

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

Will Robertson

April 10, 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

I	User guide	3
1	Defining magnets and coils	3
1.1	Cuboid magnets	3
1.2	Cylindrical and ring magnets/coils	3
1.3	Coil—unfinished!	4
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 magnetdefine() function	8
4.1.1	The attempt3Mvector() function	10
4.1.2	The grade2magn() function	10
4.1.3	The make_unit_vector() function	11
5	The magnetforces() function	12
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	19
5.6	The stiffnesses_calc_z_z() function	20
5.7	The stiffnesses_calc_z_y() function	20
5.7.1	Helpers	21
5.7.2	The multiply_x_log_y() function	21
5.7.3	The atan1() function	21
5.8	The stiffnesses_calc_z_x() function	21
5.9	The torques_calc_z_y() function	22
5.10	The torques_calc_z_x() function	22
5.11	The forces_magcyl_shell_calc() function	22
6	Magnet interactions	22
6.1	The dipole_forcetorque() function	23
6.2	The cuboid_force_z_z() function	24
6.3	The cuboid_force_z_y() function	25
6.4	The cuboid_force_z_x() function	27
6.5	The cuboid_torque_z_z() function	29
7	Mathematical functions	35
7.1	The ellipkepi() function	35
8	Magnet arrays	37
8.1	The multipoleforces() function	37

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 (3×1) 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 (3×1) 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 (3×1) 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 (1×1) element specifying the magnet radius. For ring magnets, a (2×1) vector containing the inner and outer radius, resp.

'dim' A (2×1) or (3×1) 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 (3×1) 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 (3×1) 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 force¹ 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.

¹From 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 v2.0.² 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.³

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

³<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

35 mag = attempt3Mvector(mag, 'lever');
36 mag = make_unit_vector(mag, 'magdir');
37 mag = make_unit_vector(mag, 'dir');

    defaults

41 if ~isfield(mag, 'lever')
42     mag.lever = [0; 0; 0];
43 end

45 if strcmp(mag.type, 'cylinder')
46 else
47 end

50 switch mag.type
51     case 'cylinder', mag = definecylinder(mag);
52     case 'cuboid', mag = definecuboid(mag);
53     otherwise
```



```

54     error('Magnet type "%s" unknown',mag.type)
55 end

58 if isfield(mag,'magdir')&& isfield(mag,'magn')
59     mag.magM = mag.magdir*mag.magn;
60     mag.dipolemoment = 1/(4*pi*1e-7)*mag.magM*mag.volume;
61 end

63 mag.fndefined = true;

65 end

68 function mag = definecuboid(mag)

70     if ~isfield(mag,'magdir')
71         warning('Magnet direction ("magdir")not specified; assuming +z.')
72         mag.magdir = [0; 0; 1];
73     else
74         mag = make_unit_vector(mag,'magdir');
75     end

77     if isfield(mag,'dim')
78         mag.volume = prod(mag.dim);
79     end

81 end

84 function mag = definecylinder(mag)

86 % default to +Z magnetisation
87     if ~isfield(mag,'dir')
88         if ~isfield(mag,'magdir')
89             warning('Magnet direction and magnetisation direction ("dir" and "magdir")not
specified; assuming +z for both.')
90             mag.dir = [0; 0; 1];
91             mag.magdir = [0; 0; 1];
92         else
93             mag.dir = mag.magdir;
94         end
95     else
96         if ~isfield(mag,'magdir')
97             mag.magdir = mag.dir;
98         end
99     end

101 % convert from current/turns to equiv magnetisation:
102     if ~isfield(mag,'magn')
103         if isfield(mag,'turns')&& isfield(mag,'current')
104             mag.magn = 4*pi*1e-7*mag.turns*mag.current/mag.dim(2);
105         end
106     end

108     if isfield(mag,'radius')&& isfield(mag,'height')
109         mag.dim = [mag.radius(:); mag.height];
110     end

112     if isfield(mag,'dim')

```

```

114     if numel(mag.dim)== 3
115         mag.isring = true;
116         if mag.dim(2)<= mag.dim(1)
117             error('Ring radii must be defined as [ri ro] with ro > ri.')
118         end
119         mag.volume = pi*(mag.dim(2)^2-mag.dim(1)^2)*mag.dim(3);
120     else
121         mag.isring = false;
122         mag.volume = pi*mag.dim(1)^2*mag.dim(2);
123     end
125 end
127 end

```

4.1.1 The attempt3Mvector() function

If a series of vectors (column arrays) are stacked they should create a [3 M] size array. If [M 3] is entered, transpose it. (If size [3 3], leave as is!)

```

136 function mag = attempt3Mvector(mag,vecname)
138 if isfield(mag,vecname)
140     if (size(mag.(vecname),1)~=3)&& (size(mag.(vecname),2)==3)
141         mag.(vecname)= transpose(mag.(vecname)); % attempt [3 M] shape
142     end
144 end
146 end

```

4.1.2 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.

```

165 function magn = grade2magn(grade)
167 if isnumeric(grade)
168     magn = 2*sqrt(grade/100);
169 else
170     if strcmp(grade(1),'N')
171         grade = grade(2:end);
172     end

```

```

173     magn = 2*sqrt(str2double(grade)/100);
174 end
176 end

```

4.1.3 The make_unit_vector() function

```

180 function mag = make_unit_vector(mag,vecname)

```

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

```

184 if ~isfield(mag,vecname)
185     return
186 end

188 vec_in = mag.(vecname);
190 if isnumeric(vec_in)
192     if numel(vec_in)~= 3
193         error(['"',vecname,'" has wrong number of elements (should be 3x1 vector or string
input like '+x'+'.')])
194     end
195     norm_vec_in = norm(vec_in);
196     if norm_vec_in < eps
197         norm_vec_in = 1; % to avoid 0/0
198     end
199     vec = vec_in(:)/norm_vec_in;
201 elseif ischar(vec_in)
203     switch vec_in
204         case 'x'; vec = [1;0;0];
205         case 'y'; vec = [0;1;0];
206         case 'z'; vec = [0;0;1];
207         case '+x'; vec = [1;0;0];
208         case '+y'; vec = [0;1;0];
209         case '+z'; vec = [0;0;1];
210         case '-x'; vec = [-1; 0; 0];
211         case '-y'; vec = [ 0;-1; 0];
212         case '-z'; vec = [ 0; 0;-1];
213         otherwise, error('Vector string %s not understood.',vec);
214     end
216 else
217     error('Strange input (this shouldn't happen)')
218 end

220 mag.(vecname)= vec;
222 end

```

5 The magnetforces() function

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

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

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

470 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`.

```
478 calc_force_bool    = false;
479 calc_stiffness_bool = false;
480 calc_torque_bool   = false;

482 for iii = 1:length(varargin)
483     switch varargin{iii}
484         case 'force',    calc_force_bool    = true;
485         case 'stiffness', calc_stiffness_bool = true;
486         case 'torque',   calc_torque_bool   = true;
487         case 'x', warning("Options 'x','y','z'are no longer supported.");
488         case 'y', warning("Options 'x','y','z'are no longer supported.");
489         case 'z', warning("Options 'x','y','z'are no longer supported.");
490         otherwise
491             error(['Unknown calculation option ',varargin{iii},'])
492     end
493 end

495 if ~calc_force_bool && ~calc_stiffness_bool && ~calc_torque_bool
496     varargin{end+1} = 'force';
497     calc_force_bool = true;
498 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.

```
505 if size(displ,1)== 3
506     % all good
507 elseif size(displ,2)== 3
508     displ = transpose(displ);
509 else
510     error(['Displacements matrix should be of size (3, D)',...
511         'where D is the number of displacements.'])
512 end

514 Ndispl = size(displ,2);

516 if calc_force_bool
517     forces_out = nan([3 Ndispl]);
518 end

520 if calc_stiffness_bool
```

```

521     stiffnesses_out = nan([3 Ndispl]);
522 end
524 if calc_torque_bool
525     torques_out = nan([3 Ndispl]);
526 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.)

```

540 if ~isfield(magnet_fixed, 'fdefined')
541     magnet_fixed = magnetdefine(magnet_fixed);
542 end
543 if ~isfield(magnet_float, 'fdefined')
544     magnet_float = magnetdefine(magnet_float);
545 end

549 if strcmp(magnet_fixed.type, 'coil')
551     if ~strcmp(magnet_float.type, 'cylinder')
552         error('Coil/magnet forces can only be calculated for cylindrical magnets.')
553     end

555     coil = magnet_fixed;
556     magnet = magnet_float;
557     magtype = 'coil';
558     coil_sign = +1;
560 end

562 if strcmp(magnet_float.type, 'coil')
564     if ~strcmp(magnet_fixed.type, 'cylinder')
565         error('Coil/magnet forces can only be calculated for cylindrical magnets.')
566     end

568     coil = magnet_float;
569     magnet = magnet_fixed;
570     magtype = 'coil';
571     coil_sign = -1;
573 end

576 if ~strcmp(magnet_fixed.type, magnet_float.type)
577     error('Magnets must be of same type (cuboid/cuboid or cylinder/cylinder)')
578 end
579 magtype = magnet_fixed.type;

584 switch magtype
585     case 'cuboid'

```

```

587     size1 = magnet_fixed.dim(:)/2;
588     size2 = magnet_float.dim(:)/2;

590     swap_x_y = @(vec)vec([2 1 3],:);
591     swap_x_z = @(vec)vec([3 2 1],:);
592     swap_y_z = @(vec)vec([1 3 2],:);

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

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

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

603     size1_x = swap_x_z(size1);
604     size2_x = swap_x_z(size2);
605     J1_x     = rotate_x_to_z(magnet_fixed.magM);
606     J2_x     = rotate_x_to_z(magnet_float.magM);

