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

Will Robertson

July 2, 2018

**Abstract**

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

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

# Contents

# 1 User guide

(See Section 2 for installation instructions.)

## 1.1 Forces between magnets

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

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

`magnetforces` takes three mandatory inputs to specify the position and magnetisation of the first and second magnets and the displacement between them. Optional arguments appended indicate whether to calculate force and/or torque and/or stiffness and whether to calculate components in $x$- and/or $y$- and/or $z$- components respectively. The force[1] 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.

**Cuboid magnets** The first two inputs are structures containing the following fields:

**`magnet.dim`** A $(3 \times 1)$ vector of the side-lengths of the magnet.
**`magnet.grade`** The 'grade' of the magnet as a string such as `'N42'`.
**`magnet.magdir`** A vector representing the direction of the magnetisation. This may be either a $(3 \times 1)$ vector in cartesian coordinates or a $(2 \times 1)$ vector in spherical coordidates.

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

[1] 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'

**magnet.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 $(\theta, \phi)$, $\theta$ is the vertical projection of the angle around the $x$–$y$ plane ($\theta = 0$ coincident with the $x$-axis), and $\phi$ 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. $\theta$ and $\phi$ 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.[2]

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

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

**Cylindrical magnets/coils** If the dimension of the magnet (`magnet.dim`) only has two elements, or the `magnet.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.)

**magnet.dim** A $(2 \times 1)$ vector containing, respectively, the magnet radius and length.

**magnet.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:

**coil.turns** A scalar representing the number of axial turns of the coil.

**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 `magnet.type` to be ‘`coil`’ are

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

**coil.turns** A $(2 \times 1)$ containing, resp., the number of radial turns and the number of axial turns of the coil.

**coil.current** Scalar coil current flowing CCW-from-top.

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

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

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

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

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

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

## 1.2 Forces between multipole arrays of magnets

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

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

Because multipole arrays can be defined in various ways, there are several overlapping methods for specifying the structures defining an array. Please 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 $(3 \times 1)$ vector defining the size of each magnet in the array.
**`array.Nmag`** The number of magnets composing the array.
**`array.magn`** The magnetisation magnitude of each magnet.

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

Notes:

- The array must `face` in a direction orthogonal to its alignment.
- '`up`' and '`down`' are defined as synonyms for facing '`+z`' and '`-z`', respectively, and '`linear`' for array type '`linear-x`'.
- Singleton input to `msize` assumes a cube-shaped magnet.

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

`array.magdir_first` This is the angle of magnetisation in degrees around the direction of magnetisation rotation for the first magnet. It defaults to $\pm 90°$ 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.

# 2 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 I

# Magnet forces

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

Finish this off later. Please read the PDF documentation instead for now.

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;
```

Undefined calculation flags for the three directions:

```
calc_xyz = [false; false; 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', calc_xyz(1)= true;
    case 'y', calc_xyz(2)= true;
    case 'z', calc_xyz(3)= true;
    otherwise
      error(['Unknown calculation option ''',varargin{iii},''''])
  end
end
```

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

```
if all( ~calc_xyz )
  calc_xyz = [true; true; true];
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 ($-\pi/2 \leq \phi \leq \pi/2$) and `theta` is the angle around the horizontal plane ($0 \leq \theta \leq 2\pi$). This follows Matlab's definition; other conventions are commonly used as well. Remember:

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

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

```
if ~isfield(magnet_fixed,'type')
  if length(magnet_fixed.dim)== 2
    magnet_fixed.type = 'cylinder';
  else
    magnet_fixed.type = 'cuboid';
  end
end
```

```
if ~isfield(magnet_float,'type')
  if length(magnet_float.dim)== 2
    magnet_float.type = 'cylinder';
  else
    magnet_float.type = 'cuboid';
  end
end

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

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

coil_bool = false;

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

```
end

if coil_bool

  error('to do')

else

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

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

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

  elseif strcmp(magtype,'cylinder')

