

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

Will Robertson

December 19, 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	3
1	Defining magnets and coils	3
2	Forces	4
2.1	Forces between magnets	4
2.2	Forces between multipole arrays of magnets	5
3	Meta-information	7
II	Typeset code / implementation	8
4	Magnets setup	8
4.1	The <code>magnetdefine()</code> function	8
4.1.1	The <code>grade2magn()</code> function	9
4.1.2	The <code>resolve_magdir()</code> function	10

5	The magnetforces() function	11
5.1	The single_magnet_cyl_force() function	15
5.2	The single_magnet_force() function	16
5.3	The single_magnet_torque() function	17
5.4	The stiffnesses_calc_z_z() function	19
5.5	The stiffnesses_calc_z_y() function	20
5.5.1	Helpers	20
5.5.2	The multiply_x_log_y() function	20
5.5.3	The atan1() function	21
5.6	The stiffnesses_calc_z_x() function	21
5.7	The torques_calc_z_y() function	21
5.8	The torques_calc_z_x() function	21
5.9	The forces_magcyl_shell_calc() function	22
6	Magnet interactions	22
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
7	Mathematical functions	28
7.1	The ellipkepi() function	28
8	Magnet arrays	29
8.1	The multipoleforces() function	29

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:

$$\begin{aligned}(1, 0, 0)_{\text{cartesian}} &\equiv (0, 0)_{\text{spherical}} \\ (0, 1, 0)_{\text{cartesian}} &\equiv (90, 0)_{\text{spherical}} \\ (0, 0, 1)_{\text{cartesian}} &\equiv (0, 90)_{\text{spherical}}\end{aligned}$$

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 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 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');
```

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

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

⁴<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

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

49 % default to +Z magnetisation
50 if ~isfield(mag, 'dir')
51     if ~isfield(mag, 'magdir')
52         mag.dir = [0 0 1];
53         mag.magdir = [0 0 1];
54     else
```



```

55     mag.dir = mag.magdir;
56 end
57 else
58     if ~isfield(mag,'magdir')
59         mag.magdir = mag.dir;
60     end
61 end
62
63 % convert from current/turns to equiv magnetisation:
64 if ~isfield(mag,'magn')
65     if isfield(mag,'turns') && isfield(mag,'current')
66         mag.magn = 4*pi*1e-7*mag.turns*mag.current/mag.dim(2);
67     end
68 end
69
70 if isfield(mag,'radius') && isfield(mag,'height')
71     mag.dim = [mag.radius, mag.height];
72 end
73
74 else
75     if ~isfield(mag,'magdir')
76         warning('Magnet direction ("magdir") not specified; assuming +z.')
77         mag.magdir = [0; 0; 1];
78     else
79         mag.magdir = resolve_magdir(mag.magdir);
80     end
81 end
82
83 end
84
85 mag.fundefined = true;
86
87 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.

```

104 function magn = grade2magn(grade)
105
106 if isnumeric(grade)
107     magn = 2*sqrt(grade/100);
108 else
109     if strcmp(grade(1),'N')
110         grade = grade(2:end);
111     end

```

```

112     magn = 2*sqrt(str2double(grade)/100);
113 end
115 end

```

4.1.2 The resolve_magdir() function

```

120 function magdir = resolve_magdir(magdir)

```

Magnetisation directions are specified in either cartesian or spherical coordinates.

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).

```

129     if numel(magdir)== 2
130         theta = magdir(1);
131         phi = magdir(2);
132         magdir = [ cosd(phi)*cosd(theta); cosd(phi)*sind(theta); sind(phi)];
133     elseif numel(magdir)== 3
134         if all(magdir == 0)
135             magdir = [0; 0; 0]; % this looks redundant but ensures column vector
136         else
137             magdir = magdir(:)/norm(magdir);
138         end
139     elseif numel(magdir)== 1
140         switch magdir
141             case 'x'; magdir = [1;0;0];
142             case 'y'; magdir = [0;1;0];
143             case 'z'; magdir = [0;0;1];
144             case '+x'; magdir = [1;0;0];
145             case '+y'; magdir = [0;1;0];
146             case '+z'; magdir = [0;0;1];
147             case '-x'; magdir = [-1; 0; 0];
148             case '-y'; magdir = [ 0;-1; 0];
149             case '-z'; magdir = [ 0; 0;-1];
150             otherwise, error('Magnetisation %s not understood.',magdir);
151         end
152     else
153         error('magdir has wrong number of elements.')
154     end
157 end

```

5 The magnetforces() function

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

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

403 [index_i, index_j, index_k, index_l, index_p, index_q] = ndgrid([0 1]);

405 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`.

```
413 debug_disp = @(str)disp([]);
414 calc_force_bool = false;
415 calc_stiffness_bool = false;
416 calc_torque_bool = false;

418 for iii = 1:length(varargin)
419     switch varargin{iii}
420         case 'debug',    debug_disp = @(str)disp(str);
421         case 'force',    calc_force_bool = true;
422         case 'stiffness', calc_stiffness_bool = true;
423         case 'torque',    calc_torque_bool = true;
424         case 'x', warning("Options 'x','y','z'are no longer supported.");
425         case 'y', warning("Options 'x','y','z'are no longer supported.");
426         case 'z', warning("Options 'x','y','z'are no longer supported.");
427         otherwise
428             error(['Unknown calculation option ',varargin{iii},'])
429     end
430 end

432 if ~calc_force_bool && ~calc_stiffness_bool && ~calc_torque_bool
433     varargin{end+1} = 'force';
434     calc_force_bool = true;
435 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.

```
442 if size(displ,1)== 3
443     % all good
444 elseif size(displ,2)== 3
445     displ = transpose(displ);
446 else
447     error(['Displacements matrix should be of size (3, D)',...
448         'where D is the number of displacements.'])
449 end

451 Ndispl = size(displ,2);

453 if calc_force_bool
454     forces_out = nan([3 Ndispl]);
455 end
```

```

457 if calc_stiffness_bool
458     stiffnesses_out = nan([3 Ndispl]);
459 end
461 if calc_torque_bool
462     torques_out = nan([3 Ndispl]);
463 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.

