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

Will Robertson

February 11, 2010

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);
stiffnesses = magnetforces( ... , 'stiffness');
      [f s] = magnetforces( ... , 'force', 'stiffness');
```

`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 or stiffness or both; the output arguments must match to reflect this choice. 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'. If you wish to calculate the force on the first magnet instead, simply reverse the sign of the output.

Inputs and outputs The first two inputs are structures containing the following fields:

`magnet.dim` A (3×1) vector of the side-lengths of the magnet.
`magnet.magn` The magnetisation magnitude of the magnet.
`magnet.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.

In cartesian coordinates, the 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:

$$(1,0,0)_{\text{cartesian}} \equiv (0,0)_{\text{spherical}}$$
$$(0,1,0)_{\text{cartesian}} \equiv (90,0)_{\text{spherical}}$$
$$(0,0,1)_{\text{cartesian}} \equiv (0,90)_{\text{spherical}}$$

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] From now I will omit most mention of calculating stiffnesses; assume whenever I say 'force' I mean 'force *and* stiffness'

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 (3×1) 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');
```

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.

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.[2] This work is Copyright 2009–2010 by Will Robertson.

Contributing and feedback Please report problems and suggestions at the GitHub issue tracker.[3]

The Matlab source code is written using Matlabweb.[4] After it is installed, use `mtangle magnetforces` to extract the Matlab files `magnetforces.m` and `multipoleforces.m`, as well as extracting the test suite (such as it is, for now). Running the Makefile with no targets (i.e., `make`) will perform this step as well as compiling the documentation you are currently reading.

3 Implementation

magnetforces

[2] `http://www.apache.org/licenses/LICENSE-2.0`
[3] `http://github.com/wspr/magnetocode/issues`
[4] `http://www.ctan.org/tex-archive/web/matlabweb/`

1. About this file. This is a 'literate programming' approach to writing Matlab code using MATLABWEB[5]. To be honest I don't know if it's any better than simply using the Matlab programming language directly. The big advantage for me is that you have access to the entire LaTeX document environment, which gives you access to vastly better tools for cross-referencing, maths typesetting, structured formatting, bibliography generation, and so on.

The downside is obviously that you miss out on Matlab's IDE with its integrated M-Lint program, debugger, profiler, and so on. Depending on one's work habits, this may be more or less of limiting factor to using literate programming in this way.

2. This work consists of the source file `magnetforces.web` and its associated derived files. It is released under the Apache License v2.0.[6]

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.

Copyright © 2009 Will Robertson.

3. Calculating forces between magnets. This is the source of some code to calculate the forces and/or stiffnesses between two cuboid-shaped magnets with arbitary displacements and magnetisation direction. (A cuboid is like a three dimensional rectangle; its faces are all orthogonal but may have different side lengths.)

[5] http://tug.ctan.org/pkg/matlabweb
[6] http://www.apache.org/licenses/LICENSE-2.0

4. The main function is *magnetforces*, which takes three mandatory arguments: *magnet_fixed*, *magnet_float*, and *displ*. These will be described in more detail below.

Optional string arguments may be any combination of `'force'`, and/or `'stiffness'` to indicate which calculations should be output. If no calculation is specified, `'force'` is the default.

Inputs:	*magnet_fixed*	structure describing first magnet
	magnet_float	structure describing the second magnet
	displ	displacement between the magnets
	[*what to calculate*]	‘force’ and/or ‘stiffness’
Outputs:	*forces*	forces on the second magnet
	stiffnesses	stiffnesses on the second magnet
Magnet properties:	*dim*	size of each magnet
	magn	magnetisation magnitude
	magdir	magnetisation direction

```
⟨magnetforces.m 4⟩ ≡
 function [varargout] =magnetforces(magnet_fixed, magnet_float, displ, varargin)
   ⟨Matlab help text (forces) 34⟩
   ⟨Parse calculation args 7⟩
   ⟨Organise input displacements 6⟩
   ⟨Initialise main variables 5⟩
   ⟨Precompute rotations 29⟩
   ⟨Calculate for each displacement 9⟩
   ⟨Return all results 8⟩
   ⟨Function for resolving magnetisations 28⟩
   ⟨Function for single force calculation 10⟩
   ⟨Function for single stiffness calculation 11⟩
   ⟨Functions for calculating forces and stiffnesses 16⟩
   end
```

5. Variables and data structures. 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 one of Matlab's structures 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 θ 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:

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

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.

⟨ Initialise main variables 5 ⟩ ≡

```
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);
```

See also section 25.

This code is used in section 4.

6. 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.

⟨ Organise input displacements 6 ⟩ ≡

```
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 = repmat(NaN, [3 Ndispl]);
end
if calc_stiffness_bool
  stiffnesses_out = repmat(NaN, [3 Ndispl]);
end
```

This code is used in sections 4 and 56.

7. Wrangling user input and output. 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`.

⟨ Parse calculation args 7 ⟩ ≡

```
debug_disp =@(str) disp([]);
calc_force_bool = false;
calc_stiffness_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;
  otherwise
    error(['Unknown␣calculation␣option␣''', varargin{ii}, ''''])
  end
end
if NOT calc_force_bool ∧ ∧ NOT calc_stiffness_bool
calc_force_bool = true;
end
```

This code is used in sections 4 and 56.

