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

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 coordidates.

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.)

[1] Try 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 force[2] 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.

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

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

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

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

with something similar for `magnet_float`. The displacement matrix is then built up as a list of (3×1) 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 escuse 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 $\pm 90°$ 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 essense, 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

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

if nargin == 1
  mag = varargin{1};
else
  mag = struct(varargin{:});
end

if ~isfield(mag,'type')
  warning('Magnets should always define their "type". E.g., {''type'',''cuboid''} for a cuboid magnet.')
  if length(mag.dim)== 2
    mag.type = 'cylinder';
  else
    mag.type = 'cuboid';
  end
end

if isfield(mag,'grade')
  if isfield(mag,'magn')
    error('Cannot specify both ''magn''and ''grade''.')
  else
    mag.magn = grade2magn(mag.grade);
  end
end


if ~isfield(mag,'lever')
  mag.lever = [0; 0; 0];
else
  ss = size(mag.lever);
  if (ss(1)~=3)&& (ss(2)==3)
    mag.lever = mag.lever.'; % attempt [3 M] shape
  end
end


if strcmp(mag.type,'cylinder')

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

```
      mag.dir = mag.magdir;
    end
  else
    if ~isfield(mag,'magdir')
      mag.magdir = mag.dir;
    end
  end

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

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

else

  if ~isfield(mag,'magdir')
    warning('Magnet direction ("magdir")not specified; assuming +z.')
    mag.magdir = [0; 0; 1];
  else
    mag.magdir = resolve_magdir(mag.magdir);
  end

end

mag.fndefined = true;

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 $[BH]_{\max} = 42\,\mathrm{MG\,Oe}$. Since $1\,\mathrm{MG\,Oe} = 100/(4\pi)\,\mathrm{kJ/m^3}$, the calculation simplifies to

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

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

```
function magn = grade2magn(grade)

if isnumeric(grade)
  magn = 2*sqrt(grade/100);
else
  if strcmp(grade(1),'N')
    grade = grade(2:end);
  end
```

```
  magn = 2*sqrt(str2double(grade)/100);
end

end
```

4.1.2 The `resolve_magdir()` function

```
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 pi rather than symbolically).

```
  if numel(magdir)== 2
    theta = magdir(1);
    phi = magdir(2);
    magdir = [ cosd(phi)*cosd(theta); cosd(phi)*sind(theta); sind(phi)];
  elseif numel(magdir)== 3
    if all(magdir == 0)
      magdir = [0; 0; 0]; % this looks redundant but ensures column vector
    else
      magdir = magdir(:)/norm(magdir);
    end
  elseif numel(magdir)== 1
    switch magdir
      case 'x'; magdir = [1;0;0];
      case 'y'; magdir = [0;1;0];
      case 'z'; magdir = [0;0;1];
      case '+x'; magdir = [1;0;0];
      case '+y'; magdir = [0;1;0];
      case '+z'; magdir = [0;0;1];
      case '-x'; magdir = [-1; 0; 0];
      case '-y'; magdir = [ 0;-1; 0];
      case '-z'; magdir = [ 0; 0;-1];
      otherwise, error('Magnetisation % s not understood.',magdir);
    end
  else
    error('magdir has wrong number of elements.')
  end

end
```

5 The magnetforces() function

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

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

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

```
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`.

```
debug_disp = @(str)disp([]);
calc_force_bool    = false;
calc_stiffness_bool = false;
calc_torque_bool   = false;
```

```
for iii = 1:length(varargin)
  switch varargin{iii}
    case 'debug',     debug_disp = @(str)disp(str);
    case 'force',     calc_force_bool    = true;
    case 'stiffness', calc_stiffness_bool = true;
    case 'torque',    calc_torque_bool   = true;
    case 'x', warning("Options 'x','y','z'are no longer supported.");
    case 'y', warning("Options 'x','y','z'are no longer supported.");
    case 'z', warning("Options 'x','y','z'are no longer supported.");
    otherwise
      error(['Unknown calculation option ''',varargin{iii},''''])
  end
end
```

```
if ~calc_force_bool && ~calc_stiffness_bool && ~calc_torque_bool
  varargin{end+1} = 'force';
  calc_force_bool = true;
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 me.