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

    if ~isfield(magnet_fixed,'dir')
      if ~isfield(magnet_fixed,'magdir')
        magnet_fixed.dir   = [0 0 1];
        magnet_fixed.magdir = [0 0 1];
      else
        magnet_fixed.dir = magnet_fixed.magdir;
      end
    else
% have dir
      if ~isfield(magnet_fixed,'magdir')
        magnet_fixed.magdir = magnet_fixed.dir;
      else
        magnet_fixed.magdir = [0 0 1];
      end
```

```
    end
    if ~isfield(magnet_float,'dir')
      if ~isfield(magnet_float,'magdir')
        magnet_float.dir  = [0 0 1];
        magnet_float.magdir = [0 0 1];
      else
        magnet_float.dir = magnet_float.magdir;
      end
    else
% have dir
      if ~isfield(magnet_float,'magdir')
        magnet_float.magdir = magnet_float.dir;
      else
        magnet_float.magdir = [0 0 1];
      end
    end

    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(:);

    if ~isfield(magnet_fixed,'magn')
      magnet_fixed.magn = 4*pi*1e-7*magnet_fixed.turns*magnet_fixed.current/magnet_fixed
.dim(2);
    end
```

```
    if ~isfield(magnet_float,'magn')
      magnet_float.magn = 4*pi*1e-7*magnet_float.turns*magnet_float.current/magnet_float
.dim(2);
    end

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

  end

end

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);

if strcmp(magtype,'cuboid')

  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);

end
```

# 3 Calculate for each displacement

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

```
if coil_bool

  forces_out = coil_sign*coil.dir*...
    forces_magcyl_shell_calc(mag.dim, coil.dim, squeeze(displ(cyldir,:)), J1(cyldir
), coil.current, coil.turns);

else

  if 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

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.

```
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
```

# 4 grade2magn

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

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

where $[BH]_{\max}$ is the numeric value given in the grade in MG Oe. I.e., an N42 magnet has $[BH]_{\max} = 42\,\mathrm{MG\,Oe}$. 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')
        magn = 2*sqrt(str2num(grade(2:end))/100);
      else
        magn = 2*sqrt(str2num(grade)/100);
      end
    end

  end
```

## 5 resolve_magnetisations

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

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

```
function J = resolve_magnetisations(magn,magdir)

  if length(magdir)==2
    J_r = magn;
    J_t = magdir(1);
    J_p = magdir(2);
    J   = [ J_r * cosd(J_p)* cosd(J_t); ...
      J_r * cosd(J_p)* sind(J_t); ...
      J_r * sind(J_p)];
  else
    if all(magdir == zeros(size(magdir)))
      J = [0; 0; 0];
    else
      J = magn*magdir/norm(magdir);
      J = reshape(J,[3 1]);
    end
  end

end
```

## 6 single_magnet_cyl_force

```
function forces_out = single_magnet_cyl_force(displ)

  forces_out = nan(size(displ));

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

  if ecc < eps
    forces_out = magnet_fixed.magdir*forces_cyl_calc(size1, size2, displ(cyldir),
 J1(cyldir), J2(cyldir)).';
  else
    ecc_forces = forces_cyl_ecc_calc(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);
% not 100
```

```
  end
end
```

# 7 single_magnet_force

```
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')
```

The other forces (i.e., x and y components) 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.

```
  calc_xyz = swap_x_z(calc_xyz);

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

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

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

  calc_xyz = swap_x_z(calc_xyz);

  calc_xyz = swap_y_z(calc_xyz);

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

```
  debug_disp('Forces y-y:')
  force_components(5,:)= ...
    rotate_z_to_y( forces_calc_z_z(size1_y,size2_y,d_y,J1_y,J2_y));

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

  calc_xyz = swap_y_z(calc_xyz);


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

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

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


  force_out = sum(force_components);
end
```