8. After all of the calculations have occured, they're placed back into `varargout`. (This happens at the very end, obviously.)

⟨ Return all results 8 ⟩ ≡

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

This code is used in sections 4 and 56.

9. 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.

⟨ Calculate for each displacement 9 ⟩ ≡

```
if calc_force_bool
  for ii = 1 : Ndispl
    forces_out(:, ii) = single_magnet_force(displ(:, ii));
  end
end
if calc_stiffness_bool
  for ii = 1 : Ndispl
    stiffnesses_out(:, ii) = single_magnet_stiffness(displ(:, ii));
  end
end
```

This code is used in section 4.

10. And this is what does the calculations.

⟨ Function for single force calculation 10 ⟩ ≡

```
function force_out = single_magnet_force(displ)
    force_components = repmat(NaN, [9 3]);
    ⟨ Precompute displacement rotations 30 ⟩
    ⟨ Print diagnostics 33 ⟩
    ⟨ Calculate x force 13 ⟩
    ⟨ Calculate y force 14 ⟩
    ⟨ Calculate z force 12 ⟩
    force_out = sum(force_components);
    end
```

This code is used in section 4.

11. And this is what does the calculations for stiffness.

⟨ Function for single stiffness calculation 11 ⟩ ≡

```
function stiffness_out = single_magnet_stiffness(displ)
    stiffness_components = repmat(NaN, [9 3]);
    ⟨ Precompute displacement rotations 30 ⟩
    ⟨ Print diagnostics 33 ⟩
    ⟨ Calculate stiffnesses 15 ⟩
    stiffness_out = sum(stiffness_components);
    end
```

This code is used in section 4.

12. The easy one first, where our magnetisation components align with the direction expected by the force functions.

⟨ Calculate z force 12 ⟩ ≡

```
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);
```

This code is used in section 10.

13. 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.

⟨ Calculate x force 13 ⟩ ≡

```
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));
```

This code is used in section 10.

14. Same again, this time making y the 'up' direction.

⟨ Calculate y force 14 ⟩ ≡

```
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));
```

This code is used in section 10.

15. Same as all the above. Except not really. Because stiffness isn't the same sort of vector quantity (if at all, really) as force, we simply 'flip' the directions around between different coordinate systems rather than rotate them.

⟨ Calculate stiffnesses 15 ⟩ ≡

```
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));
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);
```

This code is used in section 11.

16. Functions for calculating forces and stiffnesses. The calculations for forces between differently-oriented cuboid magnets are all directly from the literature. The stiffnesses have been derived by differentiating the force expressions, but that's the easy part.

```
⟨ Functions for calculating forces and stiffnesses 16 ⟩ ≡
  ⟨ Parallel magnets force calculation 17 ⟩
  ⟨ Orthogonal magnets force calculation 18 ⟩
  ⟨ Parallel magnets stiffness calculation 21 ⟩
  ⟨ Orthogonal magnets stiffness calculation 22 ⟩
  ⟨ Helper functions 31 ⟩
```

This code is used in section 4.

17. 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

```
⟨ Parallel magnets force calculation 17 ⟩ ≡
  function calc_out = forces_calc_z_z(size1, size2, offset, J1, J2)
      J1 = J1(3);
      J2 = J2(3);
      ⟨ Initialise subfunction variables 24 ⟩
      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;
      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;
      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;
      ⟨ Finish up 26 ⟩
```

This code is used in section 16.

18. Orthogonal magnets forces given by Yonnet and Allag [2].

⟨ Orthogonal magnets force calculation 18 ⟩ ≡

```
function $calc\_out = forces\_calc\_z\_y(size1, size2, offset, J1, J2)$
    $J1 = J1(3);$
    $J2 = J2(2);$
    ⟨ Initialise subfunction variables 24 ⟩
    $component\_x = \ldots$
    $-multiply\_x\_log\_y(v .* w, r - u) \ldots$
    $+multiply\_x\_log\_y(v .* u, r + w) \ldots$
    $+multiply\_x\_log\_y(u .* w, r + v) \ldots$
    $-0.5*u .\hat{}\ 2 .* atan1(v .* w, u .* r) \ldots$
    $-0.5*v .\hat{}\ 2 .* atan1(u .* w, v .* r) \ldots$
    $-0.5*w .\hat{}\ 2 .* atan1(u .* v, w .* r);$
    $component\_y = \ldots$
    $0.5*multiply\_x\_log\_y(u .\hat{}\ 2 - v .\hat{}\ 2, r + w) \ldots$
    $-multiply\_x\_log\_y(u .* w, r - u) \ldots$
    $-u .* v .* atan1(u .* w, v .* r) \ldots$
    $-0.5*w .* r;$
    $component\_z = \ldots$
    $0.5*multiply\_x\_log\_y(u .\hat{}\ 2 - w .\hat{}\ 2, r + v) \ldots$
    $-multiply\_x\_log\_y(u .* v, r - u) \ldots$
    $-u .* w .* atan1(u .* v, w .* r) \ldots$
    $-0.5*v .* r;$
    $allag\_correction = -1;$
    $component\_x = allag\_correction*component\_x;$
    $component\_y = allag\_correction*component\_y;$
    $component\_z = allag\_correction*component\_z;$
    ⟨ Finish up 26 ⟩
```

See also section 20.