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

```
Ndispl = size(displ,2);
```

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

```
if calc_stiffness_bool
  stiffnesses_out = nan([3 Ndispl]);
end

if calc_torque_bool
  torques_out = nan([3 Ndispl]);
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 ($-\pi/2 \le \phi \le \pi/2$) and `theta` is the angle around the horizontal plane ($0 \le \theta \le 2\pi$). This follows Matlab's definition; other conventions are commonly used as well. Remember:

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

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 coordindates.

```
if ~isfield(magnet_fixed,'fndefined')
  magnet_fixed = magnetdefine(magnet_fixed);
end
if ~isfield(magnet_float,'fndefined')
  magnet_float = magnetdefine(magnet_float);
end


if strcmp(magnet_fixed.type, 'coil')

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

  coil = magnet_fixed;
  magnet = magnet_float;
  magtype = 'coil';
  coil_sign = +1;

end

if strcmp(magnet_float.type, 'coil')

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

  coil = magnet_float;
  magnet = magnet_fixed;
  magtype = 'coil';
  coil_sign = -1;
```

```
end

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

if strcmp(magtype,'cuboid')

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

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

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

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

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

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

  size1_x = swap_x_z(size1);
  size2_x = swap_x_z(size2);
  J1_x    = rotate_x_to_z(J1);
  J2_x    = rotate_x_to_z(J2);

  size1_y = swap_y_z(size1);
  size2_y = swap_y_z(size2);
  J1_y    = rotate_y_to_z(J1);
  J2_y    = rotate_y_to_z(J2);

elseif strcmp(magtype,'cylinder')

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

  if any(abs(magnet_fixed.dir)~= abs(magnet_float.dir))
    error('Cylindrical magnets must be oriented in the same direction')
  end
  if any(abs(magnet_fixed.magdir)~= abs(magnet_float.magdir))
    error('Cylindrical magnets must be oriented in the same direction')
  end
  if any(abs(magnet_fixed.dir)~= abs(magnet_fixed.magdir))
    error('Cylindrical magnets must be magnetised in the same direction as their orientation')
  end
  if any(abs(magnet_float.dir)~= abs(magnet_float.magdir))
    error('Cylindrical magnets must be magnetised in the same direction as their orientation')
  end
```

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

  magnet_float.magdir = magnet_float.magdir(:);
  magnet_fixed.magdir = magnet_fixed.magdir(:);
  magnet_float.dir = magnet_float.dir(:);
  magnet_fixed.dir = magnet_fixed.dir(:);

  J1 = magnet_fixed.magn*magnet_fixed.magdir;
  J2 = magnet_float.magn*magnet_float.magdir;
  debug_disp('Magnetisation vectors:')
  debug_disp(J1)
  debug_disp(J2)

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.

```
if strcmp(magtype,'coil')

  for iii = 1:Ndispl
    forces_out(:,iii)= coil_sign*coil.dir*...
      forces_magcyl_shell_calc(...
        magnet.dim, ...
        coil.dim, ...
        squeeze(displ(cyldir,:)), ...
        J1(cyldir), ...
        coil.current, ...
        coil.turns);
  end

elseif strcmp(magtype,'cuboid')

  if calc_force_bool
    for iii = 1:Ndispl
      forces_out(:,iii)= single_magnet_force(displ(:,iii));
    end
  end

  if calc_stiffness_bool
    for iii = 1:Ndispl
      stiffnesses_out(:,iii)= single_magnet_stiffness(displ(:,iii));
    end
  end

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

elseif strcmp(magtype,'cylinder')

  if calc_force_bool
    for iii = 1:Ndispl
```

```
      forces_out(:,iii)=  single_magnet_cyl_force(displ(:,iii));
    end
  end

  if calc_stiffness_bool
    error('Stiffness cannot be calculated for cylindrical magnets yet.')
  end

  if calc_torque_bool
    error('Torques cannot be calculated for cylindrical magnets yet.')
  end

end
```

After all of the calculations have occured, 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.