# 8 single_magnet_torque

```
function force_out = single_magnet_force(displ)

  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)= torques_calc_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 );

  calc_xyz = swap_x_z(calc_xyz);

  debug_disp('Torques x-x:')
  torque_components(:,:,1)= ...
    rotate_z_to_x( torques_calc_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));

  calc_xyz = swap_x_z(calc_xyz);

  calc_xyz = swap_y_z(calc_xyz);

  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( torques_calc_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));

  calc_xyz = swap_y_z(calc_xyz);

  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 );

    calc_xyz = swap_x_z(calc_xyz);

    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 ));

    calc_xyz = swap_x_z(calc_xyz);

    calc_xyz = swap_y_z(calc_xyz);

    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 ));

    calc_xyz = swap_y_z(calc_xyz);


    stiffness_out = sum(stiffness_components);
  end
```

## 9 forces_calc_z_z

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

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

```
function calc_out = forces_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);


  if calc_xyz(1)
    component_x = ...
      + multiply_x_log_y( 0.5*(v.^2-w.^2), r-u )...
      + multiply_x_log_y( u.*v, r-v )...
      + v.*w.*atan1(u.*v,r.*w)...
      + 0.5*r.*u;
  end

  if calc_xyz(2)
    component_y = ...
      + multiply_x_log_y( 0.5*(u.^2-w.^2), r-v )...
      + multiply_x_log_y( u.*v, r-u )...
      + u.*w.*atan1(u.*v,r.*w)...
      + 0.5*r.*v;
  end

  if calc_xyz(3)
    component_z = ...
      - multiply_x_log_y( u.*w, r-u )...
      - multiply_x_log_y( v.*w, r-v )...
      + u.*v.*atan1(u.*v,r.*w)...
      - r.*w;
```

```
    end

    if calc_xyz(1)
      component_x = index_sum.*component_x;
    else
      component_x = 0;
    end

    if calc_xyz(2)
      component_y = index_sum.*component_y;
    else
      component_y = 0;
    end

    if calc_xyz(3)
      component_z = index_sum.*component_z;
    else
      component_z = 0;
    end

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

    debug_disp(calc_out')

  end
```

## 10 forces_calc_z_y

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

```
  function calc_out = forces_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);

allag_correction = -1;

if calc_xyz(1)
  component_x = ...
    - multiply_x_log_y ( v .* w , r-u )...
    + multiply_x_log_y ( v .* u , r+w )...
    + multiply_x_log_y ( u .* w , r+v )...
    - 0.5 * u.^2 .* atan1( v .* w , u .* r )...
    - 0.5 * v.^2 .* atan1( u .* w , v .* r )...
    - 0.5 * w.^2 .* atan1( u .* v , w .* r );
  component_x = allag_correction*component_x;
end

if calc_xyz(2)
  component_y = ...
    0.5 * multiply_x_log_y( u.^2 - v.^2 , r+w )...
    - multiply_x_log_y( u .* w , r-u )...
    - u .* v .* atan1( u .* w , v .* r )...
    - 0.5 * w .* r;
  component_y = allag_correction*component_y;
end

if calc_xyz(3)
  component_z = ...
    0.5 * multiply_x_log_y( u.^2 - w.^2 , r+v )...
    - multiply_x_log_y( u .* v , r-u )...
    - u .* w .* atan1( u .* v , w .* r )...
    - 0.5 * v .* r;
  component_z = allag_correction*component_z;
end


if calc_xyz(1)
  component_x = index_sum.*component_x;
else
  component_x = 0;
end

if calc_xyz(2)
  component_y = index_sum.*component_y;
else
  component_y = 0;
end

if calc_xyz(3)
  component_z = index_sum.*component_z;
else
```

```
    component_z = 0;
  end

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

  debug_disp(calc_out')

end
```

## 11 forces_calc_z_x

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

  calc_xyz = swap_x_y(calc_xyz);

  forces_xyz = forces_calc_z_y(...
    swap_x_y(size1), swap_x_y(size2), rotate_x_to_y(offset),...
    J1, rotate_x_to_y(J2));

  calc_xyz = swap_x_y(calc_xyz);
  calc_out = rotate_y_to_x( forces_xyz );

end


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);