This code is used in section 16.

19. This is the same calculation with Janssen's equations instead. By default this code never runs, but if you like it can be enabled to prove that the equations are consistent.

⟨Test against Janssen results 19⟩ ≡

```
S = u;
T = v;
U = w;
R = r;
component_x_ii = ...
(0.5*atan1(U, S) + 0.5*atan1(T .* U, S .* R)) .* S .^ 2...
+T .* S - 3/2*U .* S - multiply_x_log_y(S .* T, U + R) - T .^ 2 .* atan1(S,
      T)...
+U .* (U .* (...
      0.5*atan1(S, U) + 0.5*atan1(S .* T, U .* R)...
      )...
   -multiply_x_log_y(T, S + R) + multiply_x_log_y(S, R - T)...
   )...
+0.5*T .^ 2 .* atan1(S .* U, T .* R)...
;
component_y_ii = ...
0.5*U .* (R - 2*S) + ...
multiply_x_log_y(0.5*(T .^ 2 - S .^ 2), U + R) + ...
S .* T .* (atan1(U, T) + atan1(S .* U, T .* R)) + ...
multiply_x_log_y(S .* U, R - S)...
;
component_z_ii = ...
0.5*T .* (R - 2*S) + ...
multiply_x_log_y(0.5*(U .^ 2 - S .^ 2), T + R) + ...
S .* U .* (atan1(T, U) + atan1(S .* T, U .* R)) + ...
multiply_x_log_y(S .* T, R - S)...
;
xx = index_sum .* component_x;
xx_ii = index_sum .* component_x_ii;
assert(abs(sum(xx(:)) - sum(xx_ii(:))) < $1 \cdot 10^{-8}$)
yy = index_sum .* component_y;
yy_ii = index_sum .* component_y_ii;
assert(abs(sum(yy(:)) - sum(yy_ii(:))) < $1 \cdot 10^{-8}$)
zz = index_sum .* component_z;
zz_ii = index_sum .* component_z_ii;
assert(abs(sum(zz(:)) - sum(zz_ii(:))) < $1 \cdot 10^{-8}$)
component_x = component_x_ii;
component_y = component_y_ii;
component_z = component_z_ii;
```

20. The improvement in processing time between typing in the actual equals compared to just transforming the z–y case isn't worth the tedium of actually doing it.

⟨ Orthogonal magnets force calculation 18 ⟩ +≡

```
  function calc_out = forces_calc_z_x(size1, size2, offset, J1, J2)
      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_out = rotate_y_to_x(forces_xyz);
      end
```

21. Stiffness calculations are simply differentiated (in Mathematica) from the forces.

⟨ Parallel magnets stiffness calculation 21 ⟩ ≡

```
  function calc_out = stiffnesses_calc_z_z(size1, size2, offset, J1, J2)
      J1 = J1(3);
      J2 = J2(3);
      ⟨ Initialise subfunction variables 24 ⟩
      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;
      ⟨ Finish up 26 ⟩
```

This code is used in section 16.

22. Orthogonal magnets stiffnesses derived from Yonnet and Allag [2]. First the z–y magnetisation.

⟨ Orthogonal magnets stiffness calculation 22 ⟩ ≡

```
function calc_out = stiffnesses_calc_z_y(size1, size2, offset, J1, J2)
    J1 = J1(3);
    J2 = J2(2);
    ⟨ Initialise subfunction variables 24 ⟩
    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;
    ⟨ Finish up 26 ⟩
```

See also section 23.

This code is used in section 16.

23. Now the z–x magnetisation, which is z–y rotated.

⟨ Orthogonal magnets stiffness calculation 22 ⟩ +≡

```
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
```

24. Some shared setup code. First **return** early if either of the magnetisations are zero — that's the trivial solution. Assume that the magnetisation has already been rounded down to zero if necessary; i.e., that we don't need to check for *J1* or *J2* are less than $1 \cdot 10^{-12}$ or whatever.

⟨ Initialise subfunction variables 24 ⟩ ≡

```
**if** ( $J1 \equiv 0$ OR $J2 \equiv 0$ )
*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);
```

This code is used in sections 17, 18, 21, and 22.

25. Here are some variables used above that only need to be computed once. The idea here is to vectorise instead of using **for** loops because it allows more convenient manipulation of the data later on.

⟨ Initialise main variables 5 ⟩ +≡

```
*magconst* = 1/(4∗π∗(4∗π∗1 · 10^−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*);
```

26. And some shared finishing code.

⟨ Finish up 26 ⟩ ≡

```
*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**
```

This code is used in sections 17, 18, 21, and 22.

27. Setup code.

28. 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 for resolving magnetisations 28 ⟩ ≡

```
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
```

This code is used in section 4.

