
1749     + 2.*v(d).*(v2(d)+3.*u(d).*(2.*Cu(d)+u(d))).*atan((u(d).*w(d))./(v(d)
.*r(d)))...
1750     - 6.*(Cv(d)+v(d)).*(...
1751     + u2(d).*( atan(w(d)./u(d))+ atan((v(d).*w(d))./(u(d).*r(d))))...

```

```

1752         + v2(d).*atan((u(d).*w(d))./(v(d).*r(d)))...
1753         + w2(d).*( ...
1754             + 2.*atan(u(d)./w(d))...
1755             + atan(w(d)./u(d))...
1756             + atan((u(d).*v(d))./(w(d).*r(d)))...
1757             ) ...
1758             )...
1759         );
1760     end

1762     ind_sum = (-1)^(ii+jj+kk+ll+mm+nn);
1763     Tx = Tx + ind_sum.*Ex;
1764     Ty = Ty + ind_sum.*Ey;
1765     Tz = Tz + ind_sum.*Ez;

1767 end
1768 end
1769 end
1770 end
1771 end
1772 end

1774 torque_zy = bzc1*byr2/(16*pi*pi*1e-7).*[Tx; Ty; Tz];
1776 end

```

## 7 Mathematical functions

### 7.1 The `ellipkepi()` function

Complete elliptic integrals calculated with the arithmetic-geometric mean algorithms contained here:  
<http://dlmf.nist.gov/19.8>. Valid for  $0 \leq a \leq 1$  and  $0 \leq m \leq 1$ .

```

2009 function [K,E,PI] = ellipkepi(a,m)

2011 a1 = 1;
2012 g1 = sqrt(1-m);
2013 p1 = sqrt(1-a);
2014 q1 = 1;
2015 w1 = 1;

2017 nn = 0;
2018 qq = 1;
2019 ww = m;

2021 while max(abs(w1(:)))> eps || max(abs(q1(:)))> eps

2023 % Update from previous loop
2024 a0 = a1;
2025 g0 = g1;
2026 p0 = p1;
2027 q0 = q1;

2029 % for Elliptic I
2030 a1 = (a0+g0)/2;
2031 g1 = sqrt(a0.*g0);

```

```

2033 % for Elliptic II
2034 nn = nn + 1;
2035 d1 = (a0-g0)/2;
2036 w1 = 2^nn*d1.^2;
2037 ww = ww + w1;

2039 % for Elliptic III
2040 rr = p0.^2+a0.*g0;
2041 p1 = rr./p0/2;
2042 q1 = q0.*(p0.^2-a0.*g0)./rr/2;
2043 qq = qq + q1;

2045 end

2047 K = 1./a1*pi/2;
2048 E = K.*(1-ww/2);
2049 PI = K.*(1+a./(2-2*a).*qq);

2051 im = find(m == 1);
2052 if ~isempty(im)
2053     K(im) = inf;
2054     E(im) = ones(length(im),1);
2055     PI(im) = inf;
2056 end

2058 end

```

## 8 Magnet arrays

### 8.1 The multipoleforces() function

```

2070 function [varargout] = multipoleforces(fixed_array, float_array, displ, varargin)

2072 debug_disp = @(str)disp([]);
2073 calc_force_bool = false;
2074 calc_stiffness_bool = false;
2075 calc_torque_bool = false;

2077 for ii = 1:length(varargin)
2078     switch varargin{ii}
2079         case 'debug',    debug_disp = @(str)disp(str);
2080         case 'force',    calc_force_bool    = true;
2081         case 'stiffness', calc_stiffness_bool = true;
2082         case 'torque',   calc_torque_bool   = true;
2083         otherwise
2084             error(['Unknown calculation option ',varargin{ii},'])
2085         end
2086     end

2088 if ~calc_force_bool && ~calc_stiffness_bool && ~calc_torque_bool
2089     varargin{end+1} = 'force';
2090     calc_force_bool = true;
2091 end

```