```

491 if ~isfield(magnet_fixed,'fundefined')
492     magnet_fixed = magnetdefine(magnet_fixed);
493 end
494 if ~isfield(magnet_float,'fundefined')
495     magnet_float = magnetdefine(magnet_float);
496 end

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

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

506     coil = magnet_fixed;
507     magnet = magnet_float;
508     magtype = 'coil';
509     coil_sign = +1;

511 end

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

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

519     coil = magnet_float;
520     magnet = magnet_fixed;
521     magtype = 'coil';
522     coil_sign = -1;

```

```

524 end

527 if ~strcmp(magnet_fixed.type, magnet_float.type)
528     error('Magnets must be of same type')
529 end
530 magtype = magnet_fixed.type;

533 if strcmp(magtype, 'cuboid')

535     size1 = magnet_fixed.dim(:)/2;
536     size2 = magnet_float.dim(:)/2;

538     J1 = magnet_fixed.magn*magnet_fixed.magdir;
539     J2 = magnet_float.magn*magnet_float.magdir;

541     swap_x_y = @(vec)vec([2 1 3],:);
542     swap_x_z = @(vec)vec([3 2 1],:);
543     swap_y_z = @(vec)vec([1 3 2],:);

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

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

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

554     size1_x = swap_x_z(size1);
555     size2_x = swap_x_z(size2);
556     J1_x     = rotate_x_to_z(J1);
557     J2_x     = rotate_x_to_z(J2);

559     size1_y = swap_y_z(size1);
560     size2_y = swap_y_z(size2);
561     J1_y     = rotate_y_to_z(J1);
562     J2_y     = rotate_y_to_z(J2);

564 elseif strcmp(magtype, 'cylinder')

566     size1 = magnet_fixed.dim(:);
567     size2 = magnet_float.dim(:);

569     if any(abs(magnet_fixed.dir)~= abs(magnet_float.dir))
570         error('Cylindrical magnets must be oriented in the same direction')
571     end
572     if any(abs(magnet_fixed.magdir)~= abs(magnet_float.magdir))
573         error('Cylindrical magnets must be oriented in the same direction')
574     end
575     if any(abs(magnet_fixed.dir)~= abs(magnet_fixed.magdir))
576         error('Cylindrical magnets must be magnetised in the same direction as their orientation
577 ')
578     end
579     if any(abs(magnet_float.dir)~= abs(magnet_float.magdir))
580         error('Cylindrical magnets must be magnetised in the same direction as their orientation
581 ')
582     end
583 end

```

```

582 cylmdir = find(magnet_float.magdir ~= 0);
583 cylnotdir = find(magnet_float.magdir == 0);
584 if length(cylmdir)~= 1
585     error('Cylindrical magnets must be aligned in one of the x, y or z directions')
586 end

588 magnet_float.magdir = magnet_float.magdir(:);
589 magnet_fixed.magdir = magnet_fixed.magdir(:);
590 magnet_float.dir = magnet_float.dir(:);
591 magnet_fixed.dir = magnet_fixed.dir(:);

593 J1 = magnet_fixed.magn*magnet_fixed.magdir;
594 J2 = magnet_float.magn*magnet_float.magdir;
595 debug_disp('Magnetisation vectors:')
596 debug_disp(J1)
597 debug_disp(J2)

599 end

```

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.

```

607 if strcmp(magtype,'coil')
609     for iii = 1:Ndispl
610         forces_out(:,iii)= coil_sign*coil.dir*...
611             forces_magcyl_shell_calc(...
612                 magnet.dim, ...
613                 coil.dim, ...
614                 squeeze(displ(cylmdir,:)), ...
615                 J1(cylmdir), ...
616                 coil.current, ...
617                 coil.turns);
618     end
620 elseif strcmp(magtype,'cuboid')
622     if calc_force_bool
623         for iii = 1:Ndispl
624             forces_out(:,iii)= single_magnet_force(displ(:,iii));
625         end
626     end
628     if calc_stiffness_bool
629         for iii = 1:Ndispl
630             stiffnesses_out(:,iii)= single_magnet_stiffness(displ(:,iii));
631         end
632     end
634     if calc_torque_bool
635         torques_out = single_magnet_torque(displ,magnet_float.lever);
636     end
638 elseif strcmp(magtype,'cylinder')
640     if calc_force_bool
641         for iii = 1:Ndispl

```

```

642     forces_out(:,iii)= single_magnet_cyl_force(displ(:,iii));
643 end
644 end
646 if calc_stiffness_bool
647     error('Stiffness cannot be calculated for cylindrical magnets yet.')
648 end
650 if calc_torque_bool
651     error('Torques cannot be calculated for cylindrical magnets yet.')
652 end
654 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, which makes the code a bit uglier but is presumably a bit nicer for the user and/or just a bit more flexible.

```

663 argcount = 0;
665 for iii = 1:length(varargin)
666     switch varargin{iii}
667         case 'force',    argcount = argcount+1;
668         case 'stiffness', argcount = argcount+1;
669         case 'torque',   argcount = argcount+1;
670     end
671 end
673 varargout = cell(argcount,1);
675 argcount = 0;
677 for iii = 1:length(varargin)
678     switch varargin{iii}
679         case 'force',    argcount = argcount+1; varargout{argcount} = forces_out;
680         case 'stiffness', argcount = argcount+1; varargout{argcount} = stiffnesses_out;
681         case 'torque',   argcount = argcount+1; varargout{argcount} = torques_out;
682     end
683 end

```

That is the end of the main function.

5.1 The `single_magnet_cyl_force()` function

```

692 function forces_out = single_magnet_cyl_force(displ)
694     forces_out = nan(size(displ));
696     ecc = sqrt(sum(displ(cylnotdir).^2));
698     if ecc < eps
699         debug_disp('Coaxial')
700         magdir = [0;0;0];
701         magdir(cylidir)= 1;

```

```

702     forces_out = magdir*cylinder_force_coaxial(J1(cyldir), J2(cyldir), size1(1), size2
(1), size1(2), size2(2), displ(cyldir)).';
703     else
704         debug_disp('Non-coaxial')
705         ecc_forces = cylinder_force_eccentric(size1, size2, displ(cyldir), ecc, J1(cyldir
), J2(cyldir)).';
706         forces_out(cyldir)= ecc_forces(2);
707         forces_out(cylnotdir(1))= displ(cylnotdir(1))/ecc*ecc_forces(1);
708         forces_out(cylnotdir(2))= displ(cylnotdir(2))/ecc*ecc_forces(1);
709 % Need to check this division into components is correct...
710     end
712 end

```

5.2 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.

```

724 function force_out = single_magnet_force(displ)
726     force_components = nan([9 3]);
728     d_x = rotate_x_to_z(displ);
729     d_y = rotate_y_to_z(displ);
731     debug_disp(' ')
732     debug_disp('CALCULATING THINGS')
733     debug_disp('=====')
734     debug_disp('Displacement:')
735     debug_disp(displ)
736     debug_disp('Magnetisations:')
737     debug_disp(J1)
738     debug_disp(J2)
740     debug_disp('Forces x-x:')
741     force_components(1,:)= ...
742         rotate_z_to_x( cuboid_force_z_z(size1_x,size2_x,d_x,J1_x,J2_x));
744     debug_disp('Forces x-y:')
745     force_components(2,:)= ...
746         rotate_z_to_x( cuboid_force_z_y(size1_x,size2_x,d_x,J1_x,J2_x));
748     debug_disp('Forces x-z:')
749     force_components(3,:)= ...
750         rotate_z_to_x( cuboid_force_z_x(size1_x,size2_x,d_x,J1_x,J2_x));
752     debug_disp('Forces y-x:')
753     force_components(4,:)= ...
754         rotate_z_to_y( cuboid_force_z_x(size1_y,size2_y,d_y,J1_y,J2_y));
756     debug_disp('Forces y-y:')
757     force_components(5,:)= ...