```
argcount = 0;

for iii = 1:length(varargin)
  switch varargin{iii}
    case 'force',    argcount = argcount+1;
    case 'stiffness', argcount = argcount+1;
    case 'torque',   argcount = argcount+1;
  end
end

varargout = cell(argcount,1);

argcount = 0;

for iii = 1:length(varargin)
  switch varargin{iii}
    case 'force',     argcount = argcount+1; varargout{argcount} = forces_out;
    case 'stiffness', argcount = argcount+1; varargout{argcount} = stiffnesses_out;
    case 'torque',    argcount = argcount+1; varargout{argcount} = torques_out;
  end
end
```

That is the end of the main function.

5.1 The `single_magnet_cyl_force()` function

```
  function forces_out = single_magnet_cyl_force(displ)

    forces_out = nan(size(displ));

    ecc = sqrt(sum(displ(cylnotdir).^2));

    if ecc < eps
      debug_disp('Coaxial')
      magdir = [0;0;0];
      magdir(cyldir)= 1;
```

```
      forces_out = magdir*cylinder_force_coaxial(J1(cyldir), J2(cyldir), size1(1), size2(1), size1(2), size2(2), displ(cyldir)).';
    else
      debug_disp('Non-coaxial')
      ecc_forces = cylinder_force_eccentric(size1, size2, displ(cyldir), ecc, J1(cyldir), J2(cyldir)).';
      forces_out(cyldir)= ecc_forces(2);
      forces_out(cylnotdir(1))= displ(cylnotdir(1))/ecc*ecc_forces(1);
      forces_out(cylnotdir(2))= displ(cylnotdir(2))/ecc*ecc_forces(1);
% Need to check this division into components is correct...
    end

  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.

```
  function force_out = single_magnet_force(displ)

    force_components = nan([9 3]);

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

    debug_disp(' ')
    debug_disp('CALCULATING THINGS')
    debug_disp('==================')
    debug_disp('Displacement:')
    debug_disp(displ')
    debug_disp('Magnetisations:')
    debug_disp(J1')
    debug_disp(J2')

    debug_disp('Forces x-x:')
    force_components(1,:)= ...
      rotate_z_to_x( cuboid_force_z_z(size1_x,size2_x,d_x,J1_x,J2_x));

    debug_disp('Forces x-y:')
    force_components(2,:)= ...
      rotate_z_to_x( cuboid_force_z_y(size1_x,size2_x,d_x,J1_x,J2_x));

    debug_disp('Forces x-z:')
    force_components(3,:)= ...
      rotate_z_to_x( cuboid_force_z_x(size1_x,size2_x,d_x,J1_x,J2_x));

    debug_disp('Forces y-x:')
    force_components(4,:)= ...
      rotate_z_to_y( cuboid_force_z_x(size1_y,size2_y,d_y,J1_y,J2_y));

    debug_disp('Forces y-y:')
    force_components(5,:)= ...
```

```
    rotate_z_to_y( cuboid_force_z_z(size1_y,size2_y,d_y,J1_y,J2_y));

  debug_disp('Forces y-z:')
  force_components(6,:)= ...
    rotate_z_to_y( cuboid_force_z_y(size1_y,size2_y,d_y,J1_y,J2_y));

  debug_disp('z-z force:')
  force_components(9,:)= cuboid_force_z_z( size1,size2,displ,J1,J2 );

  debug_disp('z-y force:')
  force_components(8,:)= cuboid_force_z_y( size1,size2,displ,J1,J2 );

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

  force_out = sum(force_components);
end
```

5.3 The single_magnet_torque() function

```
function torques_out = single_magnet_torque(displ,lever)

  torque_components = nan([size(displ)9]);

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

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

  debug_disp(' ')
  debug_disp('CALCULATING THINGS')
  debug_disp('==================')
  debug_disp('Displacement:')
  debug_disp(displ')
  debug_disp('Magnetisations:')
  debug_disp(J1')
  debug_disp(J2')

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

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

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

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

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

```
    rotate_z_to_x( torques_calc_z_y(size1_x,size2_x,d_x,l_x,J1_x,J2_x));

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

  debug_disp('Torques y-x:')
  torque_components(:,:,4)= ...
    rotate_z_to_y( torques_calc_z_x(size1_y,size2_y,d_y,l_y,J1_y,J2_y));

  debug_disp('Torques y-y:')
  torque_components(:,:,5)= ...
    rotate_z_to_y( cuboid_torque_z_z(size1_y,size2_y,d_y,l_y,J1_y,J2_y));

  debug_disp('Torques y-z:')
  torque_components(:,:,6)= ...
    rotate_z_to_y( torques_calc_z_y(size1_y,size2_y,d_y,l_y,J1_y,J2_y));

  torques_out = sum(torque_components,3);
end


function stiffness_out = single_magnet_stiffness(displ)

  stiffness_components = nan([9 3]);