```

2094 if size(displ,1)== 3
2095 % all good
2096 elseif size(displ,2)== 3
2097     displ = transpose(displ);
2098 else
2099     error(['Displacements matrix should be of size (3, D)',...
2100           'where D is the number of displacements.'])
2101 end
2103 Ndispl = size(displ,2);
2105 if calc_force_bool
2106     forces_out = nan([3 Ndispl]);
2107 end
2109 if calc_stiffness_bool
2110     stiffnesses_out = nan([3 Ndispl]);
2111 end
2113 if calc_torque_bool
2114     torques_out = nan([3 Ndispl]);
2115 end
2118 part = @(x,y)x(y);
2120 fixed_array = complete_array_from_input(fixed_array);
2121 float_array = complete_array_from_input(float_array);
2123 if calc_force_bool
2124     array_forces = nan([3 Ndispl fixed_array.total float_array.total]);
2125 end
2127 if calc_stiffness_bool
2128     array_stiffnesses = nan([3 Ndispl fixed_array.total float_array.total]);
2129 end
2131 displ_from_array_corners = displ ...
2132     + repmat(fixed_array.size/2,[1 Ndispl])...
2133     - repmat(float_array.size/2,[1 Ndispl]);
2136 for ii = 1:fixed_array.total
2138     fixed_magnet = magnetdefine(...
2139         'type', 'cuboid',...
2140         'dim',   fixed_array.dim(ii,:), ...
2141         'magn',  fixed_array.magn(ii), ...
2142         'magdir', fixed_array.magdir(ii,:)...
2143     );
2145     for jj = 1:float_array.total
2147         float_magnet = magnetdefine(...
2148             'type', 'cuboid',...
2149             'dim',   float_array.dim(jj,:), ...
2150             'magn',  float_array.magn(jj), ...
2151             'magdir', float_array.magdir(jj,:)...
2152         );

```

```

2154     mag_displ = displ_from_array_corners ...
2155         - repmat(fixed_array.magloc(ii,:),[1 Ndispl])...
2156         + repmat(float_array.magloc(jj,:),[1 Ndispl]);

2158     if calc_force_bool && ~calc_stiffness_bool
2159         array_forces(:, :, ii, jj) = ...
2160             magnetforces(fixed_magnet, float_magnet, mag_displ, varargin{:});
2161     elseif calc_stiffness_bool && ~calc_force_bool
2162         array_stiffnesses(:, :, ii, jj) = ...
2163             magnetforces(fixed_magnet, float_magnet, mag_displ, varargin{:});
2164     else
2165         [array_forces(:, :, ii, jj) array_stiffnesses(:, :, ii, jj)] = ...
2166             magnetforces(fixed_magnet, float_magnet, mag_displ, varargin{:});
2167     end

2169 end
2170 end

2172 if calc_force_bool
2173     forces_out = sum(sum(array_forces, 4), 3);
2174 end

2176 if calc_stiffness_bool
2177     stiffnesses_out = sum(sum(array_stiffnesses, 4), 3);
2178 end

2181 varargout = {};

2183 for ii = 1:length(varargin)
2184     switch varargin{ii}
2185         case 'force'
2186             varargout{end+1} = forces_out;

2188         case 'stiffness'
2189             varargout{end+1} = stiffnesses_out;

2191         case 'torque'
2192             varargout{end+1} = torques_out;
2193     end
2194 end

2200 function array = complete_array_from_input(array)

2202 if ~isfield(array, 'type')
2203     array.type = 'generic';
2204 end

2207 if ~isfield(array, 'face')
2208     array.face = 'undefined';
2209 end

2211 linear_index = 0;
2212 planar_index = [0 0];

2214 switch array.type
2215     case 'generic'

```

```

2216 case 'linear',          linear_index = 1;
2217 case 'linear-quasi',    linear_index = 1;
2218 case 'planar',          planar_index = [1 2];
2219 case 'quasi-halbach',   planar_index = [1 2];
2220 case 'patchwork',       planar_index = [1 2];
2221 otherwise
2222     error(['Unknown array type ',array.type,','.'])
2223 end

2225 if ~isequal(array.type,'generic')
2226     if linear_index == 1
2227         if ~isfield(array,'align')
2228             array.align = 'x';
2229         end
2230         switch array.align
2231             case 'x', linear_index = 1;
2232             case 'y', linear_index = 2;
2233             case 'z', linear_index = 3;
2234             otherwise
2235                 error('Alignment for linear array must be 'x', 'y', or 'z'.')
2236             end
2237         else
2238             if ~isfield(array,'align')
2239                 array.align = 'xy';
2240             end
2241             switch array.align
2242                 case 'xy', planar_index = [1 2];
2243                 case 'yz', planar_index = [2 3];
2244                 case 'xz', planar_index = [1 3];
2245                 otherwise
2246                     error('Alignment for planar array must be 'xy', 'yz', or 'xz'.')
2247                 end
2248             end
2249         end