```



```

758     rotate_z_to_y( cuboid_force_z_z(size1_y,size2_y,d_y,J1_y,J2_y));
760     debug_disp('Forces y-z:')
761     force_components(6,:)= ...
762     rotate_z_to_y( cuboid_force_z_y(size1_y,size2_y,d_y,J1_y,J2_y));
764     debug_disp('z-z force:')
765     force_components(9,:)= cuboid_force_z_z( size1,size2,displ,J1,J2 );
767     debug_disp('z-y force:')
768     force_components(8,:)= cuboid_force_z_y( size1,size2,displ,J1,J2 );
770     debug_disp('z-x force:')
771     force_components(7,:)= cuboid_force_z_x( size1,size2,displ,J1,J2 );

774     force_out = sum(force_components);
775 end

```

5.3 The single_magnet_torque() function

```

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

787     d_x = rotate_x_to_z(displ);
788     d_y = rotate_y_to_z(displ);

790     l_x = rotate_x_to_z(lever);
791     l_y = rotate_y_to_z(lever);

794     debug_disp(' ')
795     debug_disp('CALCULATING THINGS')
796     debug_disp('=====')
797     debug_disp('Displacement:')
798     debug_disp(displ)
799     debug_disp('Magnetisations:')
800     debug_disp(J1)
801     debug_disp(J2)

804     debug_disp('Torque: z-z:')
805     torque_components(:, :, 9)= cuboid_torque_z_z( size1,size2,displ,lever,J1,J2 );

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

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

813     debug_disp('Torques x-x:')
814     torque_components(:, :, 1)= ...
815     rotate_z_to_x( cuboid_torque_z_z(size1_x,size2_x,d_x,l_x,J1_x,J2_x));

817     debug_disp('Torques x-y:')
818     torque_components(:, :, 2)= ...

```

```

819     rotate_z_to_x( torques_calc_z_y(size1_x,size2_x,d_x,l_x,J1_x,J2_x));
821     debug_disp('Torques x-z:')
822     torque_components(:, :, 3)= ...
823     rotate_z_to_x( torques_calc_z_x(size1_x,size2_x,d_x,l_x,J1_x,J2_x));
825     debug_disp('Torques y-x:')
826     torque_components(:, :, 4)= ...
827     rotate_z_to_y( torques_calc_z_x(size1_y,size2_y,d_y,l_y,J1_y,J2_y));
829     debug_disp('Torques y-y:')
830     torque_components(:, :, 5)= ...
831     rotate_z_to_y( cuboid_torque_z_z(size1_y,size2_y,d_y,l_y,J1_y,J2_y));
833     debug_disp('Torques y-z:')
834     torque_components(:, :, 6)= ...
835     rotate_z_to_y( torques_calc_z_y(size1_y,size2_y,d_y,l_y,J1_y,J2_y));
837     torques_out = sum(torque_components,3);
838 end

844 function stiffness_out = single_magnet_stiffness(displ)
846     stiffness_components = nan([9 3]);

849     d_x = rotate_x_to_z(displ);
850     d_y = rotate_y_to_z(displ);

853     debug_disp(' ')
854     debug_disp('CALCULATING THINGS')
855     debug_disp('=====')
856     debug_disp('Displacement:')
857     debug_disp(displ)
858     debug_disp('Magnetisations:')
859     debug_disp(J1)
860     debug_disp(J2)

863     debug_disp('z-x stiffness:')
864     stiffness_components(7,:)= ...
865     stiffnesses_calc_z_x( size1,size2,displ,J1,J2 );

867     debug_disp('z-y stiffness:')
868     stiffness_components(8,:)= ...
869     stiffnesses_calc_z_y( size1,size2,displ,J1,J2 );

871     debug_disp('z-z stiffness:')
872     stiffness_components(9,:)= ...
873     stiffnesses_calc_z_z( size1,size2,displ,J1,J2 );

875     debug_disp('x-x stiffness:')
876     stiffness_components(1,:)= ...
877     swap_x_z( stiffnesses_calc_z_z( size1_x,size2_x,d_x,J1_x,J2_x ));

879     debug_disp('x-y stiffness:')
880     stiffness_components(2,:)= ...
881     swap_x_z( stiffnesses_calc_z_y( size1_x,size2_x,d_x,J1_x,J2_x ));

```

```

883     debug_disp('x-z stiffness:')
884     stiffness_components(3,:)= ...
885         swap_x_z( stiffnesses_calc_z_x( size1_x,size2_x,d_x,J1_x,J2_x ));
887     debug_disp('y-x stiffness:')
888     stiffness_components(4,:)= ...
889         swap_y_z( stiffnesses_calc_z_x( size1_y,size2_y,d_y,J1_y,J2_y ));
891     debug_disp('y-y stiffness:')
892     stiffness_components(5,:)= ...
893         swap_y_z( stiffnesses_calc_z_x( size1_y,size2_y,d_y,J1_y,J2_y ));
895     debug_disp('y-z stiffness:')
896     stiffness_components(6,:)= ...
897         swap_y_z( stiffnesses_calc_z_y( size1_y,size2_y,d_y,J1_y,J2_y ));
899     stiffness_out = sum(stiffness_components);
900 end

```

5.4 The stiffnesses_calc_z_z() function

```

909 function calc_out = stiffnesses_calc_z_z(size1,size2,offset,J1,J2)
911     J1 = J1(3);
912     J2 = J2(3);
914
915     if (J1==0 || J2==0)
916         debug_disp('Zero magnetisation.')
917         calc_out = [0; 0; 0];
918         return;
919     end
921
922     u = offset(1)+ size2(1)*(-1).^index_j - size1(1)*(-1).^index_i;
923     v = offset(2)+ size2(2)*(-1).^index_l - size1(2)*(-1).^index_k;
924     w = offset(3)+ size2(3)*(-1).^index_q - size1(3)*(-1).^index_p;
925     r = sqrt(u.^2+v.^2+w.^2);
927
928     component_x = - r - (u.^2 .*v)./(u.^2+w.^2)- v.*log(r-v);
929
930     component_y = - r - (v.^2 .*u)./(v.^2+w.^2)- u.*log(r-u);
931
932     component_z = - component_x - component_y;
933
934     component_x = index_sum.*component_x;
935     component_y = index_sum.*component_y;
936     component_z = index_sum.*component_z;
938
939     calc_out = J1*J2*magconst .* ...
940         [ sum(component_x(:));
941           sum(component_y(:));
942           sum(component_z(:))] ;
944
945     debug_disp(calc_out')
946 end

```

5.5 The `stiffnesses_calc_z_y()` function

```
949 function calc_out = stiffnesses_calc_z_y(size1,size2,offset,J1,J2)
951     J1 = J1(3);
952     J2 = J2(2);
955     if (J1==0 || J2==0)
956         debug_disp('Zero magnetisation.')
957         calc_out = [0; 0; 0];
958         return;
959     end
961     u = offset(1)+ size2(1)*(-1).^index_j - size1(1)*(-1).^index_i;
962     v = offset(2)+ size2(2)*(-1).^index_l - size1(2)*(-1).^index_k;
963     w = offset(3)+ size2(3)*(-1).^index_q - size1(3)*(-1).^index_p;
964     r = sqrt(u.^2+v.^2+w.^2);
966     component_x = ((u.^2 .*v)./(u.^2 + v.^2))+ (u.^2 .*w)./(u.^2 + w.^2)...
967         - u.*atan1(v.*w,r.*u)+ multiply_x_log_y( w , r + v )+ ...
968         + multiply_x_log_y( v , r + w );
969     component_y = - v/2 + (u.^2 .*v)./(u.^2 + v.^2)- (u.*v.*w)./(v.^2 + w.^2)...
970         - u.*atan1(u.*w,r.*v)- multiply_x_log_y( v , r + w );
971     component_z = - component_x - component_y;
973     component_x = index_sum.*component_x;
974     component_y = index_sum.*component_y;
975     component_z = index_sum.*component_z;
977     calc_out = J1*J2*magconst .* ...
978         [ sum(component_x(:));
979           sum(component_y(:));
980           sum(component_z(:)) ] ;
982     debug_disp(calc_out')
```

5.5.1 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.