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

  debug_disp(' ')
  debug_disp('CALCULATING THINGS')
  debug_disp('==================')
  debug_disp('Displacement:')
  debug_disp(displ')
  debug_disp('Magnetisations:')
  debug_disp(J1')
  debug_disp(J2')

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

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

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

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

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

```
  debug_disp('x-z stiffness:')
  stiffness_components(3,:)= ...
    swap_x_z( stiffnesses_calc_z_x( size1_x,size2_x,d_x,J1_x,J2_x ));

  debug_disp('y-x stiffness:')
  stiffness_components(4,:)= ...
    swap_y_z( stiffnesses_calc_z_x( size1_y,size2_y,d_y,J1_y,J2_y ));

  debug_disp('y-y stiffness:')
  stiffness_components(5,:)= ...
    swap_y_z( stiffnesses_calc_z_z( size1_y,size2_y,d_y,J1_y,J2_y ));

  debug_disp('y-z stiffness:')
  stiffness_components(6,:)= ...
    swap_y_z( stiffnesses_calc_z_y( size1_y,size2_y,d_y,J1_y,J2_y ));

  stiffness_out = sum(stiffness_components);
end
```

5.4 The `stiffnesses_calc_z_z()` function

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

  J1 = J1(3);
  J2 = J2(3);

  if (J1==0 || J2==0)
    debug_disp('Zero magnetisation.')
    calc_out =  [0; 0; 0];
    return;
  end

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

  component_x = - r - (u.^2 .*v)./(u.^2+w.^2)- v.*log(r-v);

  component_y = - r - (v.^2 .*u)./(v.^2+w.^2)- u.*log(r-u);

  component_z = - component_x - component_y;

  component_x = index_sum.*component_x;
  component_y = index_sum.*component_y;
  component_z = index_sum.*component_z;

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

  debug_disp(calc_out')

end
```

5.5 The `stiffnesses_calc_z_y()` function

```
function calc_out = stiffnesses_calc_z_y(size1,size2,offset,J1,J2)
  J1 = J1(3);
  J2 = J2(2);

  if (J1==0 || J2==0)
    debug_disp('Zero magnetisation.')
    calc_out =  [0; 0; 0];
    return;
  end
  u = offset(1)+ size2(1)*(-1).^index_j - size1(1)*(-1).^index_i;
  v = offset(2)+ size2(2)*(-1).^index_l - size1(2)*(-1).^index_k;
  w = offset(3)+ size2(3)*(-1).^index_q - size1(3)*(-1).^index_p;
  r = sqrt(u.^2+v.^2+w.^2);
  component_x = ((u.^2 .*v)./(u.^2 + v.^2))+ (u.^2 .*w)./(u.^2 + w.^2)...
    - u.*atan1(v.*w,r.*u)+ multiply_x_log_y( w , r + v )+ ...
    + multiply_x_log_y( v , r + w );
  component_y = - v/2 + (u.^2 .*v)./(u.^2 + v.^2)- (u.*v.*w)./(v.^2 + w.^2)...
    -  u.*atan1(u.*w,r.*v)- multiply_x_log_y( v , r + w );
  component_z = - component_x - component_y;
  component_x = index_sum.*component_x;
  component_y = index_sum.*component_y;
  component_z = index_sum.*component_z;
  calc_out = J1*J2*magconst .* ...
    [ sum(component_x(:));
    sum(component_y(:));
    sum(component_z(:))] ;
  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.)

```
  function out = multiply_x_log_y(x,y)
    out = x.*log(y);
    out(~isfinite(out))=0;
  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`.

```
    function out = atan1(x,y)
      out = zeros(size(x));
      ind = x~=0 & y~=0;
      out(ind)= atan(x(ind)./y(ind));
    end

  end
```

5.6 The stiffnesses_calc_z_x() function

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

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

    calc_out = swap_x_y(stiffnesses_xyz);

  end
```

5.7 The torques_calc_z_y() function

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

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

    calc_out = 0*offset;

  end
```

5.8 The torques_calc_z_x() function

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

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

    calc_out = 0*offset;

  end
```