608     size1_y = swap_y_z(size1);
609     size2_y = swap_y_z(size2);
610     J1_y     = rotate_y_to_z(magnet_fixed.magM);
611     J2_y     = rotate_y_to_z(magnet_float.magM);

613     if calc_force_bool
614         for iii = 1:Ndispl
615             forces_out(:,iii)= single_magnet_force(displ(:,iii));
616         end
617     end

619     if calc_stiffness_bool
620         for iii = 1:Ndispl
621             stiffnesses_out(:,iii)= single_magnet_stiffness(displ(:,iii));
622         end
623     end

625     if calc_torque_bool
626         torques_out = single_magnet_torque(displ,magnet_float.lever);
627     end

629     case 'cylinder'

631         if any(abs(magnet_fixed.dir)~= abs(magnet_float.dir))
632             error('Cylindrical magnets must be oriented in the same direction')
633         end
634         if any(abs(magnet_fixed.magdir)~= abs(magnet_float.magdir))
635             error('Cylindrical magnets must be oriented in the same direction')
636         end
637         if any(abs(magnet_fixed.dir)~= abs(magnet_fixed.magdir))
638             error('Cylindrical magnets must be magnetised in the same direction as their orientation
        ')
639         end
640         if any(abs(magnet_float.dir)~= abs(magnet_float.magdir))
641             error('Cylindrical magnets must be magnetised in the same direction as their orientation
        ')
642         end

```

```

644     cyldir    = find(magnet_float.magdir ~= 0);
645     cylnotdir = find(magnet_float.magdir == 0);
646     if length(cyldir)~= 1
647         error('Cylindrical magnets must be aligned in one of the x, y or z directions')
648     end
649
650     if calc_force_bool
651         if magnet_fixed.isring && magnet_float.isring
652             for iii = 1:Ndispl
653                 forces_out(:,iii)= single_magnet_ring_force(displ(:,iii));
654             end
655         else
656             for iii = 1:Ndispl
657                 forces_out(:,iii)= single_magnet_cyl_force(displ(:,iii));
658             end
659         end
660     end
661
662     if calc_stiffness_bool
663         error('Stiffness cannot be calculated for cylindrical magnets yet.')
664     end
665
666     if calc_torque_bool
667         error('Torques cannot be calculated for cylindrical magnets yet.')
668     end
669
670     case 'coil'
671
672         warning('Code for coils in Matlab has never been completed :( See the Mathematica
673         code for more details')!
674         for iii = 1:Ndispl
675             forces_out(:,iii)= coil_sign*coil.dir*...
676             forces_magcyl_shell_calc(...
677             magnet.dim, ...
678             coil.dim, ...
679             squeeze(displ(cyldir,:)), ...
680             magnet.magM(cyldir), ...
681             coil.current, ...
682             coil.turns);
683         end
684     end
685 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.

```

698 argcount = 0;
699
700 for iii = 1:length(varargin)
701     switch varargin{iii}
702         case 'force',    argcount = argcount+1;
703         case 'stiffness', argcount = argcount+1;
704         case 'torque',   argcount = argcount+1;

```

```

705     end
706 end

708 varargout = cell(argcount,1);
710 argcount = 0;
712 for iii = 1:length(varargin)
713     switch varargin{iii}
714         case 'force',    argcount = argcount+1; varargout{argcount} = forces_out;
715         case 'stiffness', argcount = argcount+1; varargout{argcount} = stiffnesses_out;
716         case 'torque',    argcount = argcount+1; varargout{argcount} = torques_out;
717     end
718 end

```

That is the end of the main function.

5.1 The single_magnet_cyl_force() function

```

727 function forces_out = single_magnet_cyl_force(displ)
729     forces_out = nan(size(displ));
731     ecc = sqrt(sum(displ(cylnotdir).^2));
733     if ecc < eps
734         magdir = [0;0;0];
735         magdir(cyldir)= 1;
736         forces_out = magdir*cylinder_force_coaxial(magnet_fixed.magM(cyldir), magnet_float
.magM(cyldir), magnet_fixed.dim(1), magnet_float.dim(1), magnet_fixed.dim(2), magnet_float
.dim(2), displ(cyldir)).';
737     else
738         ecc_forces = cylinder_force_eccentric(magnet_fixed.dim, magnet_float.dim, displ(
cyldir), ecc, magnet_fixed.magM(cyldir), magnet_float.magM(cyldir)).';
739         forces_out(cyldir)= ecc_forces(2);
740         forces_out(cylnotdir(1))= displ(cylnotdir(1))/ecc*ecc_forces(1);
741         forces_out(cylnotdir(2))= displ(cylnotdir(2))/ecc*ecc_forces(1);
742 % Need to check this division into components is correct...
743     end
745 end

```

5.2 The single_magnet_ring_force() function

```

749 function forces_out = single_magnet_ring_force(displ)
751     forces_out = nan(size(displ));
753     ecc = sqrt(sum(displ(cylnotdir).^2));
755     if ecc < eps
756         magdir = [0;0;0];

```



```

757     magdir(cyldir)= 1;
758     forces11 = magdir*cylinder_force_coaxial(-magnet_fixed.magM(cyldir), -magnet_float
.magM(cyldir), magnet_fixed.dim(1), magnet_float.dim(1), magnet_fixed.dim(3), magnet_float
.dim(3), displ(cyldir)).';
759     forces12 = magdir*cylinder_force_coaxial(-magnet_fixed.magM(cyldir), +magnet_float
.magM(cyldir), magnet_fixed.dim(1), magnet_float.dim(2), magnet_fixed.dim(3), magnet_float
.dim(3), displ(cyldir)).';
760     forces21 = magdir*cylinder_force_coaxial(+magnet_fixed.magM(cyldir), -magnet_float
.magM(cyldir), magnet_fixed.dim(2), magnet_float.dim(1), magnet_fixed.dim(3), magnet_float
.dim(3), displ(cyldir)).';
761     forces22 = magdir*cylinder_force_coaxial(+magnet_fixed.magM(cyldir), +magnet_float
.magM(cyldir), magnet_fixed.dim(2), magnet_float.dim(2), magnet_fixed.dim(3), magnet_float
.dim(3), displ(cyldir)).';
762     forces_out = forces11 + forces12 + forces21 + forces22;
763     else
764         ecc_forces = cylinder_force_eccentric(magnet_fixed.dim, magnet_float.dim, displ(
cyldir), ecc, magnet_fixed.magM(cyldir), magnet_float.magM(cyldir)).';
765         forces_out(cyldir)= ecc_forces(2);
766         forces_out(cylnotdir(1))= displ(cylnotdir(1))/ecc*ecc_forces(1);
767         forces_out(cylnotdir(2))= displ(cylnotdir(2))/ecc*ecc_forces(1);
768 % Need to check this division into components is correct...
769     end
771 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.

```

783 function force_out = single_magnet_force(displ)
784
785     force_components = nan([9 3]);
786
787     d_x = rotate_x_to_z(displ);
788     d_y = rotate_y_to_z(displ);
789
790     force_components(1,:)= ...
791         rotate_z_to_x( cuboid_force_z_z(size1_x,size2_x,d_x,J1_x,J2_x));
792
793     force_components(2,:)= ...
794         rotate_z_to_x( cuboid_force_z_y(size1_x,size2_x,d_x,J1_x,J2_x));
795
796     force_components(3,:)= ...
797         rotate_z_to_x( cuboid_force_z_x(size1_x,size2_x,d_x,J1_x,J2_x));
798
799     force_components(4,:)= ...
800         rotate_z_to_y( cuboid_force_z_x(size1_y,size2_y,d_y,J1_y,J2_y));
801
802     force_components(5,:)= ...
803         rotate_z_to_y( cuboid_force_z_z(size1_y,size2_y,d_y,J1_y,J2_y));
804
805     force_components(6,:)= ...

```

```

806     rotate_z_to_y( cuboid_force_z_y(size1_y,size2_y,d_y,J1_y,J2_y));
808     force_components(9,:)= cuboid_force_z_z( size1,size2,displ,magnet_fixed.magM,magnet_float
.magM );
810     force_components(8,:)= cuboid_force_z_y( size1,size2,displ,magnet_fixed.magM,magnet_float
.magM );
812     force_components(7,:)= cuboid_force_z_x( size1,size2,displ,magnet_fixed.magM,magnet_float
.magM );

815     force_out = sum(force_components);
816     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.

```

828     function torques_out = single_magnet_torque(displ,lever)
830         torque_components = nan([size(displ)9]);
832         d_x = rotate_x_to_z(displ);
833         d_y = rotate_y_to_z(displ);
834         d_z = displ;
836         l_x = rotate_x_to_z(lever);
837         l_y = rotate_y_to_z(lever);
838         l_z = lever;
840         torque_components(:, :, 9)= cuboid_torque_z_z( size1,size2,d_z,l_z,magnet_fixed.magM
,magnet_float.magM );
842         torque_components(:, :, 8)= cuboid_torque_z_y( size1,size2,d_z,l_z,magnet_fixed.magM
,magnet_float.magM );
844         torque_components(:, :, 7)= torques_calc_z_x( size1,size2,d_z,l_z,magnet_fixed.magM
,magnet_float.magM );
846         torque_components(:, :, 1)= ...
847             rotate_z_to_x( cuboid_torque_z_z(size1_x,size2_x,d_x,l_x,J1_x,J2_x));
849         torque_components(:, :, 2)= ...
850             rotate_z_to_x( cuboid_torque_z_y(size1_x,size2_x,d_x,l_x,J1_x,J2_x));
852         torque_components(:, :, 3)= ...
853             rotate_z_to_x( torques_calc_z_x(size1_x,size2_x,d_x,l_x,J1_x,J2_x));
855         torque_components(:, :, 4)= ...
856             rotate_z_to_y( torques_calc_z_x(size1_y,size2_y,d_y,l_y,J1_y,J2_y));
858         torque_components(:, :, 5)= ...
859             rotate_z_to_y( cuboid_torque_z_z(size1_y,size2_y,d_y,l_y,J1_y,J2_y));
861         torque_components(:, :, 6)= ...
862             rotate_z_to_y( cuboid_torque_z_y(size1_y,size2_y,d_y,l_y,J1_y,J2_y));