5.5.2 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.)

```
994 function out = multiply_x_log_y(x,y)
995     out = x.*log(y);
996     out(~isfinite(out))=0;
997 end
```

5.5.3 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.

```
1004     function out = atan1(x,y)
1005         out = zeros(size(x));
1006         ind = x~=0 & y~=0;
1007         out(ind)= atan(x(ind)./y(ind));
1008     end
1011 end
```

5.6 The stiffnesses_calc_z_x() function

```
1017     function calc_out = stiffnesses_calc_z_x(size1,size2,offset,J1,J2)
1019         stiffnesses_xyz = stiffnesses_calc_z_y(...
1020             swap_x_y(size1), swap_x_y(size2), rotate_x_to_y(offset),...
1021             J1, rotate_x_to_y(J2));
1023         calc_out = swap_x_y(stiffnesses_xyz);
1025     end
```

5.7 The torques_calc_z_y() function

```
1030     function calc_out = torques_calc_z_y(size1,size2,offset,lever,J1,J2)
1032         if J1(3)~=0 && J2(2)~=0
1033             error('Torques cannot be calculated for orthogonal magnets yet.')
1034         end
1036         calc_out = 0*offset;
1038     end
```

5.8 The torques_calc_z_x() function

```
1042     function calc_out = torques_calc_z_x(size1,size2,offset,lever,J1,J2)
1044         if J1(3)~=0 && J2(1)~=0
1045             error('Torques cannot be calculated for orthogonal magnets yet.')
1046         end
1048         calc_out = 0*offset;
1050     end
```

5.9 The forces_magcyl_shell_calc() function

```
1055 function Fz = forces_magcyl_shell_calc(magsize,coilsize,displ,Jmag,Nrz,I)
1057     Jcoil = 4*pi*1e-7*Nrz(2)*I/coil.dim(3);
1059     shell_forces = nan([length(displ)Nrz(1)]);
1061     for rr = 1:Nrz(1)
1063         this_radius = coilsize(1)+(rr-1)/(Nrz(1)-1)*(coilsize(2)-coilsize(1));
1064         shell_size = [this_radius, coilsize(3)];
1066         shell_forces(:,rr)= cylinder_force_coaxial(magsize,shell_size,displ,Jmag,Jcoil);

1068     end
1070     Fz = sum(shell_forces,2);
1072 end

1076 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

```
1100 function forces_xyz = cuboid_force_z_z(size1,size2,offset,J1,J2)
1102 magconst = 1/(4*pi*(4*pi*1e-7));
1104 J1 = J1(3);
1105 J2 = J2(3);
1107 if ( abs(J1)<eps || abs(J2)<eps )
1108     forces_xyz = [0; 0; 0];
1109     return;
1110 end
1112 component_x = 0;
1113 component_y = 0;
```

```

1114 component_z = 0;
1116 for ii = [1 -1]
1117     for jj = [1 -1]
1118         for kk = [1 -1]
1119             for ll = [1 -1]
1120                 for pp = [1 -1]
1121                     for qq = [1 -1]
1123                         u = offset(1)+ size2(1)*jj - size1(1)*ii;
1124                         v = offset(2)+ size2(2)*ll - size1(2)*kk;
1125                         w = offset(3)+ size2(3)*qq - size1(3)*pp;
1126                         r = sqrt(u.^2+v.^2+w.^2);
1128                         if w == 0
1129                             atan_term = 0;
1130                         else
1131                             atan_term = atan(u.*v./(r.*w));
1132                         end
1133                         if abs(r-u)< eps
1134                             log_ru = 0;
1135                         else
1136                             log_ru = log(r-u);
1137                         end
1138                         if abs(r-v)< eps
1139                             log_rv = 0;
1140                         else
1141                             log_rv = log(r-v);
1142                         end
1144                         cx = ...
1145                             + 0.5*(v.^2-w.^2).*log_ru ...
1146                             + u.*v.*log_rv ...
1147                             + v.*w.*atan_term...
1148                             + 0.5*r.*u;
1150                         cy = ...
1151                             + 0.5*(u.^2-w.^2).*log_rv ...
1152                             + u.*v.*log_ru ...
1153                             + u.*w.*atan_term ...
1154                             + 0.5*r.*v;
1156                         cz = ...
1157                             - u.*w.*log_ru ...
1158                             - v.*w.*log_rv ...
1159                             + u.*v.*atan_term ...
1160                             - r.*w;
1162                         ind_sum = ii*jj*kk*ll*pp*qq;
1163                         component_x = component_x + ind_sum.*cx;
1164                         component_y = component_y + ind_sum.*cy;
1165                         component_z = component_z + ind_sum.*cz;
1167                     end
1168                 end
1169             end

```

```

1170     end
1171 end
1172 end

1174 forces_xyz = J1*J2*magconst.*[component_x; component_y; component_z];

1176 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

```

1205 function forces_xyz = cuboid_force_z_y(size1,size2,offset,J1,J2)

1207     J1 = J1(3);
1208     J2 = J2(2);

1210     if ( abs(J1)<eps || abs(J2)<eps )
1211         forces_xyz = [0; 0; 0];
1212         return;
1213     end

1215     component_x = 0;
1216     component_y = 0;
1217     component_z = 0;

1219     for ii = [1 -1]
1220         for jj = [1 -1]
1221             for kk = [1 -1]
1222                 for ll = [1 -1]
1223                     for pp = [1 -1]
1224                         for qq = [1 -1]

1226                             ind_sum = ii*jj*kk*ll*pp*qq;

1228                             u = offset(1)+ size2(1)*jj - size1(1)*ii;
1229                             v = offset(2)+ size2(2)*ll - size1(2)*kk;
1230                             w = offset(3)+ size2(3)*qq - size1(3)*pp;
1231                             r = sqrt(u.^2+v.^2+w.^2);

1233                             if u == 0
1234                                 atan_term_u = 0;
1235                             else
1236                                 atan_term_u = atan(v.*w./(r.*u));
1237                             end