5.9 The `forces_magcyl_shell_calc()` function

```
  function Fz = forces_magcyl_shell_calc(magsize,coilsize,displ,Jmag,Nrz,I)

    Jcoil = 4*pi*1e-7*Nrz(2)*I/coil.dim(3);

    shell_forces = nan([length(displ)Nrz(1)]);

    for rr = 1:Nrz(1)

      this_radius = coilsize(1)+(rr-1)/(Nrz(1)-1)*(coilsize(2)-coilsize(1));
      shell_size = [this_radius, coilsize(3)];

      shell_forces(:,rr)= cylinder_force_coaxial(magsize,shell_size,displ,Jmag,Jcoil);


    end

    Fz = sum(shell_forces,2);

  end


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

```
function forces_xyz = cuboid_force_z_z(size1,size2,offset,J1,J2)

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

J1 = J1(3);
J2 = J2(3);

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

component_x = 0;
component_y = 0;
```

```
component_z = 0;

for ii = [1 -1]
  for jj = [1 -1]
    for kk = [1 -1]
      for ll = [1 -1]
        for pp = [1 -1]
          for qq = [1 -1]

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

            if w == 0
              atan_term = 0;
            else
              atan_term = atan(u.*v./(r.*w));
            end
            if abs(r-u)< eps
              log_ru = 0;
            else
              log_ru = log(r-u);
            end
            if abs(r-v)< eps
              log_rv = 0;
            else
              log_rv = log(r-v);
            end

            cx = ...
              + 0.5*(v.^2-w.^2).*log_ru ...
              + u.*v.*log_rv ...
              + v.*w.*atan_term...
              + 0.5*r.*u;

            cy = ...
              + 0.5*(u.^2-w.^2).*log_rv ...
              + u.*v.*log_ru ...
              + u.*w.*atan_term ...
              + 0.5*r.*v;

            cz = ...
              - u.*w.*log_ru ...
              - v.*w.*log_rv ...
              + u.*v.*atan_term ...
              - r.*w;

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

          end
        end
      end
```

```
    end
  end
end

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

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

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

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

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

  component_x = 0;
  component_y = 0;
  component_z = 0;

  for ii = [1 -1]
    for jj = [1 -1]
      for kk = [1 -1]
        for ll = [1 -1]
          for pp = [1 -1]
            for qq = [1 -1]

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

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

              if u == 0
                atan_term_u = 0;
              else
                atan_term_u = atan(v.*w./(r.*u));
              end
```

```
        if v == 0
          atan_term_v = 0;
        else
          atan_term_v = atan(u.*w./(r.*v));
        end
        if w == 0
          atan_term_w = 0;
        else
          atan_term_w = atan(u.*v./(r.*w));
        end

        if abs(r-u)< eps
          log_ru = 0;
        else
          log_ru = log(r-u);
        end
        if abs(r+w)< eps
          log_rw = 0;
        else
          log_rw = log(r+w);
        end
        if abs(r+v)< eps
          log_rv = 0;
        else
          log_rv = log(r+v);
        end

        cx = ...
          + v.*w.*log_ru ...
          - v.*u.*log_rw ...
          - u.*w.*log_rv ...
          + 0.5*u.^2.*atan_term_u ...
          + 0.5*v.^2.*atan_term_v ...
          + 0.5*w.^2.*atan_term_w;

        cy = ...
          - 0.5*(u.^2-v.^2).*log_rw ...
          + u.*w.*log_ru ...
          + u.*v.*atan_term_v ...
          + 0.5*w.*r;

        cz = ...
          - 0.5*(u.^2-w.^2).*log_rv ...
          + u.*v.*log_ru ...
          + u.*w.*atan_term_w ...
          + 0.5*v.* r;

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

      end
    end
  end
end
```

```
      end
    end

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

  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

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

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

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

component_x = 0;
component_y = 0;
component_z = 0;

for ii = [1 -1]
  for jj = [1 -1]
    for kk = [1 -1]
      for ll = [1 -1]
        for pp = [1 -1]
          for qq = [1 -1]

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

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

            if u == 0
              atan_term_u = 0;
            else
              atan_term_u = atan(v.*w./(r.*u));
            end
            if v == 0
              atan_term_v = 0;
            else
```