```

```

864     torques_out = sum(torque_components,3);
865 end

```

5.5 The single_magnet_stiffness() function

```

872 function stiffness_out = single_magnet_stiffness(displ)
873
874     stiffness_components = nan([9 3]);
875
876     d_x = rotate_x_to_z(displ);
877     d_y = rotate_y_to_z(displ);
878
879     stiffness_components(7,:)= ...
880         stiffnesses_calc_z_x( size1,size2,displ,magnet_fixed.magM,magnet_float.magM );
881
882     stiffness_components(8,:)= ...
883         stiffnesses_calc_z_y( size1,size2,displ,magnet_fixed.magM,magnet_float.magM );
884
885     stiffness_components(9,:)= ...
886         stiffnesses_calc_z_z( size1,size2,displ,magnet_fixed.magM,magnet_float.magM );
887
888     stiffness_components(1,:)= ...
889         swap_x_z( stiffnesses_calc_z_z( size1_x,size2_x,d_x,J1_x,J2_x ));
890
891     stiffness_components(2,:)= ...
892         swap_x_z( stiffnesses_calc_z_y( size1_x,size2_x,d_x,J1_x,J2_x ));
893
894     stiffness_components(3,:)= ...
895         swap_x_z( stiffnesses_calc_z_x( size1_x,size2_x,d_x,J1_x,J2_x ));
896
897     stiffness_components(4,:)= ...
898         swap_y_z( stiffnesses_calc_z_x( size1_y,size2_y,d_y,J1_y,J2_y ));
899
900     stiffness_components(5,:)= ...
901         swap_y_z( stiffnesses_calc_z_z( size1_y,size2_y,d_y,J1_y,J2_y ));
902
903     stiffness_components(6,:)= ...
904         swap_y_z( stiffnesses_calc_z_y( size1_y,size2_y,d_y,J1_y,J2_y ));
905
906     stiffness_out = sum(stiffness_components);
907 end

```

5.6 The `stiffnesses_calc_z_z()` function

```
917 function calc_out = stiffnesses_calc_z_z(size1,size2,offset,J1,J2)
918
919     J1 = J1(3);
920     J2 = J2(3);
921
922     if (J1==0 || J2==0)
923         calc_out = [0; 0; 0];
924         return;
925     end
926
927     u = offset(1)+ size2(1)*(-1).^index_j - size1(1)*(-1).^index_i;
928     v = offset(2)+ size2(2)*(-1).^index_l - size1(2)*(-1).^index_k;
929     w = offset(3)+ size2(3)*(-1).^index_q - size1(3)*(-1).^index_p;
930     r = sqrt(u.^2+v.^2+w.^2);
931
932     component_x = - r - (u.^2 .*v)./(u.^2+w.^2)- v.*log(r-v);
933
934     component_y = - r - (v.^2 .*u)./(v.^2+w.^2)- u.*log(r-u);
935
936     component_z = - component_x - component_y;
937
938     component_x = index_sum.*component_x;
939     component_y = index_sum.*component_y;
940     component_z = index_sum.*component_z;
941
942     calc_out = J1*J2*magconst .* ...
943         [ sum(component_x(:));
944           sum(component_y(:));
945           sum(component_z(:))] ;
946
947 end
```

5.7 The `stiffnesses_calc_z_y()` function

```
954 function calc_out = stiffnesses_calc_z_y(size1,size2,offset,J1,J2)
955
956     J1 = J1(3);
957     J2 = J2(2);
958
959     if (J1==0 || J2==0)
960         calc_out = [0; 0; 0];
961         return;
962     end
963
964     u = offset(1)+ size2(1)*(-1).^index_j - size1(1)*(-1).^index_i;
965     v = offset(2)+ size2(2)*(-1).^index_l - size1(2)*(-1).^index_k;
966     w = offset(3)+ size2(3)*(-1).^index_q - size1(3)*(-1).^index_p;
967     r = sqrt(u.^2+v.^2+w.^2);
968
969     component_x = ((u.^2 .*v)./(u.^2 + v.^2))+ (u.^2 .*w)./(u.^2 + w.^2)...
970         - u.*atan1(v.*w,r.*u)+ multiply_x_log_y( w , r + v )+ ...
971         + multiply_x_log_y( v , r + w );
972     component_y = - v/2 + (u.^2 .*v)./(u.^2 + v.^2)- (u.*v.*w)./(v.^2 + w.^2)...
```

```

974     - u.*atan1(u.*w,r.*v)- multiply_x_log_y( v , r + w );
975 component_z = - component_x - component_y;

977 component_x = index_sum.*component_x;
978 component_y = index_sum.*component_y;
979 component_z = index_sum.*component_z;

981 calc_out = J1*J2*magconst .* ...
982     [ sum(component_x(:));
983       sum(component_y(:));
984       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.)

```

996 function out = multiply_x_log_y(x,y)
997     out = x.*log(y);
998     out(~isfinite(out))=0;
999 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.

```

1006 function out = atan1(x,y)
1007     out = zeros(size(x));
1008     ind = x~=0 & y~=0;
1009     out(ind)= atan(x(ind)./y(ind));
1010 end

1013 end

```

5.8 The stiffnesses_calc_z_x() function

```

1019 function calc_out = stiffnesses_calc_z_x(size1,size2,offset,J1,J2)

1021 stiffnesses_xyz = stiffnesses_calc_z_y(...
1022     swap_x_y(size1), swap_x_y(size2), rotate_x_to_y(offset),...
1023     J1, rotate_x_to_y(J2));

1025 calc_out = swap_x_y(stiffnesses_xyz);

1027 end

```

5.9 The `torques_calc_z_y()` function

```
1032 function calc_out = torques_calc_z_y(size1,size2,offset,lever,J1,J2)
1034     if J1(3)~=0 && J2(2)~=0
1035         error('Torques cannot be calculated for orthogonal magnets yet.')
1036     end
1038     calc_out = 0*offset;
1040 end
```

5.10 The `torques_calc_z_x()` function

```
1044 function calc_out = torques_calc_z_x(size1,size2,offset,lever,J1,J2)
1046     if J1(3)~=0 && J2(1)~=0
1047         error('Torques cannot be calculated for orthogonal magnets yet.')
1048     end
1050     calc_out = 0*offset;
1052 end
```

5.11 The `forces_magcyl_shell_calc()` function

```
1057 function Fz = forces_magcyl_shell_calc(magsize,coilsize,displ,Jmag,Nrz,I)
1059     Jcoil = 4*pi*1e-7*Nrz(2)*I/coil.dim(3);
1061     shell_forces = nan([length(displ)Nrz(1)]);
1063     for rr = 1:Nrz(1)
1065         this_radius = coilsize(1)+(rr-1)/(Nrz(1)-1)*(coilsize(2)-coilsize(1));
1066         shell_size = [this_radius, coilsize(3)];
1068         shell_forces(:,rr)= cylinder_force_coaxial(magsize,shell_size,displ,Jmag,Jcoil);
1070     end
1072     Fz = sum(shell_forces,2);
1074 end
1078 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 dipole_forcetorque() function

```
1087 function [f,t] = dipole_forcetorque(m_a,m_b,r_ab)
```

```
    [F,T] = CALCDIPOLEFORCETORQUE(MA,MB,R)
```

Calculates the force and torque on magnetic dipole MB due to magnetic dipole MA with distance vector RAB.

MA and MB must have equal size and be 3x1 or 3xM vector arrays. RAB

Magnetic dipole moments can be calculated for a hard magnet using the equation $m = 1/(4\pi \times 10^{-7}) \times B_r \times V$ where B_r is the remanence magnetisation vector in Tesla and V is the magnet volume.

For a coil, the magnetic dipole moment is $m = N \times I \times A \times n$ where I is the current, N the number of turns, A the cross-sectional area of the current loop(s), and n is the normal vector to the loop(s).

These equations have been adapted from the following pair of papers: * Yung 1998: <http://doi.org/10.1155/1998/79537>
* Landecker 1999: <http://doi.org/10.1155/1999/97902>

```
1110 assert( size(r_ab,1)==3 , "Displacement vector RAB must be 3xM size")
```

```
1111 assert( size(m_a,1)==3 && size(m_b,1)==3 , "Dipole moment vectors MA and MB must be 3  
xM size")
```

```
1112 assert( size(m_a,2)==size(m_b,2), "Dipole moment vectors MA and MB must be equal sizes  
")
```

```
1113 assert( size(m_a,2)==size(r_ab,2), "Dipole moment vectors MA and MB and displacement  
vector RAB must be equal sizes")
```

replicate dipole moment vectors to the same length as the displacement vector:

```
1116 N = size(r_ab,2);
```

```
1117 if size(m_a,2) == 1
```

```
1118     m_a = repmat(m_a,[1,N]);
```

```
1119     m_b = repmat(m_b,[1,N]);
```

```
1120 end
```

```
1122 R_ab = sqrt(r_ab(1,:).^2+r_ab(2,:).^2+r_ab(3,:).^2);
```

```
1123 rnorm = r_ab./R_ab;
```

```
1125 dot_rma = dot(rnorm,m_a);
```

```
1126 dot_rmb = dot(rnorm,m_b);
```

```
1128 f = 3e-7./R_ab.^4.*( ...  
1129     + rnorm.*dot(m_a,m_b) ...  
1130     + m_a.*dot_rmb ...  
1131     + m_b.*dot_rma ...  
1132     - 5*rnorm.*dot_rma.*dot_rmb ...  
1133 );
```