```



```

1238     if v == 0
1239         atan_term_v = 0;
1240     else
1241         atan_term_v = atan(u.*w./(r.*v));
1242     end
1243     if w == 0
1244         atan_term_w = 0;
1245     else
1246         atan_term_w = atan(u.*v./(r.*w));
1247     end
1249     if abs(r-u)< eps
1250         log_ru = 0;
1251     else
1252         log_ru = log(r-u);
1253     end
1254     if abs(r+w)< eps
1255         log_rw = 0;
1256     else
1257         log_rw = log(r+w);
1258     end
1259     if abs(r+v)< eps
1260         log_rv = 0;
1261     else
1262         log_rv = log(r+v);
1263     end
1265     cx = ...
1266         + v.*w.*log_ru ...
1267         - v.*u.*log_rw ...
1268         - u.*w.*log_rv ...
1269         + 0.5*u.^2.*atan_term_u ...
1270         + 0.5*v.^2.*atan_term_v ...
1271         + 0.5*w.^2.*atan_term_w;
1273     cy = ...
1274         - 0.5*(u.^2-v.^2).*log_rw ...
1275         + u.*w.*log_ru ...
1276         + u.*v.*atan_term_v ...
1277         + 0.5*w.*r;
1279     cz = ...
1280         - 0.5*(u.^2-w.^2).*log_rv ...
1281         + u.*v.*log_ru ...
1282         + u.*w.*atan_term_w ...
1283         + 0.5*v.* r;
1285     component_x = component_x + ind_sum.*cx;
1286     component_y = component_y + ind_sum.*cy;
1287     component_z = component_z + ind_sum.*cz;
1289     end
1290     end
1291     end
1292     end

```

```

1293     end
1294 end

1296     forces_xyz = J1*J2/(4*pi*(4*pi*1e-7))*[ component_x; component_y; component_z ];

1298 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

```

1324 function forces_xyz = cuboid_force_z_x(size1,size2,offset,J1,J2)

1326 J1 = J1(3);
1327 J2 = J2(1);

1329 if ( abs(J1)<eps || abs(J2)<eps )
1330     forces_xyz = [0; 0; 0];
1331     return;
1332 end

1334 component_x = 0;
1335 component_y = 0;
1336 component_z = 0;

1338 for ii = [1 -1]
1339     for jj = [1 -1]
1340         for kk = [1 -1]
1341             for ll = [1 -1]
1342                 for pp = [1 -1]
1343                     for qq = [1 -1]

1345                         ind_sum = ii*jj*kk*ll*pp*qq;

1347                         u = -offset(2)- size2(2)*jj + size1(2)*ii;
1348                         v = offset(1)+ size2(1)*ll - size1(1)*kk;
1349                         w = offset(3)+ size2(3)*qq - size1(3)*pp;
1350                         r = sqrt(u.^2+v.^2+w.^2);

1352                         if u == 0
1353                             atan_term_u = 0;
1354                         else
1355                             atan_term_u = atan(v.*w./(r.*u));
1356                         end
1357                         if v == 0
1358                             atan_term_v = 0;
1359                         else

```

```

1360         atan_term_v = atan(u.*w./(r.*v));
1361     end
1362     if w == 0
1363         atan_term_w = 0;
1364     else
1365         atan_term_w = atan(u.*v./(r.*w));
1366     end
1367
1368     if abs(r-u)< eps
1369         log_ru = 0;
1370     else
1371         log_ru = log(r-u);
1372     end
1373     if abs(r+w)< eps
1374         log_rw = 0;
1375     else
1376         log_rw = log(r+w);
1377     end
1378     if abs(r+v)< eps
1379         log_rv = 0;
1380     else
1381         log_rv = log(r+v);
1382     end
1383
1384     CX = ...
1385         + v.*w.*log_ru ...
1386         - v.*u.*log_rw ...
1387         - u.*w.*log_rv ...
1388         + 0.5*u.^2.*atan_term_u ...
1389         + 0.5*v.^2.*atan_term_v ...
1390         + 0.5*w.^2.*atan_term_w;
1391
1392     cy = ...
1393         - 0.5*(u.^2-v.^2).*log_rw ...
1394         + u.*w.*log_ru ...
1395         + u.*v.*atan_term_v ...
1396         + 0.5*w.*r;
1397
1398     CZ = ...
1399         - 0.5*(u.^2-w.^2).*log_rv ...
1400         + u.*v.*log_ru ...
1401         + u.*w.*atan_term_w ...
1402         + 0.5*v.* r;
1403
1404     component_x = component_x + ind_sum.*cx;
1405     component_y = component_y + ind_sum.*cy;
1406     component_z = component_z + ind_sum.*cz;
1407
1408     end
1409     end
1410     end
1411     end
1412     end
1413     end

```

```

1415 forces_xyz = J1*J2/(4*pi*(4*pi*1e-7))*[ component_y; -component_x; component_z ];
1417 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 $0 \leq m \leq 1$.

```

1654 function [K,E,PI] = ellipkepi(a,m)

1656 a1 = 1;
1657 g1 = sqrt(1-m);
1658 p1 = sqrt(1-a);
1659 q1 = 1;
1660 w1 = 1;

1662 nn = 0;
1663 qq = 1;
1664 ww = m;

1666 while max(abs(w1(:)))> eps || max(abs(q1(:)))> eps

1668 % Update from previous loop
1669 a0 = a1;
1670 g0 = g1;
1671 p0 = p1;
1672 q0 = q1;

1674 % for Elliptic I
1675 a1 = (a0+g0)/2;
1676 g1 = sqrt(a0.*g0);

1678 % for Elliptic II
1679 nn = nn + 1;
1680 d1 = (a0-g0)/2;
1681 w1 = 2^nn*d1.^2;
1682 ww = ww + w1;

1684 % for Elliptic III
1685 rr = p0.^2+a0.*g0;
1686 p1 = rr./p0/2;
1687 q1 = q0.*(p0.^2-a0.*g0)./rr/2;
1688 qq = qq + q1;

1690 end

1692 K = 1./a1*pi/2;
1693 E = K.*(1-ww/2);
1694 PI = K.*(1+a./(2-2*a).*qq);

1696 im = find(m == 1);
1697 if ~isempty(im)

```

```

1698 K(im) = inf;
1699 E(im) = ones(length(im),1);
1700 PI(im) = inf;
1701 end
1703 end

```

8 Magnet arrays

8.1 The multipoleforces() function

```

1715 function [varargout] = multipoleforces(fixed_array, float_array, displ, varargin)
1717 debug_disp = @(str)disp([]);
1718 calc_force_bool = false;
1719 calc_stiffness_bool = false;
1720 calc_torque_bool = false;
1722 for ii = 1:length(varargin)
1723     switch varargin{ii}
1724         case 'debug',    debug_disp = @(str)disp(str);
1725         case 'force',    calc_force_bool = true;
1726         case 'stiffness', calc_stiffness_bool = true;
1727         case 'torque',    calc_torque_bool = true;
1728         otherwise
1729             error(['Unknown calculation option ''',varargin{ii},'''])
1730         end
1731     end
1733 if ~calc_force_bool && ~calc_stiffness_bool && ~calc_torque_bool
1734     varargin{end+1} = 'force';
1735     calc_force_bool = true;
1736 end
1739 if size(displ,1)== 3
1740     % all good
1741 elseif size(displ,2)== 3
1742     displ = transpose(displ);
1743 else
1744     error(['Displacements matrix should be of size (3, D)',...
1745         'where D is the number of displacements.'])
1746 end
1748 Ndispl = size(displ,2);
1750 if calc_force_bool
1751     forces_out = nan([3 Ndispl]);
1752 end
1754 if calc_stiffness_bool
1755     stiffnesses_out = nan([3 Ndispl]);
1756 end