```
          atan_term_v = atan(u.*w./(r.*v));
        end
        if w == 0
          atan_term_w = 0;
        else
          atan_term_w = atan(u.*v./(r.*w));
        end

        if abs(r-u)< eps
          log_ru = 0;
        else
          log_ru = log(r-u);
        end
        if abs(r+w)< eps
          log_rw = 0;
        else
          log_rw = log(r+w);
        end
        if abs(r+v)< eps
          log_rv = 0;
        else
          log_rv = log(r+v);
        end

        cx = ...
          + v.*w.*log_ru ...
          - v.*u.*log_rw ...
          - u.*w.*log_rv ...
          + 0.5*u.^2.*atan_term_u ...
          + 0.5*v.^2.*atan_term_v ...
          + 0.5*w.^2.*atan_term_w;

        cy = ...
          - 0.5*(u.^2-v.^2).*log_rw ...
          + u.*w.*log_ru ...
          + u.*v.*atan_term_v ...
          + 0.5*w.*r;

        cz = ...
          - 0.5*(u.^2-w.^2).*log_rv ...
          + u.*v.*log_ru ...
          + u.*w.*atan_term_w ...
          + 0.5*v.* r;

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

      end
    end
  end
 end
 end
end
```

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

end
```

7 Mathematical functions

7.1 The ellipkepi() function

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

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

a1 = 1;
g1 = sqrt(1-m);
p1 = sqrt(1-a);
q1 = 1;
w1 = 1;

nn = 0;
qq = 1;
ww = m;

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

% Update from previous loop
  a0 = a1;
  g0 = g1;
  p0 = p1;
  q0 = q1;

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

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

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

end

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

im = find(m == 1);
if ~isempty(im)
```

```
  K(im) = inf;
  E(im) = ones(length(im),1);
  PI(im) = inf;
end

end
```

8 Magnet arrays

8.1 The multipoleforces() function

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

debug_disp = @(str)disp([]);
calc_force_bool = false;
calc_stiffness_bool = false;
calc_torque_bool = false;

for ii = 1:length(varargin)
  switch varargin{ii}
    case 'debug',     debug_disp = @(str)disp(str);
    case 'force',     calc_force_bool    = true;
    case 'stiffness', calc_stiffness_bool = true;
    case 'torque',    calc_torque_bool   = true;
    otherwise
      error(['Unknown calculation option ''',varargin{ii},''''])
  end
end

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


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

Ndispl = size(displ,2);

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

if calc_stiffness_bool
  stiffnesses_out = nan([3 Ndispl]);
end
```

```
if calc_torque_bool
  torques_out = nan([3 Ndispl]);
end

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

fixed_array = complete_array_from_input(fixed_array);
float_array = complete_array_from_input(float_array);

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

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

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

for ii = 1:fixed_array.total

  fixed_magnet = magnetdefine(...
        'type',  'cuboid',...
        'dim',   fixed_array.dim(ii,:), ...
        'magn',  fixed_array.magn(ii), ...
        'magdir', fixed_array.magdir(ii,:)...
  );

  for jj = 1:float_array.total

    float_magnet = magnetdefine(...
      'type',  'cuboid',...
      'dim',   float_array.dim(jj,:), ...
      'magn',  float_array.magn(jj), ...
      'magdir', float_array.magdir(jj,:)...
    );

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

    if calc_force_bool && ~calc_stiffness_bool
      array_forces(:,:,ii,jj)= ...
          magnetforces(fixed_magnet, float_magnet, mag_displ,varargin{:});
    elseif calc_stiffness_bool && ~calc_force_bool
      array_stiffnesses(:,:,ii,jj)= ...
          magnetforces(fixed_magnet, float_magnet, mag_displ,varargin{:});
    else