2251 switch array.face
2252     case {'+x','-x'}, facing_index = 1;
2253     case {'+y','-y'}, facing_index = 2;
2254     case {'up','down'}, facing_index = 3;
2255     case {'+z','-z'}, facing_index = 3;
2256     case 'undefined', facing_index = 0;
2257 end

2259 if linear_index ~= 0
2260     if linear_index == facing_index
2261         error('Arrays cannot face into their alignment direction.')
2262     end
2263 elseif ~isequal( planar_index, [0 0] )
2264     if any( planar_index == facing_index )
2265         error('Planar-type arrays can only face into their orthogonal direction')
2266     end
2267 end

2270 switch array.type

```

```

2271     case 'linear'

2273 array = extrapolate_variables(array);

2275 array.mcount = ones(1,3);
2276 array.mcount(linear_index)= array.Nmag;

2278     case 'linear-quasi'

2281 if isfield(array,'ratio')&& isfield(array,'mlength')
2282     error('Cannot specify both ''ratio''and ''mlength''.')
2283 elseif ~isfield(array,'ratio')&& ~isfield(array,'mlength')
2284     error('Must specify either ''ratio''or ''mlength''.')
2285 end

2288 array.Nmag_per_wave = 4;
2289 array.magdir_rotate = 90;

2291 if isfield(array,'Nwaves')
2292     array.Nmag = array.Nmag_per_wave*array.Nwaves+1;
2293 else
2294     error('''Nwaves''must be specified.')
2295 end

2297 if isfield(array,'mlength')
2298     if numel(array.mlength)~=2
2299         error('''mlength''must have length two for linear-quasi arrays.')
2300     end
2301     array.ratio = array.mlength(2)/array.mlength(1);
2302 else
2303     if isfield(array,'length')
2304         array.mlength(1)= 2*array.length/(array.Nmag*(1+array.ratio)+1-array.ratio);
2305         array.mlength(2)= array.mlength(1)*array.ratio;
2306     else
2307         error('''length''must be specified.')
2308     end
2309 end

2311 array.mcount = ones(1,3);
2312 array.mcount(linear_index)= array.Nmag;

2314 array.msize = nan([array.mcount 3]);

2316 [sindex_x sindex_y sindex_z] = ...
2317     meshgrid(1:array.mcount(1), 1:array.mcount(2), 1:array.mcount(3));

2321 all_indices = [1 1 1];
2322 all_indices(linear_index)= 0;
2323 all_indices(facing_index)= 0;
2324 width_index = find(all_indices);

2326 for ii = 1:array.Nmag
2327     array.msize(sindex_x(ii),sindex_y(ii),sindex_z(ii),linear_index)= ...
2328         array.mlength(mod(ii-1,2)+1);
2329     array.msize(sindex_x(ii),sindex_y(ii),sindex_z(ii),facing_index)= ...
2330         array.height;

```

```

2331     array.msize(sindex_x(ii),sindex_y(ii),sindex_z(ii),width_index)= ...
2332         array.width;
2333 end

2336     case 'planar'

2338     if isfield(array,'length')
2339         if length(array.length)== 1
2340             if isfield(array,'width')
2341                 array.length = [ array.length array.width ];
2342             else
2343                 array.length = [ array.length array.length ];
2344             end
2345         end
2346     end

2348     if isfield(array,'mlength')
2349         if length(array.mlength)== 1
2350             if isfield(array,mwidth)
2351                 array.mlength = [ array.mlength array.mwidth ];
2352             else
2353                 array.mlength = [ array.mlength array.mlength ];
2354             end
2355         end
2356     end

2358     var_names = {'length','mlength','wavelength','Nwaves',...
2359                 'Nmag','Nmag_per_wave','magdir_rotate'};

2361     tmp_array1 = struct();
2362     tmp_array2 = struct();
2363     var_index = zeros(size(var_names));