```

```

1758 if calc_torque_bool
1759     torques_out = nan([3 Ndispl]);
1760 end

1763 part = @(x,y)x(y);

1765 fixed_array = complete_array_from_input(fixed_array);
1766 float_array = complete_array_from_input(float_array);

1768 if calc_force_bool
1769     array_forces = nan([3 Ndispl fixed_array.total float_array.total]);
1770 end

1772 if calc_stiffness_bool
1773     array_stiffnesses = nan([3 Ndispl fixed_array.total float_array.total]);
1774 end

1776 displ_from_array_corners = displ ...
1777     + repmat(fixed_array.size/2,[1 Ndispl])...
1778     - repmat(float_array.size/2,[1 Ndispl]);

1781 for ii = 1:fixed_array.total
1783     fixed_magnet = magnetdefine(...
1784         'type', 'cuboid',...
1785         'dim',   fixed_array.dim(ii,:), ...
1786         'magn',  fixed_array.magn(ii), ...
1787         'magdir', fixed_array.magdir(ii,:)...
1788     );
1790     for jj = 1:float_array.total
1792         float_magnet = magnetdefine(...
1793             'type', 'cuboid',...
1794             'dim',   float_array.dim(jj,:), ...
1795             'magn',  float_array.magn(jj), ...
1796             'magdir', float_array.magdir(jj,:)...
1797         );
1799         mag_displ = displ_from_array_corners ...
1800             - repmat(fixed_array.magloc(ii,:),[1 Ndispl])...
1801             + repmat(float_array.magloc(jj,:),[1 Ndispl]);

1803         if calc_force_bool && ~calc_stiffness_bool
1804             array_forces(:, :, ii, jj) = ...
1805                 magnetforces(fixed_magnet, float_magnet, mag_displ, varargin{:});
1806         elseif calc_stiffness_bool && ~calc_force_bool
1807             array_stiffnesses(:, :, ii, jj) = ...
1808                 magnetforces(fixed_magnet, float_magnet, mag_displ, varargin{:});
1809         else
1810             [array_forces(:, :, ii, jj) array_stiffnesses(:, :, ii, jj)] = ...
1811                 magnetforces(fixed_magnet, float_magnet, mag_displ, varargin{:});
1812         end
1814     end
1815 end

1817 if calc_force_bool

```

```

1818     forces_out = sum(sum(array_forces,4),3);
1819 end

1821 if calc_stiffness_bool
1822     stiffnesses_out = sum(sum(array_stiffnesses,4),3);
1823 end

1826 varargout = {};

1828 for ii = 1:length(varargin)
1829     switch varargin{ii}
1830         case 'force'
1831             varargout{end+1} = forces_out;

1833         case 'stiffness'
1834             varargout{end+1} = stiffnesses_out;

1836         case 'torque'
1837             varargout{end+1} = torques_out;
1838     end
1839 end

1845 function array = complete_array_from_input(array)

1847 if ~isfield(array,'type')
1848     array.type = 'generic';
1849 end

1852 if ~isfield(array,'face')
1853     array.face = 'undefined';
1854 end

1856 linear_index = 0;
1857 planar_index = [0 0];

1859 switch array.type
1860     case 'generic'
1861     case 'linear',           linear_index = 1;
1862     case 'linear-quasi',    linear_index = 1;
1863     case 'planar',         planar_index = [1 2];
1864     case 'quasi-halbach',  planar_index = [1 2];
1865     case 'patchwork',      planar_index = [1 2];
1866     otherwise
1867         error(['Unknown array type ''',array.type,('').'])
1868 end

1870 if ~isequal(array.type,'generic')
1871     if linear_index == 1
1872         if ~isfield(array,'align')
1873             array.align = 'x';
1874         end
1875         switch array.align
1876             case 'x', linear_index = 1;
1877             case 'y', linear_index = 2;
1878             case 'z', linear_index = 3;

```

```

1879     otherwise
1880         error('Alignment for linear array must be 'x'', 'y'', or 'z''.')
1881     end
1882 else
1883     if ~isfield(array,'align')
1884         array.align = 'xy';
1885     end
1886     switch array.align
1887         case 'xy', planar_index = [1 2];
1888         case 'yz', planar_index = [2 3];
1889         case 'xz', planar_index = [1 3];
1890     otherwise
1891         error('Alignment for planar array must be 'xy'', 'yz'', or 'xz''.')
1892     end
1893 end
1894 end

1896 switch array.face
1897     case {'+x','-x'}, facing_index = 1;
1898     case {'+y','-y'}, facing_index = 2;
1899     case {'up','down'}, facing_index = 3;
1900     case {'+z','-z'}, facing_index = 3;
1901     case 'undefined', facing_index = 0;
1902 end

1904 if linear_index ~= 0
1905     if linear_index == facing_index
1906         error('Arrays cannot face into their alignment direction.')
1907     end
1908 elseif ~isequal( planar_index, [0 0] )
1909     if any( planar_index == facing_index )
1910         error('Planar-type arrays can only face into their orthogonal direction')
1911     end
1912 end

1915 switch array.type
1916     case 'linear'

1918 array = extrapolate_variables(array);

1920 array.mcount = ones(1,3);
1921 array.mcount(linear_index)= array.Nmag;

1923     case 'linear-quasi'

1926 if isfield(array,'ratio')&& isfield(array,'mlength')
1927     error('Cannot specify both 'ratio'and 'mlength''.')
1928 elseif ~isfield(array,'ratio')&& ~isfield(array,'mlength')
1929     error('Must specify either 'ratio'or 'mlength''.')
1930 end

1933 array.Nmag_per_wave = 4;
1934 array.magdir_rotate = 90;

1936 if isfield(array,'Nwaves')

```



```

1937     array.Nmag = array.Nmag_per_wave*array.Nwaves+1;
1938 else
1939     error('''Nwaves''must be specified.')
1940 end

1942 if isfield(array,'mlength')
1943     if numel(array.mlength)~=2
1944         error('''mlength''must have length two for linear-quasi arrays.')
1945     end
1946     array.ratio = array.mlength(2)/array.mlength(1);
1947 else
1948     if isfield(array,'length')
1949         array.mlength(1)= 2*array.length/(array.Nmag*(1+array.ratio)+1-array.ratio);
1950         array.mlength(2)= array.mlength(1)*array.ratio;
1951     else
1952         error('''length''must be specified.')
1953     end
1954 end

1956 array.mcount = ones(1,3);
1957 array.mcount(linear_index)= array.Nmag;

1959 array.msize = nan([array.mcount 3]);

1961 [sindex_x sindex_y sindex_z] = ...
1962     meshgrid(1:array.mcount(1), 1:array.mcount(2), 1:array.mcount(3));

1966 all_indices = [1 1 1];
1967 all_indices(linear_index)= 0;
1968 all_indices(facing_index)= 0;
1969 width_index = find(all_indices);

1971 for ii = 1:array.Nmag
1972     array.msize(sindex_x(ii),sindex_y(ii),sindex_z(ii),linear_index)= ...
1973         array.mlength(mod(ii-1,2)+1);
1974     array.msize(sindex_x(ii),sindex_y(ii),sindex_z(ii),facing_index)= ...
1975         array.height;
1976     array.msize(sindex_x(ii),sindex_y(ii),sindex_z(ii),width_index)= ...
1977         array.width;
1978 end

1981 case 'planar'

1983 if isfield(array,'length')
1984     if length(array.length)== 1
1985         if isfield(array,'width')
1986             array.length = [ array.length array.width ];
1987         else
1988             array.length = [ array.length array.length ];
1989         end
1990     end
1991 end

1993 if isfield(array,'mlength')
1994     if length(array.mlength)== 1