```
1135 t = 1e-7./R_ab.^3.*( ...  
1136     + cross(3*m_b,dot_rma.*rnorm) ...  
1137     - cross(m_a,m_b) ...  
1138 );
```

```
1140 end
```

6.2 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

```

1167 function forces_xyz = cuboid_force_z_z(size1,size2,offset,J1,J2)
1169 magconst = 1/(4*pi*(4*pi*1e-7));
1171 J1 = J1(3);
1172 J2 = J2(3);
1174 if ( abs(J1)<eps || abs(J2)<eps )
1175     forces_xyz = [0; 0; 0];
1176     return;
1177 end
1179 component_x = 0;
1180 component_y = 0;
1181 component_z = 0;
1183 for ii = [1 -1]
1184     for jj = [1 -1]
1185         for kk = [1 -1]
1186             for ll = [1 -1]
1187                 for pp = [1 -1]
1188                     for qq = [1 -1]
1190                         u = offset(1)+ size2(1)*jj - size1(1)*ii;
1191                         v = offset(2)+ size2(2)*ll - size1(2)*kk;
1192                         w = offset(3)+ size2(3)*qq - size1(3)*pp;
1193                         r = sqrt(u.^2+v.^2+w.^2);
1195                         if w == 0
1196                             atan_term = 0;
1197                         else
1198                             atan_term = atan(u.*v./(r.*w));
1199                         end
1200                         if abs(r-u)< eps
1201                             log_ru = 0;
1202                         else
1203                             log_ru = log(r-u);
1204                         end
1205                         if abs(r-v)< eps
1206                             log_rv = 0;
1207                         else
1208                             log_rv = log(r-v);
1209                         end
1211                     cx = ...

```



```

1212         + 0.5*(v.^2-w.^2).*log_ru ...
1213         + u.*v.*log_rv ...
1214         + v.*w.*atan_term...
1215         + 0.5*r.*u;

1217     cy = ...
1218         + 0.5*(u.^2-w.^2).*log_rv ...
1219         + u.*v.*log_ru ...
1220         + u.*w.*atan_term ...
1221         + 0.5*r.*v;

1223     cz = ...
1224         - u.*w.*log_ru ...
1225         - v.*w.*log_rv ...
1226         + u.*v.*atan_term ...
1227         - r.*w;

1229     ind_sum = ii*jj*kk*ll*pp*qq;
1230     component_x = component_x + ind_sum.*cx;
1231     component_y = component_y + ind_sum.*cy;
1232     component_z = component_z + ind_sum.*cz;

1234     end
1235   end
1236 end
1237 end
1238 end
1239 end

1241 forces_xyz = J1*J2*magconst.*[component_x; component_y; component_z];
1243 end

```

6.3 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

```

1272 function forces_xyz = cuboid_force_z_y(size1,size2,offset,J1,J2)
1273
1274     J1 = J1(3);
1275     J2 = J2(2);
1276
1277     if ( abs(J1)<eps || abs(J2)<eps )
1278         forces_xyz = [0; 0; 0];
1279     return;

```

```

1280     end

1282     component_x = 0;
1283     component_y = 0;
1284     component_z = 0;

1286     for ii = [1 -1]
1287         for jj = [1 -1]
1288             for kk = [1 -1]
1289                 for ll = [1 -1]
1290                     for pp = [1 -1]
1291                         for qq = [1 -1]

1293                             ind_sum = ii*jj*kk*ll*pp*qq;

1295                             u = offset(1)+ size2(1)*jj - size1(1)*ii;
1296                             v = offset(2)+ size2(2)*ll - size1(2)*kk;
1297                             w = offset(3)+ size2(3)*qq - size1(3)*pp;
1298                             r = sqrt(u.^2+v.^2+w.^2);

1300                             if u == 0
1301                                 atan_term_u = 0;
1302                             else
1303                                 atan_term_u = atan(v.*w./(r.*u));
1304                             end
1305                             if v == 0
1306                                 atan_term_v = 0;
1307                             else
1308                                 atan_term_v = atan(u.*w./(r.*v));
1309                             end
1310                             if w == 0
1311                                 atan_term_w = 0;
1312                             else
1313                                 atan_term_w = atan(u.*v./(r.*w));
1314                             end

1316                             if abs(r-u)< eps
1317                                 log_ru = 0;
1318                             else
1319                                 log_ru = log(r-u);
1320                             end
1321                             if abs(r+w)< eps
1322                                 log_rw = 0;
1323                             else
1324                                 log_rw = log(r+w);
1325                             end
1326                             if abs(r+v)< eps
1327                                 log_rv = 0;
1328                             else
1329                                 log_rv = log(r+v);
1330                             end

1332                             cx = ...
1333                                 + v.*w.*log_ru ...
1334                                 - v.*u.*log_rw ...

```

```

1335         - u.*w.*log_rv ...
1336         + 0.5*u.^2.*atan_term_u ...
1337         + 0.5*v.^2.*atan_term_v ...
1338         + 0.5*w.^2.*atan_term_w;

1340     cy = ...
1341         - 0.5*(u.^2-v.^2).*log_rw ...
1342         + u.*w.*log_ru ...
1343         + u.*v.*atan_term_v ...
1344         + 0.5*w.*r;

1346     cz = ...
1347         - 0.5*(u.^2-w.^2).*log_rv ...
1348         + u.*v.*log_ru ...
1349         + u.*w.*atan_term_w ...
1350         + 0.5*v.* r;

1352     component_x = component_x + ind_sum.*cx;
1353     component_y = component_y + ind_sum.*cy;
1354     component_z = component_z + ind_sum.*cz;

1356     end
1357   end
1358 end
1359   end
1360 end
1361 end

1363     forces_xyz = J1*J2/(4*pi*(4*pi*1e-7))*[ component_x; component_y; component_z ];
1365 end

```

6.4 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

```

1391 function forces_xyz = cuboid_force_z_x(size1,size2,offset,J1,J2)

1393 J1 = J1(3);
1394 J2 = J2(1);

1396 if ( abs(J1)<eps || abs(J2)<eps )
1397     forces_xyz = [0; 0; 0];
1398     return;
1399 end

1401 component_x = 0;

```

```

1402 component_y = 0;
1403 component_z = 0;

1405 for ii = [1 -1]
1406     for jj = [1 -1]
1407         for kk = [1 -1]
1408             for ll = [1 -1]
1409                 for pp = [1 -1]
1410                     for qq = [1 -1]

1412                         ind_sum = ii*jj*kk*ll*pp*qq;

1414                         u = -offset(2)- size2(2)*jj + size1(2)*ii;
1415                         v = offset(1)+ size2(1)*ll - size1(1)*kk;
1416                         w = offset(3)+ size2(3)*qq - size1(3)*pp;
1417                         r = sqrt(u.^2+v.^2+w.^2);

1419                         if u == 0
1420                             atan_term_u = 0;
1421                         else
1422                             atan_term_u = atan(v.*w./(r.*u));
1423                         end
1424                         if v == 0
1425                             atan_term_v = 0;
1426                         else
1427                             atan_term_v = atan(u.*w./(r.*v));
1428                         end
1429                         if w == 0
1430                             atan_term_w = 0;
1431                         else
1432                             atan_term_w = atan(u.*v./(r.*w));
1433                         end

1435                         if abs(r-u)< eps
1436                             log_ru = 0;
1437                         else
1438                             log_ru = log(r-u);
1439                         end
1440                         if abs(r+w)< eps
1441                             log_rw = 0;
1442                         else
1443                             log_rw = log(r+w);
1444                         end
1445                         if abs(r+v)< eps
1446                             log_rv = 0;
1447                         else
1448                             log_rv = log(r+v);
1449                         end

1451                         cx = ...
1452                             + v.*w.*log_ru ...
1453                             - v.*u.*log_rw ...
1454                             - u.*w.*log_rv ...
1455                             + 0.5*u.^2.*atan_term_u ...
1456                             + 0.5*v.^2.*atan_term_v ...

```

```

1457         + 0.5*w.^2.*atan_term_w;
1459     cy = ...
1460         - 0.5*(u.^2-v.^2).*log_rw ...
1461         + u.*w.*log_ru ...
1462         + u.*v.*atan_term_v ...
1463         + 0.5*w.*r;
1465     cz = ...
1466         - 0.5*(u.^2-w.^2).*log_rv ...
1467         + u.*v.*log_ru ...
1468         + u.*w.*atan_term_w ...
1469         + 0.5*v.* r;
1471     component_x = component_x + ind_sum.*cx;
1472     component_y = component_y + ind_sum.*cy;
1473     component_z = component_z + ind_sum.*cz;
1475     end
1476   end
1477 end
1478 end
1479 end
1480 end
1482 forces_xyz = J1*J2/(4*pi*(4*pi*1e-7))*[ component_y; -component_x; component_z ];
1484 end

```

6.5 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

```

1522 function torque_zz = cuboid_torque_z_z(size1,size2,offset,lever,J1,J2)
1524 br1 = J1(3);
1525 br2 = J2(3);
1527 if br1==0 || br2==0
1528     torque_zz = 0*offset;

```

```

1529     return
1530 end

1532 Txyz = zeros([3, size(offset,2)]);

1534 for ii=[0,1]
1535     for jj=[0,1]
1536         for kk=[0,1]
1537             for ll=[0,1]
1538                 for mm=[0,1]
1539                     for nn=[0,1]

1541                         Cu = (-1)^ii.*size1(1)- offset(1,:)- lever(1,:);
1542                         Cv = (-1)^kk.*size1(2)- offset(2,:)- lever(2,:);
1543                         Cw = (-1)^mm.*size1(3)- offset(3,:)- lever(3,:);