29. Forces due to magnetisations in x and y are calculated by rotating the original expressions. The rotated magnet sizes and magnetisation vectors are calculated here once only.

The rotation matrices are precalculated to avoid performing the matrix multiplications each time.

⟨ Precompute rotations 29 ⟩ ≡

```
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);
```

This code is used in section 4.

30. And the rotated displacement vectors are calculated once per loop:

⟨ Precompute displacement rotations 30 ⟩ ≡

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

This code is used in sections 10 and 11.

31. 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.

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

⟨ Helper functions 31 ⟩ ≡

```
function out = multiply_x_log_y(x, y)
    out = x .* log(y);
    out(NOT isfinite(out)) = 0;
    end
```

See also section 32.

This code is used in section 16.

32. Also, 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`.

This function computes `atan` but takes two arguments.

⟨ Helper functions 31 ⟩ +≡

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

33. Let's print some information to the terminal to aid debugging. This is especially important (for me) when looking at the rotated coordinate systems.

⟨ Print diagnostics 33 ⟩ ≡

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

This code is used in sections 10 and 11.

34. When users type `help magnetforces` this is what they see.

⟨ Matlab help text (forces) 34 ⟩ ≡

```
%% MAGNETFORCES  Calculate forces between two cuboid magnets
%
% Finish this off later.  Please read the PDF documentation instead for now.
%
```

This code is used in section 4.

35. Test files. The chunks that follow are designed to be saved into individual files and executed automatically to check for (a) correctness and (b) regression problems as the code evolves.

How do I know if the code produces the correct forces? Well, for many cases I can compare with published values in the literature. Beyond that, I'll be setting up some tests that I can logically infer should produce the same results (such as mirror-image displacements) and test that.

There are many Matlab unit test frameworks but I'll be using a fairly low-tech method. In time this test suite should be (somehow) useable for all implementations of `magnetocode`, not just Matlab. But I haven't thought about doing anything like that, yet.

36. Because I'm lazy, just run the tests manually for now. This script must be run twice if it updates itself.

```
⟨testall.m 36⟩ ≡
  clc;
  magforce_test001a
  magforce_test001b
  magforce_test001c
  magforce_test001d
```

See also section 74.

37. Force testing. This test checks that square magnets produce the same forces in the each direction when displaced in positive and negative x, y, and z directions, respectively. In other words, this tests the function *forces_calc_z_y* directly. Both positive and negative magnetisations are used.

```
⟨magforce_test001a.m 37⟩ ≡
  disp('=================')
  fprintf('TEST␣001a:␣')
  magnet_fixed.dim = [0.04 0.04 0.04];
  magnet_float.dim = magnet_fixed.dim;
  magnet_fixed.magn = 1.3;
  magnet_float.magn = 1.3;
  offset = 0.1;
  ⟨Test z–z magnetisations 38⟩
  ⟨Assert magnetisations tests 46⟩
  ⟨Test x–x magnetisations 39⟩
  ⟨Assert magnetisations tests 46⟩
  ⟨Test y–y magnetisations 40⟩
  ⟨Assert magnetisations tests 46⟩
  fprintf('passed\n')
  disp('=================')
```

38. Testing vertical forces.

⟨ Test z–z magnetisations 38 ⟩ ≡

```
f = [];
for ii = [1, −1]
   magnet_fixed.magdir = [0 ii*90];         % ±z
   for jj = [1, −1]
      magnet_float.magdir = [0 jj*90];
      for kk = [1, −1]
         displ = kk*[0 0 offset];
         f(:, end + 1) = magnetforces(magnet_fixed, magnet_float, displ);
      end
   end
end
dirforces = chop(f(3, :), 8);
otherforces = f([1 2], :);
```

This code is used in section 37.

39. Testing horizontal x forces.

⟨ Test x–x magnetisations 39 ⟩ ≡

```
f = [];
for ii = [1, −1]
   magnet_fixed.magdir = [90 + ii*90 0];         % ±x
   for jj = [1, −1]
      magnet_float.magdir = [90 + jj*90 0];
      for kk = [1, −1]
         displ = kk*[offset 0 0];
         f(:, end + 1) = magnetforces(magnet_fixed, magnet_float, displ);
      end
   end
end
dirforces = chop(f(1, :), 8);
otherforces = f([2 3], :);
```

This code is used in section 37.

40. Testing horizontal y forces.

⟨ Test y–y magnetisations 40 ⟩ ≡

```
f = [];
for ii = [1, −1]
  magnet_fixed.magdir = [ii∗90 0];        % ±y
  for jj = [1, −1]
    magnet_float.magdir = [jj∗90 0];
    for kk = [1, −1]
      displ = kk∗[0 offset 0];
      f(:, end + 1) = magnetforces(magnet_fixed, magnet_float, displ);
    end
  end
end
dirforces = chop(f(2, :), 8);
otherforces = f([1 3], :);
```

This code is used in section 37.