      [array_forces(:,:,ii,jj)array_stiffnesses(:,:,ii,jj)] = ...
          magnetforces(fixed_magnet, float_magnet, mag_displ,varargin{:});
    end

  end
end

if calc_force_bool
```

```
  forces_out = sum(sum(array_forces,4),3);
end

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


varargout = {};

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

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

    case 'torque'
      varargout{end+1} = torques_out;
  end
end




function array = complete_array_from_input(array)

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


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

linear_index = 0;
planar_index = [0 0];

switch array.type
  case 'generic'
  case 'linear',          linear_index = 1;
  case 'linear-quasi',      linear_index = 1;
  case 'planar',          planar_index = [1 2];
  case 'quasi-halbach',   planar_index = [1 2];
  case 'patchwork',       planar_index = [1 2];
  otherwise
    error(['Unknown array type ''',array.type,'''.'])
end

if ~isequal(array.type,'generic')
  if linear_index == 1
    if ~isfield(array,'align')
      array.align = 'x';
    end
    switch array.align
      case 'x', linear_index = 1;
      case 'y', linear_index = 2;
      case 'z', linear_index = 3;
```

```
      otherwise
        error('Alignment for linear array must be ''x'', ''y'', or ''z''.')
      end
    else
      if ~isfield(array,'align')
        array.align = 'xy';
      end
      switch array.align
        case 'xy', planar_index = [1 2];
        case 'yz', planar_index = [2 3];
        case 'xz', planar_index = [1 3];
      otherwise
        error('Alignment for planar array must be ''xy'', ''yz'', or ''xz''.')
      end
    end
  end

  switch array.face
    case {'+x','-x'}, facing_index = 1;
    case {'+y','-y'}, facing_index = 2;
    case {'up','down'}, facing_index = 3;
    case {'+z','-z'}, facing_index = 3;
    case 'undefined', facing_index = 0;
  end

  if linear_index ~= 0
    if linear_index == facing_index
      error('Arrays cannot face into their alignment direction.')
    end
  elseif ~isequal( planar_index, [0 0] )
    if any( planar_index == facing_index )
      error('Planar-type arrays can only face into their orthogonal direction')
    end
  end


  switch array.type
    case 'linear'

  array = extrapolate_variables(array);

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

    case 'linear-quasi'


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


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

  if isfield(array,'Nwaves')
```

```
  array.Nmag = array.Nmag_per_wave*array.Nwaves+1;
else
  error('''Nwaves''must be specified.')
end

if isfield(array,'mlength')
  if numel(array.mlength)~=2
    error('''mlength''must have length two for linear-quasi arrays.')
  end
  array.ratio = array.mlength(2)/array.mlength(1);
else
  if isfield(array,'length')
    array.mlength(1)= 2*array.length/(array.Nmag*(1+array.ratio)+1-array.ratio);
    array.mlength(2)= array.mlength(1)*array.ratio;
  else
    error('''length''must be specified.')
  end
end

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

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

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



all_indices = [1 1 1];
all_indices(linear_index)= 0;
all_indices(facing_index)= 0;
width_index = find(all_indices);

for ii = 1:array.Nmag
  array.msize(sindex_x(ii),sindex_y(ii),sindex_z(ii),linear_index)= ...
    array.mlength(mod(ii-1,2)+1);
  array.msize(sindex_x(ii),sindex_y(ii),sindex_z(ii),facing_index)= ...
    array.height;
  array.msize(sindex_x(ii),sindex_y(ii),sindex_z(ii),width_index)= ...
    array.width;
end


  case 'planar'

if isfield(array,'length')
  if length(array.length)== 1
    if isfield(array,'width')
      array.length = [ array.length array.width ];
    else
      array.length = [ array.length array.length ];
    end
  end
end

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