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

  if calc_xyz(2)|| calc_xyz(3)
```

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

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

  if calc_xyz(1)
    component_x = index_sum.*component_x;
  else
    component_x = 0;
  end

  if calc_xyz(2)
    component_y = index_sum.*component_y;
  else
    component_y = 0;
  end

  if calc_xyz(3)
    component_z = index_sum.*component_z;
  else
    component_z = 0;
  end

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

  debug_disp(calc_out')

end
```

## 12 stiffnesses_calc_z_y

```
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);

  if calc_xyz(1)|| calc_xyz(3)
    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 );
  end

  if calc_xyz(2)|| calc_xyz(3)
    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 );
  end

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

  if calc_xyz(1)
    component_x = index_sum.*component_x;
  else
    component_x = 0;
  end

  if calc_xyz(2)
    component_y = index_sum.*component_y;
  else
    component_y = 0;
  end

  if calc_xyz(3)
    component_z = index_sum.*component_z;
  else
    component_z = 0;
  end

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

  debug_disp(calc_out')

end
```

# 13 stiffnesses_calc_z_x

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

  calc_xyz = swap_x_y(calc_xyz);

  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_xyz = swap_x_y(calc_xyz);
  calc_out = swap_x_y(stiffnesses_xyz);

end
```

# 14 torques_calc_z_z

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

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

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

  br1 = J1(3);
  br2 = J2(3);

  if br1==0 || br2==0
    debug_disp('Zero magnetisation')
    calc_out = 0*offset;
    return
  end

  a1 = size1(1);
  b1 = size1(2);
  c1 = size1(3);

  a2 = size2(1);
  b2 = size2(2);
  c2 = size2(3);
```

```
    a = offset(1,:);
    b = offset(2,:);
    c = offset(3,:);

    d = a+lever(1,:);
    e = b+lever(2,:);
    f = c+lever(3,:);

    Tx=zeros([1 size(offset,2)]);
    Ty=Tx;
    Tz=Tx;

    for ii=[0,1]
      for jj=[0,1]
        for kk=[0,1]
          for ll=[0,1]
            for mm=[0,1]
              for nn=[0,1]

                Cu=(-1)^ii.*a1-d;
                Cv=(-1)^kk.*b1-e;
                Cw=(-1)^mm.*c1-f;

                u=a-(-1)^ii.*a1+(-1)^jj.*a2;
                v=b-(-1)^kk.*b1+(-1)^ll.*b2;
                w=c-(-1)^mm.*c1+(-1)^nn.*c2;

                s=sqrt(u.^2+v.^2+w.^2);

                Ex=(1/8).*(...
                  -2.*Cw.*(-4.*v.*u+s.^2+2.*v.*s)-...
                  w.*(-8.*v.*u+s.^2+8.*Cv.*s+6.*v.*s)+...
                  2.*(2.*Cw+w).*(u.^2+w.^2).*log(v+s)+...
                  4.*(...
                  2.*Cv.*u.*w.*acoth(u./s)+ ...
                  w.*(v.^2+2.*Cv.*v-w.*(2.*Cw+w)).*acoth(v./s)- ...
                  u.*(...
                  2*w.*(Cw+w).*atan(v./w)+ ...
                  2*v.*(Cw+w).*log(s-u)+ ...
                  (w.^2+2.*Cw.*w-v.*(2.*Cv+v)).*atan( u.*v./(w.*s))...
                  )...
                  )...
                  );

                Ey=(1/8)*...
                  ((2.*Cw+w).*u.^2-8.*u.*v.*(Cw+w)+8.*u.*v.*(Cw+w).*log(s-v)...
                  +4.*Cw.*u.*s+6.*w.*s.*u+(2.*Cw+w).*(v.^2+w.^2)+...
                  4.*w.*(w.^2+2.*Cw.*w-u.*(2.*Cu+u)).*acoth(u./s)+...
                  4.*v.*(-2.*Cu.*w.*acoth(v./s)+2.*w.*(Cw+w).*atan(u./w)...
                  +(w.^2+2.*Cw.*w-u.*(2.*Cu+u)).*atan(u.*v./(w.*s)))...
```

```
                  -2.*(2.*Cw+w).*(v.^2+w.^2).*log(u+s)+8.*Cu.*w.*s);

                Ez=(1/36).*(-u.^3-18.*v.*u.^2-6.*u.*(w.^2+6.*Cu...
                  .*v-3.*v.*(2.*Cv+v)+3.*Cv.*s)+v.*(v.^2+6.*(w.^2+...
                  3.*Cu.*s))+6.*w.*(w.^2-3.*v.*(2.*Cv+v)).*atan(u./w)...
                  -6.*w.*(w.^2-3.*u.*(2.*Cu+u)).*atan(v./w)-9.*...
                  (2.*(v.^2+2.*Cv.*v-u.*(2.*Cu+u)).*w.*atan(u.*v./(w.*s))...
                  -2.*u.*(2.*Cu+u).*v.*log(s-u)-(2.*Cv+v).*(v.^2-w.^2)...
                  .*log(u+s)+2.*u.*v.*(2.*Cv+v).*log(s-v)+(2.*Cu+...
                  u).*(u.^2-w.^2).*log(v+s)));

                Tx=Tx+(-1)^(ii+jj+kk+ll+mm+nn)*Ex;
                Ty=Ty+(-1)^(ii+jj+kk+ll+mm+nn)*Ey;
                Tz=Tz+(-1)^(ii+jj+kk+ll+mm+nn)*Ez;

              end
            end
          end
        end
      end
    end

    calc_out = real([Tx; Ty; Tz].*br1*br2/(16*pi^2*1e-7));

  end
```