2365     for iii = 1:length(var_names)
2366         if isfield(array,var_names(iii))
2367             tmp_array1.(var_names{iii})= array.(var_names{iii})(1);
2368             tmp_array2.(var_names{iii})= array.(var_names{iii})(end);
2369         else
2370             var_index(iii)= 1;
2371         end
2372     end

2374     tmp_array1 = extrapolate_variables(tmp_array1);
2375     tmp_array2 = extrapolate_variables(tmp_array2);

2377     for iii = find(var_index)
2378         array.(var_names{iii})= [tmp_array1.(var_names{iii})tmp_array2.(var_names{iii})];
2379     end

2381     array.width = array.length(2);
2382     array.length = array.length(1);

2384     array.mwidth = array.mlength(2);
2385     array.mlength = array.mlength(1);

2387     array.mcount = ones(1,3);
2388     array.mcount(planar_index)= array.Nmag;

```



```

2390     case 'quasi-halbach'
2391
2392     if isfield(array,'mcount')
2393         if numel(array.mcount)~=3
2394             error(''mcount''must always have three elements.')
2395         end
2396     elseif isfield(array,'Nwaves')
2397         if numel(array.Nwaves)> 2
2398             error(''Nwaves''must have one or two elements only.')
2399         end
2400         array.mcount(facing_index)= 1;
2401         array.mcount(planar_index)= 4*array.Nwaves+1;
2402     elseif isfield(array,'Nmag')
2403         if numel(array.Nmag)> 2
2404             error(''Nmag''must have one or two elements only.')
2405         end
2406         array.mcount(facing_index)= 1;
2407         array.mcount(planar_index)= array.Nmag;
2408     else
2409         error('Must specify the number of magnets (''mcount''or ''Nmag'')or wavelengths (''
2410         Nwaves'')')
2411     end
2412
2413     case 'patchwork'
2414
2415     if isfield(array,'mcount')
2416         if numel(array.mcount)~=3
2417             error(''mcount''must always have three elements.')
2418         end
2419     elseif isfield(array,'Nmag')
2420         if numel(array.Nmag)> 2
2421             error(''Nmag''must have one or two elements only.')
2422         end
2423         array.mcount(facing_index)= 1;
2424         array.mcount(planar_index)= array.Nmag;
2425     else
2426         error('Must specify the number of magnets (''mcount''or ''Nmag'')')
2427     end
2428
2429
2430     array.total = prod(array.mcount);
2431
2432     if ~isfield(array,'msize')
2433         array.msize = [NaN NaN NaN];
2434         if linear_index ~=0
2435             array.msize(linear_index)= array.mlength;
2436             array.msize(facing_index)= array.height;
2437             array.msize(isnan(array.msize))= array.width;
2438         elseif ~isequal( planar_index, [0 0] )
2439             array.msize(planar_index)= [array.mlength array.mwidth];
2440             array.msize(facing_index)= array.height;
2441         else
2442             error('The array property ''msize''is not defined and I have no way to infer it.'
2443             )

```

```

2444     end
2445 elseif numel(array.msize)== 1
2446     array.msize = repmat(array.msize,[3 1]);
2447 end

2449 if numel(array.msize)== 3
2450     array.msize_array = ...
2451         repmat(reshape(array.msize,[1 1 1 3]), array.mcount);
2452 else
2453     if isequal([array.mcount 3],size(array.msize))
2454         array.msize_array = array.msize;
2455     else
2456         error('Magnet size ''msize''must have three elements (or one element for a cube magnet
2457         ).')
2458     end
2459 array.dim = reshape(array.msize_array, [array.total 3]);

2461 if ~isfield(array,'mgap')
2462     array.mgap = [0; 0; 0];
2463 elseif length(array.mgap)== 1
2464     array.mgap = repmat(array.mgap,[3 1]);
2465 end

2469 if ~isfield(array,'magn')
2470     if isfield(array,'grade')
2471         array.magn = grade2magn(array.grade);
2472     else
2473         array.magn = 1;
2474     end
2475 end

2477 if length(array.magn)== 1
2478     array.magn = repmat(array.magn,[array.total 1]);
2479 else
2480     error('Magnetisation magnitude ''magn''must be a single value.')
2481 end

2485 if ~isfield(array,'magdir_fn')

2487     if ~isfield(array,'face')
2488         array.face = '+z';
2489     end

2491     switch array.face
2492     case {'up','+z','+y','+x'}, magdir_rotate_sign = 1;
2493     case {'down','-z','-y','-x'}, magdir_rotate_sign = -1;
2494     end