41. This test does the same thing but for orthogonally magnetised magnets.

⟨ magforce_test001b.m 41 ⟩ ≡

```
disp('=================')
fprintf('TEST␣001b:␣')
magnet_fixed.dim = [0.04 0.04 0.04];
magnet_float.dim = magnet_fixed.dim;
magnet_fixed.magn = 1.3;
magnet_float.magn = 1.3;
⟨ Test ZYZ 42 ⟩
⟨ Assert magnetisations tests 46 ⟩
⟨ Test ZXZ 43 ⟩
⟨ Assert magnetisations tests 46 ⟩
⟨ Test ZXX 45 ⟩
⟨ Assert magnetisations tests 46 ⟩
⟨ Test ZYY 44 ⟩
⟨ Assert magnetisations tests 46 ⟩
fprintf('passed\n')
disp('=================')
```

42. z–y magnetisations, z displacement.

⟨ Test ZYZ 42 ⟩ ≡

```
fzyz = [ ];
for ii = [1, −1]
  for jj = [1, −1]
    for kk = [1, −1]
      magnet_fixed.magdir = ii∗[0 90];     % ±z
      magnet_float.magdir = jj∗[90 0];     % ±y
      displ = kk∗[0 0 0.1];     % ±z
      fzyz(:, end + 1) = magnetforces(magnet_fixed, magnet_float,
          displ);
    end
  end
end
dirforces = chop(fzyz(2, :), 8);
otherforces = fzyz([1 3], :);
```

This code is used in section 41.

43. z–x magnetisations, z displacement.

⟨ Test ZXZ 43 ⟩ ≡

```
fzxz = [ ];
for ii = [1, −1]
  for jj = [1, −1]
    for kk = [1, −1]
      magnet_fixed.magdir = ii∗[0 90];     % ±z
      magnet_float.magdir = [90 + jj∗90 0];     % ±x
      displ = kk∗[0.1 0 0];     % ±x
      fzxz(:, end + 1) = magnetforces(magnet_fixed, magnet_float,
          displ);
    end
  end
end
dirforces = chop(fzxz(3, :), 8);
otherforces = fzxz([1 2], :);
```

This code is used in section 41.

44. z–y magnetisations, y displacement.

⟨ Test ZYY 44 ⟩ ≡

```
fzyy = [];
for ii = [1, −1]
  for jj = [1, −1]
    for kk = [1, −1]
      magnet_fixed.magdir = ii∗[0 90];      % ±z
      magnet_float.magdir = jj∗[90 0];      % ±y
      displ = kk∗[0 0.1 0];      % ±y
      fzyy(:, end + 1) = magnetforces(magnet_fixed, magnet_float,
          displ);
    end
  end
end
dirforces = chop(fzyy(3, :), 8);
otherforces = fzyy([1 2], :);
```

This code is used in section 41.

45. z–x magnetisations, x displacement.

⟨ Test ZXX 45 ⟩ ≡

```
fzxx = [];
for ii = [1, −1]
  for jj = [1, −1]
    for kk = [1, −1]
      magnet_fixed.magdir = ii∗[0 90];      % ±z
      magnet_float.magdir = [90 + jj∗90 0];      % ±x
      displ = kk∗[0 0 0.1];      % ±z
      fzxx(:, end + 1) = magnetforces(magnet_fixed, magnet_float,
          displ);
    end
  end
end
dirforces = chop(fzxx(1, :), 8);
otherforces = fzxx([2 3], :);
```

This code is used in section 41.

46. The assertions, common between directions.

⟨ Assert magnetisations tests 46 ⟩ ≡

```
assert(...
  all(abs(otherforces(:)) < 1 · 10^{-11}), ...
  'Orthogonal␣forces␣should␣be␣zero' ...
  )
assert(...
  all(abs(dirforces) ≡ abs(dirforces(1))), ...
  'Force␣magnitudes␣should␣be␣equal' ...
  )
assert(...
  all(dirforces(1 : 4) ≡ −dirforces(5 : 8)), ...
  'Forces␣should␣be␣opposite␣with␣reversed␣fixed␣magnet␣magnetisation' ...
  )
assert(...
  all(dirforces([1 3 5 7]) ≡ −dirforces([2 4 6 8])), ...
  'Forces␣should␣be␣opposite␣with␣reversed␣float␣magnet␣magnetisation' ...
  )
```

This code is used in sections 37 and 41.

47. Now try combinations of displacements.

⟨ magforce_test001c.m 47 ⟩ ≡

```
disp('=================')
fprintf('TEST␣001c:␣')
magnet_fixed.dim = [0.04 0.04 0.04];
magnet_float.dim = magnet_fixed.dim;
magnet_fixed.magn = 1.3;
magnet_float.magn = 1.3;
⟨ Test combinations ZZ 48 ⟩
⟨ Assert combinations tests 50 ⟩
⟨ Test combinations ZY 49 ⟩
⟨ Assert combinations tests 50 ⟩
fprintf('passed\n')
disp('=================')
```

48. Tests.

⟨ Test combinations ZZ 48 ⟩ ≡

```
f = [];
for ii = [−1 1]
  for jj = [−1 1]
    for xx = 0.12*[−1, 1]
      for yy = 0.12*[−1, 1]
        for zz = 0.12*[−1, 1]
          magnet_fixed.magdir = [0 ii*90];        % z
          magnet_float.magdir = [0 jj*90];        % z
          displ = [xx yy zz];
          f(:, end + 1) = magnetforces(magnet_fixed, magnet_float,
              displ);
        end
      end
    end
  end
end
f = chop(f, 8);
uniquedir = f(3, :);
otherdir = f([1 2], :);
```

This code is used in section 47.