## 15 torques_calc_z_y

```
  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
```

## 16 torques_calc_z_x

```
  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
```

## 17 forces_cyl_calc

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

% inputs

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

% implicit

  z = nan(4,length(h_gap));
  z(1,:)= -size1(2)/2;
  z(2,:)=  size1(2)/2;
  z(3,:)= h_gap - size2(2)/2;
  z(4,:)= h_gap + size2(2)/2;

  C_d = zeros(size(h_gap));

  for ii = [1 2]

    for jj = [3 4]

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

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

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

      f_z = a1.*a2.*a3.*( K - E./a2 - PI_term );

      f_z(abs(a1)<eps)=0; % singularity at a1=0 (i.e., coincident faces)

      C_d = C_d + (-1)^(ii+jj).*f_z;

    end

  end

  calc_out = J1*J2/(8*pi*1e-7)*C_d;

end
```

## 18 forces_cyl_ecc_calc

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

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

  z1 = -size1(2)/2;
  z2 =  size1(2)/2;
  z3 = h_gap - size2(2)/2;
  z4 = h_gap + size2(2)/2;

  h = [z4-z2; z3-z2; z4-z1; z3-z1];

  fn = @(t)[xdir(t,r1,r2,h,e_displ), zdir(t,r1,r2,h,e_displ)];
  fn_int = integral(fn,0,pi,'ArrayValued',true,'AbsTol',1e-6);

  calc_out = -1e7*J1*J2*r1*r2*fn_int/4/pi/pi;

  function gx = xdir(t,r,R,h,p)

    X = sqrt(r^2+R^2-2*r*R*cos(t));
    hh = h.^2;
    ff = (p+X)^2+hh;
    gg = (p-X)^2+hh;
    f = sqrt(ff);
    g = sqrt(gg);
    m = 1-gg./ff; % equivalent to m = 4pX/f^2

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

    Ta = f.*EE;
    Tb = (p^2-X^2).*KK./f;
    Tc = sign(p-X)*h.*( F2.*(EE-KK)+ KK.*E2 - 1 );
    Td = -pi/2*h;

    T = cos(t)/p*(Ta+Tb+Tc+Td);
    gx = -T(1)+T(2)+T(3)-T(4);

  end

  function gz = zdir(t,r,R,h,p)

    XX = p^2+R^2-2*p*R*cos(t);
    rr = r.^2;
    X = sqrt(XX);
    hh = h.^2;
    ff = (r+X)^2+hh;
    gg = (r-X)^2+hh;
    f = sqrt(ff);
    g = sqrt(gg);
    m = 1-gg./ff;
```