2496     if ~isfield(array,'magdir_first')
2497         array.magdir_first = magdir_rotate_sign*90;
2498     end

2500     magdir_fn_comp{1} = @(ii,jj,kk)0;
2501     magdir_fn_comp{2} = @(ii,jj,kk)0;

```

```

2502 magdir_fn_comp{3} = @(ii,jj,kk)0;

2504 switch array.type
2505 case 'linear'
2506     magdir_theta = @(nn)...
2507         array.magdir_first+magdir_rotate_sign*array.magdir_rotate*(nn-1);

2509     magdir_fn_comp{linear_index} = @(ii,jj,kk)...
2510         cosd(magdir_theta(part([ii,jj,kk],linear_index)));

2512     magdir_fn_comp{facing_index} = @(ii,jj,kk)...
2513         sind(magdir_theta(part([ii,jj,kk],linear_index)));

2515 case 'linear-quasi'

2517     magdir_theta = @(nn)...
2518         array.magdir_first+magdir_rotate_sign*90*(nn-1);

2520     magdir_fn_comp{linear_index} = @(ii,jj,kk)...
2521         cosd(magdir_theta(part([ii,jj,kk],linear_index)));

2523     magdir_fn_comp{facing_index} = @(ii,jj,kk)...
2524         sind(magdir_theta(part([ii,jj,kk],linear_index)));

2526 case 'planar'

2528     magdir_theta = @(nn)...
2529         array.magdir_first(1)+magdir_rotate_sign*array.magdir_rotate(1)*(nn-1);

2531     magdir_phi = @(nn)...
2532         array.magdir_first(end)+magdir_rotate_sign*array.magdir_rotate(end)*(nn-1);

2534     magdir_fn_comp{planar_index(1)} = @(ii,jj,kk)...
2535         cosd(magdir_theta(part([ii,jj,kk],planar_index(2))));

2537     magdir_fn_comp{planar_index(2)} = @(ii,jj,kk)...
2538         cosd(magdir_phi(part([ii,jj,kk],planar_index(1))));

2540     magdir_fn_comp{facing_index} = @(ii,jj,kk)...
2541         sind(magdir_theta(part([ii,jj,kk],planar_index(1))))...
2542         + sind(magdir_phi(part([ii,jj,kk],planar_index(2))));

2544 case 'patchwork'

2546     magdir_fn_comp{planar_index(1)} = @(ii,jj,kk)0;

2548     magdir_fn_comp{planar_index(2)} = @(ii,jj,kk)0;

2550     magdir_fn_comp{facing_index} = @(ii,jj,kk)...
2551         magdir_rotate_sign*(-1)^( ...
2552             part([ii,jj,kk],planar_index(1))...
2553             + part([ii,jj,kk],planar_index(2))...
2554             + 1 ...
2555         );

2557 case 'quasi-halbach'

2559     magdir_fn_comp{planar_index(1)} = @(ii,jj,kk)...
2560         sind(90*part([ii,jj,kk],planar_index(1)))...
2561         * cosd(90*part([ii,jj,kk],planar_index(2)));

2563     magdir_fn_comp{planar_index(2)} = @(ii,jj,kk)...

```

```

2564     cosd(90*part([ii,jj,kk],planar_index(1)))...
2565     * sind(90*part([ii,jj,kk],planar_index(2)));

2567     magdir_fn_comp{facing_index} = @(ii,jj,kk)...
2568     magdir_rotate_sign ...
2569     * sind(90*part([ii,jj,kk],planar_index(1)))...
2570     * sind(90*part([ii,jj,kk],planar_index(2)));

2572 otherwise
2573     error('Array property ''magdir_fn''not defined and I have no way to infer it.')
2574 end

2576 array.magdir_fn = @(ii,jj,kk)...
2577     [ magdir_fn_comp{1}(ii,jj,kk)...
2578       magdir_fn_comp{2}(ii,jj,kk)...
2579       magdir_fn_comp{3}(ii,jj,kk)];

2581 end

2587 array.magloc = nan([array.total 3]);
2588 array.magdir = array.magloc;
2589 arrat.magloc_array = nan([array.mcount(1)array.mcount(2)array.mcount(3)3]);