```
    if isfield(array.mwidth)
      array.mlength = [ array.mlength array.mwidth ];
    else
      array.mlength = [ array.mlength array.mlength ];
    end
  end
end

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

tmp_array1 = struct();
tmp_array2 = struct();
var_index = zeros(size(var_names));

for iii = 1:length(var_names)
  if isfield(array,var_names(iii))
    tmp_array1.(var_names{iii})= array.(var_names{iii})(1);
    tmp_array2.(var_names{iii})= array.(var_names{iii})(end);
  else
    var_index(iii)= 1;
  end
end

tmp_array1 = extrapolate_variables(tmp_array1);
tmp_array2 = extrapolate_variables(tmp_array2);

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

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

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

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

  case 'quasi-halbach'

if isfield(array,'mcount')
  if numel(array.mcount)~=3
    error('''mcount''must always have three elements.')
  end
elseif isfield(array,'Nwaves')
  if numel(array.Nwaves)> 2
    error('''Nwaves''must have one or two elements only.')
  end
  array.mcount(facing_index)= 1;
  array.mcount(planar_index)= 4*array.Nwaves+1;
elseif isfield(array,'Nmag')
  if numel(array.Nmag)> 2
    error('''Nmag''must have one or two elements only.')
  end
  array.mcount(facing_index)= 1;
```

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

  case 'patchwork'

if isfield(array,'mcount')
  if numel(array.mcount)~=3
    error('''mcount''must always have three elements.')
  end
elseif isfield(array,'Nmag')
  if numel(array.Nmag)> 2
    error('''Nmag''must have one or two elements only.')
  end
  array.mcount(facing_index)= 1;
  array.mcount(planar_index)= array.Nmag;
else
  error('Must specify the number of magnets (''mcount''or ''Nmag'')')
end

end


array.total = prod(array.mcount);

if ~isfield(array,'msize')
  array.msize = [NaN NaN NaN];
  if linear_index ~=0
    array.msize(linear_index)= array.mlength;
    array.msize(facing_index)= array.height;
    array.msize(isnan(array.msize))= array.width;
  elseif ~isequal( planar_index, [0 0] )
    array.msize(planar_index)= [array.mlength array.mwidth];
    array.msize(facing_index)= array.height;
  else
    error('The array property ''msize''is not defined and I have no way to infer it.')
  end
elseif numel(array.msize)== 1
  array.msize = repmat(array.msize,[3 1]);
end

if numel(array.msize)== 3
  array.msize_array = ...
    repmat(reshape(array.msize,[1 1 1 3]), array.mcount);
else
  if isequal([array.mcount 3],size(array.msize))
    array.msize_array = array.msize;
  else
    error('Magnet size ''msize''must have three elements (or one element for a cube magnet).')
  end
end
```

```
array.dim = reshape(array.msize_array, [array.total 3]);

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


if ~isfield(array,'magn')
  if isfield(array,'grade')
    array.magn = grade2magn(array.grade);
  else
    array.magn = 1;
  end
end

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


if ~isfield(array,'magdir_fn')

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

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

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

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

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

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

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

  case 'linear-quasi'

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

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

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

case 'planar'

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

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

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

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

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

case 'patchwork'

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

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

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

case 'quasi-halbach'

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

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

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

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

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

```
end

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

nn = 0;
for iii = 1:array.mcount(1)
  for jjj = 1:array.mcount(2)
    for kkk = 1:array.mcount(3)
      nn = nn + 1;
      array.magdir(nn,:)= array.magdir_fn(iii,jjj,kkk);
    end
  end
end

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

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

for iii = 2:array.mcount(1)
  magsep_x(iii)= array.msize_array(iii-1,1,1,1)/2 ...
                + array.msize_array(iii ,1,1,1)/2 ;
end
for jjj = 2:array.mcount(2)
  magsep_y(jjj)= array.msize_array(1,jjj-1,1,2)/2 ...
                + array.msize_array(1,jjj ,1,2)/2 ;
end
for kkk = 2:array.mcount(3)
  magsep_z(kkk)= array.msize_array(1,1,kkk-1,3)/2 ...
                + array.msize_array(1,1,kkk ,3)/2 ;
end

magloc_x = cumsum(magsep_x);
magloc_y = cumsum(magsep_y);
magloc_z = cumsum(magsep_z);

for iii = 1:array.mcount(1)
  for jjj = 1:array.mcount(2)
    for kkk = 1:array.mcount(3)
      array.magloc_array(iii,jjj,kkk,:)= ...
        [magloc_x(iii); magloc_y(jjj); magloc_z(kkk)] ...
        + [iii-1; jjj-1; kkk-1].*array.mgap;
    end
  end
end
array.magloc = reshape(array.magloc_array,[array.total 3]);

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

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

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

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


end


function array_out = extrapolate_variables(array)

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

if isfield(array,'Nwaves')
  mcount_extra = 1;
else
  mcount_extra = 0;
end

if isfield(array,'mlength')
  mlength_adjust = false;
else
  mlength_adjust = true;
end

variables = nan([7 1]);

for iii = 1:length(var_names);
  if isfield(array,var_names(iii))
    variables(iii)= array.(var_names{iii});
  end
end

var_matrix = ...
    [1,  0,  0, -1,  0, -1,  0;
     0,  1,  0, -1, -1,  0,  0;
     0,  0,  1,  0, -1,  1,  0;
     0,  0,  0,  0,  0,  1,  1];

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

idx = ~isnan(variables);
var_known = var_matrix(:,idx)*variables(idx);
var_calc = var_matrix(:,~idx)\(var_results-var_known);
variables(~idx)= var_calc;
variables = exp(variables);

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

array.Nmag = round(array.Nmag)+ mcount_extra;
array.Nmag_per_wave = round(array.Nmag_per_wave);
```

```
if mlength_adjust
  array.mlength = array.mlength * (array.Nmag-mcount_extra)/array.Nmag;
end

array_out = array;

end


end
```