49. Tests.

⟨ Test combinations ZY 49 ⟩ ≡

```
f = [];
for ii = [−1 1]
  for jj = [−1 1]
    for xx = 0.12*[−1, 1]
      for yy = 0.12*[−1, 1]
        for zz = 0.12*[−1, 1]
          magnet_fixed.magdir = [0 ii*90];        % ±z
          magnet_float.magdir = [jj*90 0];        % ±y
          displ = [xx yy zz];
          f(:, end + 1) = magnetforces(magnet_fixed, magnet_float,
              displ);
        end
      end
    end
  end
end
f = chop(f, 8);
uniquedir = f(1, :);
otherdir = f([2 3], :);
```

This code is used in section 47.

50. Shared tests, again.

⟨ Assert combinations tests 50 ⟩ ≡

```
test1 = abs(diff(abs(f(1, :)))) < 1 · 10^−10;
test2 = abs(diff(abs(f(2, :)))) < 1 · 10^−10;
test3 = abs(diff(abs(f(3, :)))) < 1 · 10^−10;
assert( all(test1) ∧ ∧ all(test2) ∧ ∧ all(test3), . . .
  'All␣forces␣in␣a␣single␣direction␣should␣be␣equal' )
test = abs(diff(abs(otherdir))) < 1 · 10^−11;
assert(all(test), 'Orthogonal␣forces␣should␣be␣equal')
test1 = f(:, 1 : 8) ≡ f(:, 25 : 32);
test2 = f(:, 9 : 16) ≡ f(:, 17 : 24);
assert( all(test1(:)) ∧ ∧ all(test2(:)), . . .
  'Reverse␣magnetisation␣shouldn''t␣make␣a␣difference' )
```

This code is used in section 47.

51. Now we want to try non-orthogonal magnetisation.

⟨magforce_test001d.m 51⟩ ≡

```
disp('=================')
fprintf('TEST␣001d:␣')
magnet_fixed.dim = [0.04 0.04 0.04];
magnet_float.dim = magnet_fixed.dim;
        % Fixed parameters:
magnet_fixed.magn = 1.3;
magnet_float.magn = 1.3;
magnet_fixed.magdir = [0 90];        % z
displ = 0.12*[1 1 1];
⟨Test XY superposition 52⟩
⟨Assert superposition 55⟩
⟨Test XZ superposition 53⟩
⟨Assert superposition 55⟩
⟨Test planar superposition 54⟩
⟨Assert superposition 55⟩
fprintf('passed\n')
disp('=================')
```

52. Test with a magnetisation unit vector of $(1, 1, 0)$.

⟨Test XY superposition 52⟩ ≡

```
magnet_float.magdir = [45 0];        % $\vec{e}_x + \vec{e}_y$
f1 = magnetforces(magnet_fixed, magnet_float, displ);
        % Components:
magnet_float.magdir = [0 0];        % $\vec{e}_x$
fc1 = magnetforces(magnet_fixed, magnet_float, displ);
magnet_float.magdir = [90 0];        % $\vec{e}_y$
fc2 = magnetforces(magnet_fixed, magnet_float, displ);
f2 = (fc1 + fc2)/sqrt(2);
```

This code is used in section 51.

53. Test with a magnetisation unit vector of $(1,0,1)$.

⟨ Test XZ superposition 53 ⟩ ≡

```
magnet_float.magdir = [0 45];       % $\vec{e}_y + \vec{e}_z$
f1 = magnetforces(magnet_fixed, magnet_float, displ);
    % Components:
magnet_float.magdir = [0 0];        % $\vec{e}_x$
fc1 = magnetforces(magnet_fixed, magnet_float, displ);
magnet_float.magdir = [0 90];       % $\vec{e}_z$
fc2 = magnetforces(magnet_fixed, magnet_float, displ);
f2 = (fc1 + fc2)/sqrt(2);
```

This code is used in section 51.

54. Test with a magnetisation unit vector of $(1,1,1)$. This is about as much as I can be bothered testing for now. Things seem to be working.

⟨ Test planar superposition 54 ⟩ ≡

```
[t p r] = cart2sph(1/sqrt(3), 1/sqrt(3), 1/sqrt(3));
magnet_float.magdir = [t p]*180/π;      % $\vec{e}_y + \vec{e}_z + \vec{e}_z$
f1 = magnetforces(magnet_fixed, magnet_float, displ);
    % Components:
magnet_float.magdir = [0 0];        % $\vec{e}_x$
fc1 = magnetforces(magnet_fixed, magnet_float, displ);
magnet_float.magdir = [90 0];       % $\vec{e}_y$
fc2 = magnetforces(magnet_fixed, magnet_float, displ);
magnet_float.magdir = [0 90];       % $\vec{e}_z$
fc3 = magnetforces(magnet_fixed, magnet_float, displ);
f2 = (fc1 + fc2 + fc3)/sqrt(3);
```

This code is used in section 51.