1545                         u = offset(1,:)- (-1)^ii.*size1(1)+ (-1)^jj.*size2(1);
1546                         v = offset(2,:)- (-1)^kk.*size1(2)+ (-1)^ll.*size2(2);
1547                         w = offset(3,:)- (-1)^mm.*size1(3)+ (-1)^nn.*size2(3);

1549                         Cuu = 2*Cv + u;
1550                         Cvv = 2*Cw + v;
1551                         Cww = 2*Cw + w;

1553                         u2 = u.^2;
1554                         v2 = v.^2;
1555                         w2 = w.^2;
1556                         s2 = u2+v2+w2;
1557                         s = sqrt(s2);

1559 % find indexes where cuboid faces align
1560     a = (u2<eps)& (v2<eps);
1561     b = (u2<eps)& (w2<eps);
1562     c = (v2<eps)& (w2<eps);

1564 % and all those that do not
1565     d = ~a & ~b & ~c;

1567     Ex = nan(1,size(offset,2));
1568     Ey = nan(1,size(offset,2));
1569     Ez = nan(1,size(offset,2));

1571     if any(a)
1572         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)));
1573         Ey(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)));
1574         Ez(a) = 1/4*(Cu(a)-Cv(a))*w2(a).*log(w2(a));
1575     end

1577     if any(b)
1578         Ex(b) = -1/4*Cw(b).*v(b).*(v(b)+2*abs(v(b)));
1579         Ey(b) = -1/4*Cw(b).*v2(b).*(log(v2(b))-1);
1580         Ez(b) = 1/72*v(b).*(2*v2(b)+36*Cv(b).*abs(v(b))+9*v(b).*Cvv(b).*log(v2(b)
)));
1581     end

1583     if any(c)

```

```

1584     Ex(c) = 1/4*Cw(c).*u2(c).*(log(u2(c))-1);
1585     Ey(c) = 1/4*Cw(c).*(u2(c)+2*abs(u(c)).*u(c));
1586     Ez(c) = -1/72*u(c).*(2*u2(c)+36*Cv(c).*abs(u(c))+9*u(c).*Cuu(c).*log(u2(c)
))) );
1587 end
1589 if any(d)
1591     Ex(d) = 1/8.*( ...
1592         - Cww(d).*s2(d)+ ...
1593         - 2.*s(d).*(v(d).*Cww(d)+2.*Cvv(d).*w(d))+ ...
1594         ...
1595         + 2.*Cww(d).*(s2(d)-v2(d)).*log(s(d)+v(d))+ ...
1596         - 8.*u(d).*v(d).*(Cw(d)+w(d)).*log(s(d)-u(d))+ ...
1597         ...
1598         + 8.*Cv(d).*u(d).*w(d).*acoth(s(d)./u(d))+ ...
1599         + 4.*w(d).*(v(d).*Cvv(d)-w(d).*Cww(d)).*acoth(s(d)./v(d))+ ...
1600         + 4.*u(d).*(v(d).*Cvv(d)-w(d).*Cww(d)).*atan(u(d).*v(d)./(w(d).*s(d)))+
...
1601         0);
1603     Ey(d) = 1/8*( ...
1604         + Cww(d).*s2(d)+ ...
1605         + 2.*s(d).*(u(d).*Cww(d)+2.*Cuu(d).*w(d)).* ...
1606         ...
1607         - 2.*Cww(d).*(s2(d)-u2(d)).*log(s(d)+u(d))+ ...
1608         + 8.*u(d).*v(d).*(Cw(d)+w(d)).*(log(s(d)-v(d))-1)+ ...
1609         ...
1610         - 8.*Cu(d).*v(d).*w(d).*acoth(s(d)./v(d))+ ...
1611         - 4.*w(d).*(u(d).*Cuu(d)-w(d).*Cww(d)).*acoth(s(d)./u(d))+ ...
1612         - 4.*v(d).*(u(d).*Cuu(d)-w(d).*Cww(d)).*atan(u(d).*v(d)./(w(d).*s(d)))+
...
1613         ...
1614         + 8.*v(d).*w(d).*(Cw(d)+w(d)).*atan(u(d)./w(d))+ ...
1615         0);
1617     Ez(d) = 1/36.*( ...
1618         - u(d).^3 + ...
1619         + v(d).^3 + ...
1620         + 6.*w2(d).*(v(d)-u(d))+ ...
1621         + 18.*s(d).*(Cu(d).*v(d)-u(d).*Cv(d))+ ...
1622         ...
1623         + 18.*u(d).*v(d).*( Cuu(d).*(log(s(d)-u(d))-1)- Cvv(d).*(log(s(d)-v(d))
-1))+ ...
1624         + 9.*Cvv(d).*(v2(d)-w2(d)).*log(s(d)+u(d))+ ...
1625         - 9.*Cuu(d).*(u2(d)-w2(d)).*log(s(d)+v(d)) ...
1626         ...
1627         + 6.*w(d).*(w2(d)-3.*v(d).*Cvv(d)).*atan(u(d)./w(d))+ ...
1628         - 6.*w(d).*(w2(d)-3.*u(d).*Cuu(d)).*atan(v(d)./w(d))+ ...
1629         - 18.*w(d).*(Cvv(d).*v(d)-u(d).*Cuu(d)).*atan(u(d).*v(d)./(w(d).*s(d)))
+ ...
1630         0);
1632 end

```

```

1634         Txyz = Txyz + (-1)^(ii+jj+kk+ll+mm+nn)*[Ex; Ey; Ez];
1635     end
1636 end
1637 end
1638 end
1639 end
1640 end
1641 end
1642
1643 torque_zz = Txyz.*br1*br2/(16*pi^2*1e-7);
1644
1645 end

```

```

1654 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
1676 bzr1 = J1(3);
1677 byr2 = 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

```

1681 if abs(bzr1)< eps || abs(byr2)< eps
1682     torque_zy = zeros(size(offset));
1683     return
1684 end

```

```

    preallocate sums as rows of zero with the same length as the offset array
1687 Tx = zeros([1, size(offset,2)]);
1688 Ty = Tx;
1689 Tz = Tx;

```

```

    calculate sums as described in Janssen, 2011
1692 for ii = 0:1
1693     for jj = 0:1
1694         for kk = 0:1
1695             for ll = 0:1
1696                 for mm = 0:1
1697                     for nn = 0:1
1698
1699                         Ex = nan(size(Tx));
1700                         Ey = Ex;
1701                         Ez = Ex;
1702
1703                         Cu = ((-1)^ii).*(size1(1))-offset(1,:)-lever(1,:);

```



```

1704     Cv = ((-1)^kk).*(size1(2))-offset(2,:)-lever(2,:);
1705     Cw = ((-1)^mm).*(size1(3))-offset(3,:)-lever(3,:);

1707     u = offset(1,:)-((-1)^ii).*(size1(1))+((-1)^jj).*(size2(1));
1708     v = offset(2,:)-((-1)^kk).*(size1(2))+((-1)^ll).*(size2(2));
1709     w = offset(3,:)-((-1)^mm).*(size1(3))+((-1)^nn).*(size2(3));

1711     u2 = u.^2;
1712     v2 = v.^2;
1713     w2 = w.^2;
1714     r2 = u2+v2+w2;
1715     r = sqrt(r2);

1717 % find indexes where cuboid magnets align
1718     a = (u2<eps)& (v2<eps);
1719     b = (u2<eps)& (w2<eps);
1720     c = (v2<eps)& (w2<eps);

1722 % find indexes where cuboid magnets do not align
1723     d = ~a & ~b & ~c;

1725 % if magnets are aligned in any two directions, use the following limit
1726 % expressions to calculate the sums

1728     if any(a)
1729         Ex(a) = -1/24.*w2(a).*(...
1730             + 6.*Cw(a).*(1+2.*sign(w(a)))+ ...
1731             + 8.*abs(w(a))...
1732             );
1733         Ey(a) = pi/4.*w2(a).*sign(w(a)).*Cw(a);
1734         Ez(a) = 1/36.*w2(a).*(...
1735             + w(a) ...
1736             - 1.5.*w(a).*log(w2(a))...
1737             + 9.*(2.*Cu(a)-pi*Cv(a)).*sign(w(a))...
1738             );
1739     end

1741     if any(b)
1742         Ex(b) = 1/12.*v2(b).*(...
1743             + 3.*Cw(b).*(log(v2(b))-1)+ ...
1744             + 2.*sign(v(b)).*(3.*Cv(b)+v(b))...
1745             );
1746         Janssen (wrong?) Ey(b) = 1/24.*v2(b).*(... + v(b).*( log(v2(b)) - 3 ) ... - 12.*Cu(b).*sign(v(b))
... );
1751         Ey(b) = 1/24*(...
1752             + 3*v(b).^3 ...
1753             - r(b).*v(b).*( 12*Cu(b)+10*u(b)+3*r(b))...
1754             + 2*v(b).*log(r(b)+u(b)).*( 3.*u(b).*(2*Cu(b)+u(b))+ v2(b)- 3*w(b).*(4.*
Cw(b)+w(b)))...
1755             + 2*v(b).*log(r(b)-u(b)).*( -3.*u(b).*(2*Cu(b)+u(b))+ v2(b))...
1756             );
1758         Ez(b) = 1/4.*Cu(b).*v2(b).*log(v2(b));
1759     end

1761     if any(c)