```

```

1995     if isfield(array.mwidth)
1996         array.mlength = [ array.mlength array.mwidth ];
1997     else
1998         array.mlength = [ array.mlength array.mlength ];
1999     end
2000 end
2001 end

2003 var_names = {'length','mlength','wavelength','Nwaves',...
2004             'Nmag','Nmag_per_wave','magdir_rotate'};

2006 tmp_array1 = struct();
2007 tmp_array2 = struct();
2008 var_index = zeros(size(var_names));

2010 for iii = 1:length(var_names)
2011     if isfield(array,var_names(iii))
2012         tmp_array1.(var_names{iii})= array.(var_names{iii})(1);
2013         tmp_array2.(var_names{iii})= array.(var_names{iii})(end);
2014     else
2015         var_index(iii)= 1;
2016     end
2017 end

2019 tmp_array1 = extrapolate_variables(tmp_array1);
2020 tmp_array2 = extrapolate_variables(tmp_array2);

2022 for iii = find(var_index)
2023     array.(var_names{iii})= [tmp_array1.(var_names{iii})tmp_array2.(var_names{iii})];
2024 end

2026 array.width = array.length(2);
2027 array.length = array.length(1);

2029 array.mwidth = array.mlength(2);
2030 array.mlength = array.mlength(1);

2032 array.mcount = ones(1,3);
2033 array.mcount(planar_index)= array.Nmag;

2035 case 'quasi-halbach'

2037 if isfield(array,'mcount')
2038     if numel(array.mcount)~=3
2039         error(''mcount''must always have three elements.')
2040     end
2041 elseif isfield(array,'Nwaves')
2042     if numel(array.Nwaves)> 2
2043         error(''Nwaves''must have one or two elements only.')
2044     end
2045     array.mcount(facing_index)= 1;
2046     array.mcount(planar_index)= 4*array.Nwaves+1;
2047 elseif isfield(array,'Nmag')
2048     if numel(array.Nmag)> 2
2049         error(''Nmag''must have one or two elements only.')
2050     end
2051     array.mcount(facing_index)= 1;

```

```

2052     array.mcount(planar_index)= array.Nmag;
2053 else
2054     error('Must specify the number of magnets (''mcount''or ''Nmag'')or wavelengths (''
Nwaves'')')
2055 end

2057     case 'patchwork'

2059 if isfield(array,'mcount')
2060     if numel(array.mcount)~=3
2061         error(''mcount''must always have three elements.')
2062     end
2063 elseif isfield(array,'Nmag')
2064     if numel(array.Nmag)> 2
2065         error(''Nmag''must have one or two elements only.')
2066     end
2067     array.mcount(facing_index)= 1;
2068     array.mcount(planar_index)= array.Nmag;
2069 else
2070     error('Must specify the number of magnets (''mcount''or ''Nmag'')')
2071 end

2073 end

2076 array.total = prod(array.mcount);

2078 if ~isfield(array,'msize')
2079     array.msize = [NaN NaN NaN];
2080     if linear_index ~=0
2081         array.msize(linear_index)= array.mlength;
2082         array.msize(facing_index)= array.height;
2083         array.msize(isnan(array.msize))= array.width;
2084     elseif ~isequal( planar_index, [0 0] )
2085         array.msize(planar_index)= [array.mlength array.mwidth];
2086         array.msize(facing_index)= array.height;
2087     else
2088         error('The array property ''msize''is not defined and I have no way to infer it.'
)
2089     end
2090 elseif numel(array.msize)== 1
2091     array.msize = repmat(array.msize,[3 1]);
2092 end

2094 if numel(array.msize)== 3
2095     array.msize_array = ...
2096     repmat(reshape(array.msize,[1 1 1 3]), array.mcount);
2097 else
2098     if isequal([array.mcount 3],size(array.msize))
2099         array.msize_array = array.msize;
2100     else
2101         error('Magnet size ''msize''must have three elements (or one element for a cube magnet
).')
2102     end
2103 end

```

```

2104 array.dim = reshape(array.msize_array, [array.total 3]);
2106 if ~isfield(array,'mgap')
2107     array.mgap = [0; 0; 0];
2108 elseif length(array.mgap)== 1
2109     array.mgap = repmat(array.mgap,[3 1]);
2110 end

2114 if ~isfield(array,'magn')
2115     if isfield(array,'grade')
2116         array.magn = grade2magn(array.grade);
2117     else
2118         array.magn = 1;
2119     end
2120 end

2122 if length(array.magn)== 1
2123     array.magn = repmat(array.magn,[array.total 1]);
2124 else
2125     error('Magnetisation magnitude ''magn'' must be a single value.')
2126 end

2130 if ~isfield(array,'magdir_fn')
2132     if ~isfield(array,'face')
2133         array.face = '+z';
2134     end

2136     switch array.face
2137     case {'up','+z','+y','+x'}, magdir_rotate_sign = 1;
2138     case {'down','-z','-y','-x'}, magdir_rotate_sign = -1;
2139     end

2141     if ~isfield(array,'magdir_first')
2142         array.magdir_first = magdir_rotate_sign*90;
2143     end

2145     magdir_fn_comp{1} = @(ii,jj,kk)0;
2146     magdir_fn_comp{2} = @(ii,jj,kk)0;
2147     magdir_fn_comp{3} = @(ii,jj,kk)0;

2149     switch array.type
2150     case 'linear'
2151         magdir_theta = @(nn)...
2152             array.magdir_first+magdir_rotate_sign*array.magdir_rotate*(nn-1);

2154         magdir_fn_comp{linear_index} = @(ii,jj,kk)...
2155             cosd(magdir_theta(part([ii,jj,kk],linear_index)));

2157         magdir_fn_comp{facing_index} = @(ii,jj,kk)...
2158             sind(magdir_theta(part([ii,jj,kk],linear_index)));

2160     case 'linear-quasi'

2162         magdir_theta = @(nn)...
2163             array.magdir_first+magdir_rotate_sign*90*(nn-1);