55. The assertion is the same each time.

⟨ Assert superposition 55 ⟩ ≡

```
assert(...
  isequal(chop(f1, 4), chop(f2, 4)), ...
  'Components␣should␣sum␣due␣to␣superposition' ...
  )
```

This code is used in section 51.

56. Forces between (multipole) magnet arrays. This function uses `magnetforces.m` to compute the forces between two multipole magnet arrays. As before, we can calculate either force and/or stiffness in all three directions.

The structure of the function itself should look fairly straightforward. Some of the code is repeated from *magnetforces* (an advantage of the literate programming approach) for parsing the inputs for which calculations to perform and return.

```
⟨multipoleforces.m 56⟩ ≡
  function [varargout] = multipoleforces(fixed_array, float_array, displ, varargin)
    ⟨Matlab help text (multipole) 73⟩
    ⟨Parse calculation args 7⟩
    ⟨Organise input displacements 6⟩
    ⟨Initialise multipole variables 60⟩
    ⟨Calculate array forces 59⟩
    ⟨Return all results 8⟩
    ⟨Multipole sub-functions 57⟩
    end
```

57. And sub-functions.

```
⟨Multipole sub-functions 57⟩ ≡
  ⟨Create arrays from input variables 61⟩
  ⟨Extrapolate variables from input 72⟩
```

This code is used in section 56.

58. Although the input to these functions is described in the user guide, there's a quick summary in Tables 1 and 2.

Table 1: Description of `multipoleforces` data structures.

Inputs:	*fixed_array*	structure describing first magnet array
	float_array	structure describing the second magnet array
	displ	displacement between first magnet of each array
	[*what to calculate*]	'force' and/or 'stiffness'
Outputs:	*forces*	forces on the second array
	stiffnesses	stiffnesses on the second array
Arrays:	`type`	See Table 2
	align	See Table 3
	face	See Table 4
	mcount	[i j k] magnets in each direction
	msize	size of each magnet
	mgap	gap between successive magnets
	magn	magnetisation magnitude
	magdir_fn	function to calculate the magnetisation direction

Table 2: Possibilities for the `type` of a multipole array.

`generic`	Magnetisation directions &c. are defined manually
`linear`	Linear Halbach array
`planar`	Planar Halbach array
`quasi-Halbach`	Quasi-Halbach planar array
`patchwork`	'Patchwork' planar array

Table 3: Axes or plane with which to align the array, set with `align`.

`x`, `y`, `z`	For linear arrays
`xy`, `yz`, `xz`	For planar arrays

Table 4: Facing direction for the strong side of the array, set with `face`.

`+x`, `-x`	Horizontal
`+y`, `-y`	Horizontal
`+z`, `-z`, `up`, `down`	Vertical

59. Actual calculation of the forces. To calculate these forces, let's assume that we have two large arrays enumerating the positions and magnetisations of each individual magnet in each magnet array.

Required fields for each magnet array:

`total`	M total number of magnets in the array
`dim`	$(M \times 3)$ size of each magnet
`magloc`	$(M \times 3)$ location of each magnet from the local coodinate system of the array
`magn`	$(M \times 1)$ magnetisation magnitude of each magnet
`magdir`	$(M \times 2)$ magnetisation direction of each magnet in spherical coordinates
`size`	$(M \times 3)$ total actual dimensions of the array

Then it's just a matter of actually calculating each force and summing them together, as shown below. We'll discuss how to actually populate these data structures later.

⟨ Calculate array forces 59 ⟩ ≡

```
for ii = 1 : fixed_array.total
  fixed_magnet = struct(...
    'dim', fixed_array.dim(ii, :), ...
    'magn', fixed_array.magn(ii), ...
    'magdir', fixed_array.magdir(ii, :) ...
    );
  for jj = 1 : float_array.total
    float_magnet = struct(...
      '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 ∧ ∧ NOT calc_stiffness_bool
    array_forces(:, :, ii, jj) = ...
    magnetforces(fixed_magnet, float_magnet, mag_displ, 'force');
    elseif calc_stiffness_bool ∧ ∧ NOT calc_force_bool
    array_stiffnesses(:, :, ii, jj) = ...
    magnetforces(fixed_magnet, float_magnet, mag_displ,
        'stiffness');
    else
    [array_forces(:, :, ii, jj) array_stiffnesses(:, :, ii, jj)] = ...
    magnetforces(fixed_magnet, float_magnet, mag_displ, 'force',
        'stiffness');
    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
```

This code is used in section 56.

60. This is where it begins. This is basically just initialisation, but note the important *complete_array_from_input* function. This is what takes the high-level Halbach array (or whatever array) descriptions and translates them into a more direct (if tedious) form.