```

```

Janssen (wrong?) Ex(c) = 1/12.*u2(c).*(... + 2.*u(c).*(sign(u(c))-1) ... + 3.*Cw(c).*(log(u2(c))-
1) ... );
1767 Ex(c) = ...
1768 + 1/12.*r(c).*( 2*r2(c)- 3.*Cw(c).*r(c)+ 6.*Cv(c).*v(c)- 6.*Cw(c).*w(c)
- 6.*w2(c))...
1769 + 1/2.*Cw(c).*(u2(c).*log(r(c)+w(c))+ v2(c).*log(r(c)-w(c)))...
1770 ;
1771 Ey(c) = 1/36.*u(c).^3.*(3*log(u2(c))- 5);
1772 Ez(c) = -1/12.*( 3*Cu(c)+ 2*u(c)).*u2(c).*log(u2(c));
1773 end
1775 % if magnets are not aligned in two directions, use the following
1776 % expressions to calculate the sums
1778 if any(d)
1779 Ex(d) = (...
1780 +1/12*r(d).*( 2*r2(d)+ 6.*Cv(d).*v(d)- 6.*w2(d)- 6*Cw(d).*w(d)- 3*Cw(
d).*r(d))...
1781 ...
1782 -1/2*log(r(d)-u(d)).*u(d).*(2.*w(d).*Cw(d)+r2(d)-u2(d))...
1783 +1/2*log(r(d)-w(d)).*Cw(d).*(r2(d)-w2(d))...
1784 ...
1785 - acoth(r(d)./u(d)).*u(d).*v(d).*(Cv(d)+v(d))...
1786 -1/2*acoth(r(d)./v(d)).*(Cv(d)+v(d)).*(u2(d)-w2(d))...
1787 + acoth(r(d)./w(d)).*Cw(d).*u2(d)...
1788 ...
1789 +u(d).*w(d).*Cv(d).*atan(u(d).*v(d)./(w(d).*r(d)))...
1790 -u(d).*v(d).*Cw(d).*atan(u(d).*w(d)./(v(d).*r(d)))...
1791 ...
1792 +u(d).*v(d).*w(d).*atan(u(d).*v(d)./(w(d).*r(d)))...
1793 -u(d).*v(d).*Cw(d).*atan(w(d)./r(d))...
1794 );
1795 Ey(d) = 1/72*(...
1796 - 10*u(d).^3 ...
1797 + 9*v(d).^3 ...
1798 ... - 6*u(d).*w(d).*(18*Cw(d)+7*w(d))...
1799 - 3*r(d).*v(d).*( 12*Cu(d)+10*u(d)+3*r(d))...
1800 ...
1801 - 72*u(d).*v(d).*Cw(d).*log(r(d)+w(d))...
1802 ... - 72*u(d).*v(d).*Cw(d)...
1803 ...
1804 + 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)))...
1805 + 6*v(d).*log(r(d)-u(d)).*( -3.*u(d).*(2*Cu(d)+u(d))+ v2(d))...
1806 ...
1807 + 6*log(r(d)+v(d)).*( 2.*u(d).^3 + 3*Cu(d).*(u2(d)-w2(d)
))...
1808 + 18*log(r(d)-v(d)).*( 2.*u(d).*w(d).*(2*Cw(d)+w(d))- Cu(d).*(u2(d)-w2(
d)))...
1809 ...
1810 + 24*w2(d).*(3.*Cw(d)+w(d)).*atan(u(d)./w(d))...
1811 + 36*Cw(d).*(u2(d)+w2(d)).*atan(w(d)./u(d))...
1812 + 3*v(d).^3.*Cw(d).*atan(u(d).*w(d)./(v(d).*r(d)))...

```

```

1813         + 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)))...
1814         + 36*u2(d).*Cw(d).*atan((v(d).*w(d))./(u(d).*r(d)))...
1815         );
1816 Ez(d) = 1/12.*(...
1817         + 1/3*w(d).^3 ...
1818         + 2.*w(d).*v2(d)...
1819         + r(d).*w(d).*(6.*Cu(d)+u(d))...
1820         + 6.*u(d).*w(d).*(2.*Cu(d)+u(d))...
1821         + 6.*u(d).*w(d).*(2.*Cu(d)+u(d)).*log(r(d)-u(d))...
1822         - w(d).*( 3.*v2(d)+ w2(d)).*log(r(d)+u(d))...
1823         - 2.*( 2.*u(d).^3 + 3.*Cu(d).*(u2(d)-v2(d))).*log(r(d)+w(d))...
1824         - 12.*(Cv(d)+v(d)).*(...
1825         - u(d).*v(d).*log(r(d)+w(d))...
1826         + u(d).*w(d).*log(r(d)-v(d))...
1827         - v(d).*w(d).*log(r(d)+u(d))...
1828         )...
1829         - 2.*v(d).*(v2(d)-3.*u(d).*(2.*Cu(d)+u(d))).*atan(w(d)./r(d))...
1830         + 2.*v(d).*(v2(d)+3.*u(d).*(2.*Cu(d)+u(d))).*atan((u(d).*w(d))./(v(d)
.*r(d)))...
1831         - 6.*(Cv(d)+v(d)).*(...
1832         + u2(d).*( atan(w(d)./u(d))+ atan((v(d).*w(d))./(u(d).*r(d)))...
1833         + v2(d).*atan((u(d).*w(d))./(v(d).*r(d)))...
1834         + w2(d).*( ...
1835         + 2.*atan(u(d)./w(d))...
1836         + atan(w(d)./u(d))...
1837         + atan((u(d).*v(d))./(w(d).*r(d)))...
1838         ) ...
1839         )...
1840         );
1841     end
1842
1843     ind_sum = (-1)^(ii+jj+kk+ll+mm+nn);
1844     Tx = Tx + ind_sum.*Ex;
1845     Ty = Ty + ind_sum.*Ey;
1846     Tz = Tz + ind_sum.*Ez;
1847
1848     end
1849     end
1850     end
1851     end
1852     end
1853     end
1854
1855     torque_zy = bzf1*byr2/(16*pi*pi*1e-7).*[Tx; Ty; Tz];
1856
1857     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 $0 \leq m \leq 1$.

```

2090 function [K,E,PI] = ellipkepi(a,m)

2092 a1 = 1;
2093 g1 = sqrt(1-m);
2094 p1 = sqrt(1-a);
2095 q1 = 1;
2096 w1 = 1;

2098 nn = 0;
2099 qq = 1;
2100 ww = m;

2102 while max(abs(w1(:)))> eps || max(abs(q1(:)))> eps

2104 % Update from previous loop
2105     a0 = a1;
2106     g0 = g1;
2107     p0 = p1;
2108     q0 = q1;

2110 % for Elliptic I
2111     a1 = (a0+g0)/2;
2112     g1 = sqrt(a0.*g0);

2114 % for Elliptic II
2115     nn = nn + 1;
2116     d1 = (a0-g0)/2;
2117     w1 = 2^nn*d1.^2;
2118     ww = ww + w1;

2120 % for Elliptic III
2121     rr = p0.^2+a0.*g0;
2122     p1 = rr./p0/2;
2123     q1 = q0.*(p0.^2-a0.*g0)./rr/2;
2124     qq = qq + q1;

2126 end

2128 K = 1./a1*pi/2;
2129 E = K.*(1-ww/2);
2130 PI = K.*(1+a./(2-2*a).*qq);

2132 im = find(m == 1);
2133 if ~isempty(im)
2134     K(im) = inf;
2135     E(im) = ones(length(im),1);
2136     PI(im) = inf;
2137 end

2139 end

```

8 Magnet arrays

8.1 The multipoleforces() function

```
2151 function [varargout] = multipoleforces(fixed_array, float_array, displ, varargin)
2153 debug_disp = @(str)disp([]);
2154 calc_force_bool = false;
2155 calc_stiffness_bool = false;
2156 calc_torque_bool = false;
2158 for ii = 1:length(varargin)
2159     switch varargin{ii}
2160         case 'debug',    debug_disp = @(str)disp(str);
2161         case 'force',    calc_force_bool = true;
2162         case 'stiffness', calc_stiffness_bool = true;
2163         case 'torque',    calc_torque_bool = true;
2164         otherwise
2165             error(['Unknown calculation option ''',varargin{ii},'''])
2166         end
2167     end
2169 if ~calc_force_bool && ~calc_stiffness_bool && ~calc_torque_bool
2170     varargin{end+1} = 'force';
2171     calc_force_bool = true;
2172 end
2175 if size(displ,1)== 3
2176     % all good
2177 elseif size(displ,2)== 3
2178     displ = transpose(displ);
2179 else
2180     error(['Displacements matrix should be of size (3, D)',...
2181         'where D is the number of displacements.'])
2182 end
2184 Ndispl = size(displ,2);
2186 if calc_force_bool
2187     forces_out = nan([3 Ndispl]);
2188 end
2190 if calc_stiffness_bool
2191     stiffnesses_out = nan([3 Ndispl]);
2192 end
2194 if calc_torque_bool
2195     torques_out = nan([3 Ndispl]);
2196 end
2199 part = @(x,y)x(y);
2201 fixed_array = complete_array_from_input(fixed_array);
2202 float_array = complete_array_from_input(float_array);
```

```

2204 if calc_force_bool
2205     array_forces = nan([3 Ndispl fixed_array.total float_array.total]);
2206 end

2208 if calc_stiffness_bool
2209     array_stiffnesses = nan([3 Ndispl fixed_array.total float_array.total]);
2210 end

2212 displ_from_array_corners = displ ...
2213 + repmat(fixed_array.size/2,[1 Ndispl])...
2214 - repmat(float_array.size/2,[1 Ndispl]);

2217 for ii = 1:fixed_array.total

2219     fixed_magnet = magnetdefine(...
2220         'type', 'cuboid',...
2221         'dim',   fixed_array.dim(ii,:), ...
2222         'magn',  fixed_array.magn(ii), ...
2223         'magdir', fixed_array.magdir(ii,:)...
2224     );

2226     for jj = 1:float_array.total

2228         float_magnet = magnetdefine(...
2229             'type', 'cuboid',...
2230             'dim',   float_array.dim(jj,:), ...
2231             'magn',  float_array.magn(jj), ...
2232             'magdir', float_array.magdir(jj,:)...
2233         );