```

```

2165     magdir_fn_comp{linear_index} = @(ii,jj,kk)...
2166         cosd(magdir_theta(part([ii,jj,kk],linear_index))));
2168     magdir_fn_comp{facing_index} = @(ii,jj,kk)...
2169         sind(magdir_theta(part([ii,jj,kk],linear_index))));
2171 case 'planar'
2173     magdir_theta = @(nn)...
2174         array.magdir_first(1)+magdir_rotate_sign*array.magdir_rotate(1)*(nn-1);
2176     magdir_phi = @(nn)...
2177         array.magdir_first(end)+magdir_rotate_sign*array.magdir_rotate(end)*(nn-1);
2179     magdir_fn_comp{planar_index(1)} = @(ii,jj,kk)...
2180         cosd(magdir_theta(part([ii,jj,kk],planar_index(2)))));
2182     magdir_fn_comp{planar_index(2)} = @(ii,jj,kk)...
2183         cosd(magdir_phi(part([ii,jj,kk],planar_index(1)))));
2185     magdir_fn_comp{facing_index} = @(ii,jj,kk)...
2186         sind(magdir_theta(part([ii,jj,kk],planar_index(1))))...
2187         + sind(magdir_phi(part([ii,jj,kk],planar_index(2)))));
2189 case 'patchwork'
2191     magdir_fn_comp{planar_index(1)} = @(ii,jj,kk)0;
2193     magdir_fn_comp{planar_index(2)} = @(ii,jj,kk)0;
2195     magdir_fn_comp{facing_index} = @(ii,jj,kk)...
2196         magdir_rotate_sign*(-1)^( ...
2197             part([ii,jj,kk],planar_index(1))...
2198             + part([ii,jj,kk],planar_index(2))...
2199             + 1 ...
2200         );
2202 case 'quasi-halbach'
2204     magdir_fn_comp{planar_index(1)} = @(ii,jj,kk)...
2205         sind(90*part([ii,jj,kk],planar_index(1)))...
2206         * cosd(90*part([ii,jj,kk],planar_index(2)));
2208     magdir_fn_comp{planar_index(2)} = @(ii,jj,kk)...
2209         cosd(90*part([ii,jj,kk],planar_index(1)))...
2210         * sind(90*part([ii,jj,kk],planar_index(2)));
2212     magdir_fn_comp{facing_index} = @(ii,jj,kk)...
2213         magdir_rotate_sign ...
2214         * sind(90*part([ii,jj,kk],planar_index(1)))...
2215         * sind(90*part([ii,jj,kk],planar_index(2)));
2217 otherwise
2218     error('Array property ''magdir_fn''not defined and I have no way to infer it.')
2219 end
2221 array.magdir_fn = @(ii,jj,kk)...
2222     [ magdir_fn_comp{1}(ii,jj,kk)...
2223       magdir_fn_comp{2}(ii,jj,kk)...
2224       magdir_fn_comp{3}(ii,jj,kk)];

```

```

2226 end

2232 array.magloc = nan([array.total 3]);
2233 array.magdir = array.magloc;
2234 arrat.magloc_array = nan([array.mcount(1)array.mcount(2)array.mcount(3)3]);

2236 nn = 0;
2237 for iii = 1:array.mcount(1)
2238     for jjj = 1:array.mcount(2)
2239         for kkk = 1:array.mcount(3)
2240             nn = nn + 1;
2241             array.magdir(nn,:)= array.magdir_fn(iii,jjj,kkk);
2242         end
2243     end
2244 end

2246 magsep_x = zeros(size(array.mcount(1)));
2247 magsep_y = zeros(size(array.mcount(2)));
2248 magsep_z = zeros(size(array.mcount(3)));

2250 magsep_x(1)= array.msize_array(1,1,1,1)/2;
2251 magsep_y(1)= array.msize_array(1,1,1,2)/2;
2252 magsep_z(1)= array.msize_array(1,1,1,3)/2;

2254 for iii = 2:array.mcount(1)
2255     magsep_x(iii)= array.msize_array(iii-1,1,1,1)/2 ...
2256         + array.msize_array(iii ,1,1,1)/2 ;
2257 end
2258 for jjj = 2:array.mcount(2)
2259     magsep_y(jjj)= array.msize_array(1,jjj-1,1,2)/2 ...
2260         + array.msize_array(1,jjj ,1,2)/2 ;
2261 end
2262 for kkk = 2:array.mcount(3)
2263     magsep_z(kkk)= array.msize_array(1,1,kkk-1,3)/2 ...
2264         + array.msize_array(1,1,kkk ,3)/2 ;
2265 end

2267 magloc_x = cumsum(magsep_x);
2268 magloc_y = cumsum(magsep_y);
2269 magloc_z = cumsum(magsep_z);

2271 for iii = 1:array.mcount(1)
2272     for jjj = 1:array.mcount(2)
2273         for kkk = 1:array.mcount(3)
2274             array.magloc_array(iii,jjj,kkk,:)= ...
2275                 [magloc_x(iii); magloc_y(jjj); magloc_z(kkk)] ...
2276                 + [iii-1; jjj-1; kkk-1].*array.mgap;
2277         end
2278     end
2279 end
2280 array.magloc = reshape(array.magloc_array,[array.total 3]);

2282 array.size = squeeze( array.magloc_array(end,end,end,:)...
2283     - array.magloc_array(1,1,1,:)...

```

```

2284         + array.msize_array(1,1,1,:)/2 ...
2285         + array.msize_array(end,end,end,:)/2 );

2287 debug_disp('Magnetisation directions')
2288 debug_disp(array.magdir)

2290 debug_disp('Magnet locations:')
2291 debug_disp(array.magloc)

2294 end

2298 function array_out = extrapolate_variables(array)
2300 var_names = {'wavelength','length','Nwaves','mlength',...
2301             'Nmag','Nmag_per_wave','magdir_rotate'};

2303 if isfield(array,'Nwaves')
2304     mcount_extra = 1;
2305 else
2306     mcount_extra = 0;
2307 end

2309 if isfield(array,'mlength')
2310     mlength_adjust = false;
2311 else
2312     mlength_adjust = true;
2313 end

2315 variables = nan([7 1]);

2317 for iii = 1:length(var_names);
2318     if isfield(array,var_names(iii))
2319         variables(iii)= array.(var_names{iii});
2320     end
2321 end

2323 var_matrix = ...
2324     [1, 0, 0, -1, 0, -1, 0;
2325      0, 1, 0, -1, -1, 0, 0;
2326      0, 0, 1, 0, -1, 1, 0;
2327      0, 0, 0, 0, 0, 1, 1];

2329 var_results = [0 0 0 log(360)]';
2330 variables = log(variables);

2332 idx = ~isnan(variables);
2333 var_known = var_matrix(:,idx)*variables(idx);
2334 var_calc = var_matrix(:,~idx)\(var_results-var_known);
2335 variables(~idx)= var_calc;
2336 variables = exp(variables);

2338 for iii = 1:length(var_names);
2339     array.(var_names{iii})= variables(iii);
2340 end

2342 array.Nmag = round(array.Nmag)+ mcount_extra;
2343 array.Nmag_per_wave = round(array.Nmag_per_wave);

```

```
2345 if mlength_adjust
2346     array.mlength = array.mlength * (array.Nmag-mcount_extra)/array.Nmag;
2347 end
2349 array_out = array;
2351 end
2355 end
```