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

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

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

  end

end
```

## 19 ellipkepi

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

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

  a0 = 1;
  g0 = sqrt(1-m);
  s0 = m;
  nn = 0;

  p0 = sqrt(1-a);
  Q0 = 1;
  Q1 = 1;
  QQ = Q0;

  while max(Q1(:))> eps

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

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

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

```
      a0 = a1;
      g0 = g1;
      Q0 = Q1;
      p0 = p1;

    end

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

    im = find(m == 1);
    if ~isempty(im)
      k(im) = inf;
      e(im) = ones(length(im),1);
      PI(im) = inf;
    end

  end


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

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

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

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

    m(m<eps)= 0;

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

% pre-allocate space and augment if needed
      chunk = 7;
      a = zeros(chunk,1);
      c = a;
      b = a;
      a(1,:)= 1;
      c(1,:)= sqrt(m);
      b(1,:)= sqrt(1-m);
      n = uint32( zeros(1,1));
      i = 1;
      while any(abs(c(i,:))> tol)% Arithmetic-Geometric Mean of A, B and C
        i = i + 1;
```

```
      if i > size(a,1)
        a = [a; zeros(2,1)];
        b = [b; zeros(2,1)];
        c = [c; zeros(2,1)];
      end
      a(i,:) = 0.5 * (a(i-1,:)+ b(i-1,:));
      b(i,:) = sqrt(a(i-1,:).* b(i-1,:));
      c(i,:) = 0.5 * (a(i-1,:)- b(i-1,:));
      in = uint32( find((abs(c(i,:))<= tol)& (abs(c(i-1,:))> tol)));
      if ~isempty(in)
        [mi,ni] = size(in);
        n(in) = ones(mi,ni)*(i-1);
      end
    end

    mmax = length(I);
    mn = double(max(n));
    phin = zeros(1,mmax);   C  = zeros(1,mmax);
    Cp = C; e  = uint32(C); phin(:)= signU.*u(I);
    i = 0;  c2 = c.^2;
    while i < mn % Descending Landen Transformation
      i = i + 1;
      in = uint32(find(n > i));
      if ~isempty(in)
        phin(in)= atan(b(i)./a(i).*tan(phin(in)))+ ...
          pi.*ceil(phin(in)/pi - 0.5)+ phin(in);
        e(in) = 2.^(i-1);
        C(in) = C(in) + double(e(in(1)))*c2(i);
        Cp(in)= Cp(in)+ c(i+1).*sin(phin(in));
      end
    end

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

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

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

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

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

## 20 forces_magcyl_shell_calc

```
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)= forces_cyl_calc(magsize,shell_size,displ,Jmag,Jcoil);

  end

  Fz = sum(shell_forces,2);

end
```

## 21 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.

## 22 multiply_x_log_y

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

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

## 23 atan1

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

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

end
```

# References


[1] Gilles Akoun and Jean-Paul Yonnet. "3D analytical calculation of the forces exerted between two cuboidal magnets". In: *IEEE Transactions on Magnetics* MAG-20.5 (Sept. 1984), pp. 1962–1964. DOI: 10.1109/TMAG.1984.1063554 (cit. on p. 23).

[2] J.L.G. Janssen et al. "Three-Dimensional Analytical Calculation of the Torque between Permanent Magnets in Magnetic Bearings". In: *IEEE Transactions on Magnetics* 46.6 (June 2010). DOI: 10.1109/TMAG.2010.2043224 (cit. on p. 29).

[3] Jean-Paul Yonnet and Hicham Allag. "Analytical Calculation of Cuboïdal Magnet Interactions in 3D". In: *The 7th International Symposium on Linear Drives for Industry Application*. 2009 (cit. on p. 24).