2235         mag_displ = displ_from_array_corners ...
2236             - repmat(fixed_array.magloc(ii,:),[1 Ndispl])...
2237             + repmat(float_array.magloc(jj,:),[1 Ndispl]);

2239         if calc_force_bool && ~calc_stiffness_bool
2240             array_forces(:, :, ii, jj) = ...
2241                 magnetforces(fixed_magnet, float_magnet, mag_displ, varargin{:});
2242         elseif calc_stiffness_bool && ~calc_force_bool
2243             array_stiffnesses(:, :, ii, jj) = ...
2244                 magnetforces(fixed_magnet, float_magnet, mag_displ, varargin{:});
2245         else
2246             [array_forces(:, :, ii, jj) array_stiffnesses(:, :, ii, jj)] = ...
2247                 magnetforces(fixed_magnet, float_magnet, mag_displ, varargin{:});
2248         end

2250     end
2251 end

2253 if calc_force_bool
2254     forces_out = sum(sum(array_forces,4),3);
2255 end

2257 if calc_stiffness_bool
2258     stiffnesses_out = sum(sum(array_stiffnesses,4),3);
2259 end

2262 varargout = {};

```

```

2264 for ii = 1:length(varargin)
2265     switch varargin{ii}
2266         case 'force'
2267             varargout{end+1} = forces_out;
2268
2269         case 'stiffness'
2270             varargout{end+1} = stiffnesses_out;
2271
2272         case 'torque'
2273             varargout{end+1} = torques_out;
2274     end
2275 end

```



```

2281 function array = complete_array_from_input(array)
2282
2283 if ~isfield(array,'type')
2284     array.type = 'generic';
2285 end

```



```

2288 if ~isfield(array,'face')
2289     array.face = 'undefined';
2290 end

```



```

2292 linear_index = 0;
2293 planar_index = [0 0];

```



```

2295 switch array.type
2296     case 'generic'
2297     case 'linear',          linear_index = 1;
2298     case 'linear-quasi',    linear_index = 1;
2299     case 'planar',          planar_index = [1 2];
2300     case 'quasi-halbach',   planar_index = [1 2];
2301     case 'patchwork',       planar_index = [1 2];
2302     otherwise
2303         error(['Unknown array type ''',array.type,','.'])
2304     end

```



```

2306 if ~isequal(array.type,'generic')
2307     if linear_index == 1
2308         if ~isfield(array,'align')
2309             array.align = 'x';
2310         end
2311         switch array.align
2312             case 'x', linear_index = 1;
2313             case 'y', linear_index = 2;
2314             case 'z', linear_index = 3;
2315             otherwise
2316                 error('Alignment for linear array must be ''x'', ''y'', or ''z''.')
2317             end
2318     else
2319         if ~isfield(array,'align')
2320             array.align = 'xy';
2321         end
2322         switch array.align

```

```

2323     case 'xy', planar_index = [1 2];
2324     case 'yz', planar_index = [2 3];
2325     case 'xz', planar_index = [1 3];
2326     otherwise
2327         error('Alignment for planar array must be 'xy'', 'yz'', or 'xz''.')
2328     end
2329 end
2330 end

2332 switch array.face
2333     case {'+x', '-x'}, facing_index = 1;
2334     case {'+y', '-y'}, facing_index = 2;
2335     case {'up', 'down'}, facing_index = 3;
2336     case {'+z', '-z'}, facing_index = 3;
2337     case 'undefined', facing_index = 0;
2338 end

2340 if linear_index ~= 0
2341     if linear_index == facing_index
2342         error('Arrays cannot face into their alignment direction.')
2343     end
2344 elseif ~isequal( planar_index, [0 0] )
2345     if any( planar_index == facing_index )
2346         error('Planar-type arrays can only face into their orthogonal direction')
2347     end
2348 end

2351 switch array.type
2352     case 'linear'

2354 array = extrapolate_variables(array);

2356 array.mcount = ones(1,3);
2357 array.mcount(linear_index)= array.Nmag;

2359     case 'linear-quasi'

2362 if isfield(array, 'ratio') && isfield(array, 'mlength')
2363     error('Cannot specify both 'ratio' and 'mlength''.')
2364 elseif ~isfield(array, 'ratio') && ~isfield(array, 'mlength')
2365     error('Must specify either 'ratio' or 'mlength''.')
2366 end

2369 array.Nmag_per_wave = 4;
2370 array.magdir_rotate = 90;

2372 if isfield(array, 'Nwaves')
2373     array.Nmag = array.Nmag_per_wave*array.Nwaves+1;
2374 else
2375     error(''Nwaves' must be specified.')
2376 end

2378 if isfield(array, 'mlength')
2379     if numel(array.mlength) ~= 2
2380         error(''mlength' must have length two for linear-quasi arrays.')
2381     end

```



```

2382     array.ratio = array.mlength(2)/array.mlength(1);
2383 else
2384     if isfield(array,'length')
2385         array.mlength(1)= 2*array.length/(array.Nmag*(1+array.ratio)+1-array.ratio);
2386         array.mlength(2)= array.mlength(1)*array.ratio;
2387     else
2388         error('''length''must be specified.')
2389     end
2390 end

2392 array.mcount = ones(1,3);
2393 array.mcount(linear_index)= array.Nmag;

2395 array.msize = nan([array.mcount 3]);

2397 [sindex_x sindex_y sindex_z] = ...
2398     meshgrid(1:array.mcount(1), 1:array.mcount(2), 1:array.mcount(3));

2402 all_indices = [1 1 1];
2403 all_indices(linear_index)= 0;
2404 all_indices(facing_index)= 0;
2405 width_index = find(all_indices);

2407 for ii = 1:array.Nmag
2408     array.msize(sindex_x(ii),sindex_y(ii),sindex_z(ii),linear_index)= ...
2409         array.mlength(mod(ii-1,2)+1);
2410     array.msize(sindex_x(ii),sindex_y(ii),sindex_z(ii),facing_index)= ...
2411         array.height;
2412     array.msize(sindex_x(ii),sindex_y(ii),sindex_z(ii),width_index)= ...
2413         array.width;
2414 end

2417     case 'planar'

2419     if isfield(array,'length')
2420         if length(array.length)== 1
2421             if isfield(array,'width')
2422                 array.length = [ array.length array.width ];
2423             else
2424                 array.length = [ array.length array.length ];
2425             end
2426         end
2427     end

2429     if isfield(array,'mlength')
2430         if length(array.mlength)== 1
2431             if isfield(array,mwidth)
2432                 array.mlength = [ array.mlength array.mwidth ];
2433             else
2434                 array.mlength = [ array.mlength array.mlength ];
2435             end
2436         end
2437     end

2439     var_names = {'length','mlength','wavelength','Nwaves',...

```

```

2440         'Nmag', 'Nmag_per_wave', 'magdir_rotate'};

2442 tmp_array1 = struct();
2443 tmp_array2 = struct();
2444 var_index = zeros(size(var_names));

2446 for iii = 1:length(var_names)
2447     if isfield(array, var_names{iii})
2448         tmp_array1.(var_names{iii})= array.(var_names{iii})(1);
2449         tmp_array2.(var_names{iii})= array.(var_names{iii})(end);
2450     else
2451         var_index(iii)= 1;
2452     end
2453 end

2455 tmp_array1 = extrapolate_variables(tmp_array1);
2456 tmp_array2 = extrapolate_variables(tmp_array2);

2458 for iii = find(var_index)
2459     array.(var_names{iii})= [tmp_array1.(var_names{iii})tmp_array2.(var_names{iii})];
2460 end

2462 array.width = array.length(2);
2463 array.length = array.length(1);

2465 array.mwidth = array.mlength(2);
2466 array.mlength = array.mlength(1);

2468 array.mcount = ones(1,3);
2469 array.mcount(planar_index)= array.Nmag;

2471     case 'quasi-halbach'

2473 if isfield(array, 'mcount')
2474     if numel(array.mcount)~=3
2475         error('''mcount'' must always have three elements.')
2476     end
2477 elseif isfield(array, 'Nwaves')
2478     if numel(array.Nwaves)> 2
2479         error('''Nwaves'' must have one or two elements only.')
2480     end
2481     array.mcount(facing_index)= 1;
2482     array.mcount(planar_index)= 4*array.Nwaves+1;
2483 elseif isfield(array, 'Nmag')
2484     if numel(array.Nmag)> 2
2485         error('''Nmag'' must have one or two elements only.')
2486     end
2487     array.mcount(facing_index)= 1;
2488     array.mcount(planar_index)= array.Nmag;
2489 else
2490     error('Must specify the number of magnets (''mcount'' or ''Nmag'') or wavelengths (''Nwaves'')')
2491 end

2493     case 'patchwork'

2495 if isfield(array, 'mcount')
2496     if numel(array.mcount)~=3

```

```

2497     error('''mcount''must always have three elements.')
2498 end
2499 elseif isfield(array,'Nmag')
2500     if numel(array.Nmag)> 2
2501         error('''Nmag''must have one or two elements only.')
2502     end
2503     array.mcount(facing_index)= 1;
2504     array.mcount(planar_index)= array.Nmag;
2505 else
2506     error('Must specify the number of magnets (''mcount''or ''Nmag'')')
2507 end
2509 end

2512 array.total = prod(array.mcount);

2514 if ~isfield(array,'msize')
2515     array.msize = [NaN NaN NaN];
2516     if linear_index ~=0
2517         array.msize(linear_index)= array.mlength;
2518         array.msize(facing_index)= array.height;
2519         array.msize(isnan(array.msize))= array.width;
2520     elseif ~isequal( planar_index, [0 0] )
2521         array.msize(planar_index)= [array.mlength array.mwidth];
2522         array.msize(facing_index)= array.height;
2523     else
2524         error('The array property ''msize''is not defined and I have no way to infer it.'
2525     )
2526 end
2527 elseif numel(array.msize)== 1
2528     array.msize = repmat(array.msize,[3 1]);
2529 end