2591 nn = 0;
2592 for iii = 1:array.mcount(1)
2593     for jjj = 1:array.mcount(2)
2594         for kkk = 1:array.mcount(3)
2595             nn = nn + 1;
2596             array.magdir(nn,:)= array.magdir_fn(iii,jjj,kkk);
2597         end
2598     end
2599 end

2601 magsep_x = zeros(size(array.mcount(1)));
2602 magsep_y = zeros(size(array.mcount(2)));
2603 magsep_z = zeros(size(array.mcount(3)));

2605 magsep_x(1)= array.msize_array(1,1,1,1)/2;
2606 magsep_y(1)= array.msize_array(1,1,1,2)/2;
2607 magsep_z(1)= array.msize_array(1,1,1,3)/2;

2609 for iii = 2:array.mcount(1)
2610     magsep_x(iii)= array.msize_array(iii-1,1,1,1)/2 ...
2611         + array.msize_array(iii ,1,1,1)/2 ;
2612 end
2613 for jjj = 2:array.mcount(2)
2614     magsep_y(jjj)= array.msize_array(1,jjj-1,1,2)/2 ...
2615         + array.msize_array(1,jjj ,1,2)/2 ;
2616 end
2617 for kkk = 2:array.mcount(3)
2618     magsep_z(kkk)= array.msize_array(1,1,kkk-1,3)/2 ...
2619         + array.msize_array(1,1,kkk ,3)/2 ;
2620 end

2622 magloc_x = cumsum(magsep_x);

```

```

2623 magloc_y = cumsum(magsep_y);
2624 magloc_z = cumsum(magsep_z);

2626 for iii = 1:array.mcount(1)
2627     for jjj = 1:array.mcount(2)
2628         for kkk = 1:array.mcount(3)
2629             array.magloc_array(iii,jjj,kkk,:)= ...
2630                 [magloc_x(iii); magloc_y(jjj); magloc_z(kkk)] ...
2631                 + [iii-1; jjj-1; kkk-1].*array.mgap;
2632         end
2633     end
2634 end
2635 array.magloc = reshape(array.magloc_array,[array.total 3]);

2637 array.size = squeeze( array.magloc_array(end,end,end,:)...
2638     - array.magloc_array(1,1,1,:)...
2639     + array.msize_array(1,1,1,:)/2 ...
2640     + array.msize_array(end,end,end,:)/2 );

2642 debug_disp('Magnetisation directions')
2643 debug_disp(array.magdir)

2645 debug_disp('Magnet locations:')
2646 debug_disp(array.magloc)

2649 end

2653 function array_out = extrapolate_variables(array)
2655 var_names = {'wavelength','length','Nwaves','mlength',...
2656             'Nmag','Nmag_per_wave','magdir_rotate'};

2658 if isfield(array,'Nwaves')
2659     mcount_extra = 1;
2660 else
2661     mcount_extra = 0;
2662 end

2664 if isfield(array,'mlength')
2665     mlength_adjust = false;
2666 else
2667     mlength_adjust = true;
2668 end

2670 variables = nan([7 1]);

2672 for iii = 1:length(var_names);
2673     if isfield(array,var_names(iii))
2674         variables(iii)= array.(var_names{iii});
2675     end
2676 end

2678 var_matrix = ...
2679     [1, 0, 0, -1, 0, -1, 0;
2680      0, 1, 0, -1, -1, 0, 0;
2681      0, 0, 1, 0, -1, 1, 0;
2682      0, 0, 0, 0, 0, 1, 1];

```

```

2684 var_results = [0 0 0 log(360)]';
2685 variables = log(variables);

2687 idx = ~isnan(variables);
2688 var_known = var_matrix(:,idx)*variables(idx);
2689 var_calc = var_matrix(:,~idx)\(var_results-var_known);
2690 variables(~idx)= var_calc;
2691 variables = exp(variables);

2693 for iii = 1:length(var_names);
2694     array.(var_names{iii})= variables(iii);
2695 end

2697 array.Nmag = round(array.Nmag)+ mcount_extra;
2698 array.Nmag_per_wave = round(array.Nmag_per_wave);

2700 if mlength_adjust
2701     array.mlength = array.mlength * (array.Nmag-mcount_extra)/array.Nmag;
2702 end

2704 array_out = array;

2706 end

2710 end

```