2530 if numel(array.msize)== 3
2531     array.msize_array = ...
2532     repmat(reshape(array.msize,[1 1 1 3]), array.mcount);
2533 else
2534     if isequal([array.mcount 3],size(array.msize))
2535         array.msize_array = array.msize;
2536     else
2537         error('Magnet size ''msize''must have three elements (or one element for a cube magnet
2538     ).')
2539 end
2540 array.dim = reshape(array.msize_array, [array.total 3]);

2542 if ~isfield(array,'mgap')
2543     array.mgap = [0; 0; 0];
2544 elseif length(array.mgap)== 1
2545     array.mgap = repmat(array.mgap,[3 1]);
2546 end

2550 if ~isfield(array,'magn')

```

```

2551     if isfield(array,'grade')
2552         array.magn = grade2magn(array.grade);
2553     else
2554         array.magn = 1;
2555     end
2556 end

2558 if length(array.magn)== 1
2559     array.magn = repmat(array.magn,[array.total 1]);
2560 else
2561     error('Magnetisation magnitude ''magn''must be a single value.')
2562 end

2566 if ~isfield(array,'magdir_fn')

2568     if ~isfield(array,'face')
2569         array.face = '+z';
2570     end

2572     switch array.face
2573     case {'up','+z','+y','+x'}, magdir_rotate_sign = 1;
2574     case {'down','-z','-y','-x'}, magdir_rotate_sign = -1;
2575     end

2577     if ~isfield(array,'magdir_first')
2578         array.magdir_first = magdir_rotate_sign*90;
2579     end

2581     magdir_fn_comp{1} = @(ii,jj,kk)0;
2582     magdir_fn_comp{2} = @(ii,jj,kk)0;
2583     magdir_fn_comp{3} = @(ii,jj,kk)0;

2585     switch array.type
2586     case 'linear'
2587         magdir_theta = @(nn)...
2588             array.magdir_first+magdir_rotate_sign*array.magdir_rotate*(nn-1);

2590         magdir_fn_comp{linear_index} = @(ii,jj,kk)...
2591             cosd(magdir_theta(part([ii,jj,kk],linear_index)));

2593         magdir_fn_comp{facing_index} = @(ii,jj,kk)...
2594             sind(magdir_theta(part([ii,jj,kk],linear_index)));

2596     case 'linear-quasi'

2598         magdir_theta = @(nn)...
2599             array.magdir_first+magdir_rotate_sign*90*(nn-1);

2601         magdir_fn_comp{linear_index} = @(ii,jj,kk)...
2602             cosd(magdir_theta(part([ii,jj,kk],linear_index)));

2604         magdir_fn_comp{facing_index} = @(ii,jj,kk)...
2605             sind(magdir_theta(part([ii,jj,kk],linear_index)));

2607     case 'planar'

2609         magdir_theta = @(nn)...
2610             array.magdir_first(1)+magdir_rotate_sign*array.magdir_rotate(1)*(nn-1);

```

```

2612     magdir_phi = @(nn)...
2613         array.magdir_first(end)+magdir_rotate_sign*array.magdir_rotate(end)*(nn-1);

2615     magdir_fn_comp{planar_index(1)} = @(ii,jj,kk)...
2616         cosd(magdir_theta(part([ii,jj,kk],planar_index(2)))));

2618     magdir_fn_comp{planar_index(2)} = @(ii,jj,kk)...
2619         cosd(magdir_phi(part([ii,jj,kk],planar_index(1)))));

2621     magdir_fn_comp{facing_index} = @(ii,jj,kk)...
2622         sind(magdir_theta(part([ii,jj,kk],planar_index(1))))...
2623         + sind(magdir_phi(part([ii,jj,kk],planar_index(2)))));

2625     case 'patchwork'

2627         magdir_fn_comp{planar_index(1)} = @(ii,jj,kk)0;
2629         magdir_fn_comp{planar_index(2)} = @(ii,jj,kk)0;

2631         magdir_fn_comp{facing_index} = @(ii,jj,kk)...
2632             magdir_rotate_sign*(-1)^( ...
2633                 part([ii,jj,kk],planar_index(1))...
2634                 + part([ii,jj,kk],planar_index(2))...
2635                 + 1 ...
2636             );

2638     case 'quasi-halbach'

2640         magdir_fn_comp{planar_index(1)} = @(ii,jj,kk)...
2641             sind(90*part([ii,jj,kk],planar_index(1)))...
2642             * cosd(90*part([ii,jj,kk],planar_index(2)));

2644         magdir_fn_comp{planar_index(2)} = @(ii,jj,kk)...
2645             cosd(90*part([ii,jj,kk],planar_index(1)))...
2646             * sind(90*part([ii,jj,kk],planar_index(2)));

2648         magdir_fn_comp{facing_index} = @(ii,jj,kk)...
2649             magdir_rotate_sign ...
2650             * sind(90*part([ii,jj,kk],planar_index(1)))...
2651             * sind(90*part([ii,jj,kk],planar_index(2)));

2653     otherwise
2654         error('Array property ''magdir_fn''not defined and I have no way to infer it.')
2655     end

2657     array.magdir_fn = @(ii,jj,kk)...
2658         [ magdir_fn_comp{1}(ii,jj,kk)...
2659           magdir_fn_comp{2}(ii,jj,kk)...
2660           magdir_fn_comp{3}(ii,jj,kk)];

2662 end

2668 array.magloc = nan([array.total 3]);
2669 array.magdir = array.magloc;
2670 arrat.magloc_array = nan([array.mcount(1)array.mcount(2)array.mcount(3)3]);

2672 nn = 0;
2673 for iii = 1:array.mcount(1)

```

```

2674     for jjj = 1:array.mcount(2)
2675         for kkk = 1:array.mcount(3)
2676             nn = nn + 1;
2677             array.magdir(nn,:)= array.magdir_fn(iii,jjj,kkk);
2678         end
2679     end
2680 end

2682 magsep_x = zeros(size(array.mcount(1)));
2683 magsep_y = zeros(size(array.mcount(2)));
2684 magsep_z = zeros(size(array.mcount(3)));

2686 magsep_x(1)= array.msize_array(1,1,1,1)/2;
2687 magsep_y(1)= array.msize_array(1,1,1,2)/2;
2688 magsep_z(1)= array.msize_array(1,1,1,3)/2;

2690 for iii = 2:array.mcount(1)
2691     magsep_x(iii)= array.msize_array(iii-1,1,1,1)/2 ...
2692         + array.msize_array(iii ,1,1,1)/2 ;
2693 end
2694 for jjj = 2:array.mcount(2)
2695     magsep_y(jjj)= array.msize_array(1,jjj-1,1,2)/2 ...
2696         + array.msize_array(1,jjj ,1,2)/2 ;
2697 end
2698 for kkk = 2:array.mcount(3)
2699     magsep_z(kkk)= array.msize_array(1,1,kkk-1,3)/2 ...
2700         + array.msize_array(1,1,kkk ,3)/2 ;
2701 end

2703 magloc_x = cumsum(magsep_x);
2704 magloc_y = cumsum(magsep_y);
2705 magloc_z = cumsum(magsep_z);

2707 for iii = 1:array.mcount(1)
2708     for jjj = 1:array.mcount(2)
2709         for kkk = 1:array.mcount(3)
2710             array.magloc_array(iii,jjj,kkk,:)= ...
2711                 [magloc_x(iii); magloc_y(jjj); magloc_z(kkk)] ...
2712                 + [iii-1; jjj-1; kkk-1].*array.mgap;
2713         end
2714     end
2715 end
2716 array.magloc = reshape(array.magloc_array,[array.total 3]);

2718 array.size = squeeze( array.magloc_array(end,end,end,:)...
2719     - array.magloc_array(1,1,1,:)...
2720     + array.msize_array(1,1,1,:)/2 ...
2721     + array.msize_array(end,end,end,:)/2 );

2723 debug_disp('Magnetisation directions')
2724 debug_disp(array.magdir)

2726 debug_disp('Magnet locations:')
2727 debug_disp(array.magloc)

2730 end

```

```

2734 function array_out = extrapolate_variables(array)
2736 var_names = {'wavelength','length','Nwaves','mlength',...
2737             'Nmag','Nmag_per_wave','magdir_rotate'};
2739 if isfield(array,'Nwaves')
2740     mcount_extra = 1;
2741 else
2742     mcount_extra = 0;
2743 end
2745 if isfield(array,'mlength')
2746     mlength_adjust = false;
2747 else
2748     mlength_adjust = true;
2749 end
2751 variables = nan([7 1]);
2753 for iii = 1:length(var_names);
2754     if isfield(array,var_names(iii))
2755         variables(iii)= array.(var_names{iii});
2756     end
2757 end
2759 var_matrix = ...
2760     [1, 0, 0, -1, 0, -1, 0;
2761      0, 1, 0, -1, -1, 0, 0;
2762      0, 0, 1, 0, -1, 1, 0;
2763      0, 0, 0, 0, 0, 1, 1];
2765 var_results = [0 0 0 log(360)]';
2766 variables = log(variables);
2768 idx = ~isnan(variables);
2769 var_known = var_matrix(:,idx)*variables(idx);
2770 var_calc = var_matrix(:,~idx)\(var_results-var_known);
2771 variables(~idx)= var_calc;
2772 variables = exp(variables);
2774 for iii = 1:length(var_names);
2775     array.(var_names{iii})= variables(iii);
2776 end
2778 array.Nmag = round(array.Nmag)+ mcount_extra;
2779 array.Nmag_per_wave = round(array.Nmag_per_wave);
2781 if mlength_adjust
2782     array.mlength = array.mlength * (array.Nmag-mcount_extra)/array.Nmag;
2783 end
2785 array_out = array;
2787 end
2791 end

```