⟨Initialise multipole variables 60⟩ ≡

```
  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 = repmat(NaN,
        [3 Ndispl fixed_array.total float_array.total]);
  end
  if calc_stiffness_bool
    array_stiffnesses = repmat(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]);
```

This code is used in section 56.

61. From user input to array generation. We separate the force calculation from transforming the inputs into an intermediate form used for that purpose. This will hopefully allow us a little more flexibility.

This is the magic abstraction behind *complete_array_from_input* that allows us to write readable input code describing multipole arrays in as little detail as possible.

As input variables for a linear multipole array, we want to use some combination of the following:

w	wavelength of magnetisation
l	length of the array without magnet gaps
N	number of wavelengths
d	magnet length
T	total number of magnets
M	number of magnets per wavelength
ϕ	rotation between successive magnets

These are related via the following equations of constraint:

$$w = Md \qquad l = Td \qquad N = T/M \qquad M = 360^\circ/\phi \tag{1}$$

Taking logarithms and writing in matrix form yields

$$\begin{bmatrix} 1 & 0 & 0 & -1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix} \log \begin{bmatrix} w \\ l \\ N \\ d \\ T \\ M \\ \phi \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \log(360^\circ) \end{bmatrix} \tag{2}$$

We can use this matrix to compute whichever variables we need given enough inputs.

However, we generally do not want an integer number of wavelengths of magnetisation in the magnet arrays; if $T = MN$ then we get small lateral forces that are undesirable for stability. We prefer instead to have $T = MN + 1$, but this cannot be represented by our linear (after taking logarithms) algebra above. Therefore, if the user requests a total number of wavelengths of magnetisation, we automatically add one end magnet to restore the symmetry of the forces.

More variables that can be set are:

ϕ_0	magnetisation direction of the first magnet
g	additional gap between adjacent magnet faces (optional)
e	array height (or magnet height)
f	array width (or magnet width)

For both technical reasons and reasons of convenience, the length of the array l does not take into account any specified magnet gap g. In other words, l is actually the length of the possibly discontiguous magnetic material; the length of the array will be $l + (N - 1)g$.

⟨Create arrays from input variables 61⟩ ≡

```
function array = complete_array_from_input(array)
    if NOT isfield(array, 'type')
      array.type = 'generic';
    end
    ⟨Set alignment/facing directions 63⟩
    switch array.type
    case 'linear'
      ⟨Infer linear array variables 64⟩
    case 'linear-quasi'
      ⟨Infer linear-quasi array variables 65⟩
    case 'planar'
      ⟨Infer planar array variables 66⟩
    case 'quasi-halbach'
      ⟨Infer quasi-Halbach array variables 67⟩
    case 'patchwork'
      ⟨Infer patchwork array variables 68⟩
    end
    ⟨Array sizes 69⟩
    ⟨Array magnetisation strengths 70⟩
    ⟨Array magnetisation directions 71⟩
    ⟨Fill in array structures 62⟩
    end
```

This code is used in section 57.

62. This is the part where those big data structures are filled up based on the user input data. I guess you could consider the process to consist of three stages. User input is the most abstract, from which the code above infers the other variables that have only been implied. Then the following code uses all that to construct a most basic description of the arrays, literally a listing of each magnet, its dimensions and position, and its magnetisation vector.

⟨ Fill in array structures 62 ⟩ ≡

```
array.magloc = repmat(NaN, [array.total 3]);
array.magdir = array.magloc;
arrat.magloc_array = repmat(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)
```
This code is used in section 61.

63. For all arrays that aren't *generic*, an alignment direction(s) and facing direction can be specified. By default, arrays face upwards and are aligned along x for linear arrays and on the x–y plane for planar.

```
⟨Set alignment/facing directions 63⟩ ≡
  if NOT 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 NOT isequal(array.type, 'generic')
    if linear_index ≡ 1
      if NOT 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 NOT 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 $linear\_index \neq 0$
  if $linear\_index \equiv facing\_index$
    error('Arrays␣cannot␣face␣into␣their␣alignment␣direction.')
  end
elseif NOT isequal($planar\_index$, [0 0])
  if any($planar\_index \equiv facing\_index$)
    error('Planar-type␣arrays␣can␣only␣face␣into␣their␣orthogonal␣direction')
  end
end
```

This code is used in section 61.

64. We need to finish off infering those variables that weren't specified but are implicit. This will be different for each type of multipole array, as you would have picked up on by now.

⟨Infer linear array variables 64⟩ ≡

```
$array = extrapolate\_variables(array)$;
$array.mcount$ = ones(1, 3);
$array.mcount(linear\_index) = array.Nmag$;
```

This code is used in section 61.

65. The *linear − quasi* array is like the linear Halbach array but always has (except in the degenerate case) four magnets per wavelength. The magnet sizes are not equal.

⟨ Infer linear-quasi array variables 65 ⟩ ≡

```
if isfield(array, 'ratio') ∧ ∧ isfield(array, 'mlength')
error('Cannot␣specify␣both␣''ratio''␣and␣''mlength''.')
elseif NOT isfield(array, 'ratio') ∧ ∧ NOT 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 = repmat(NaN, [array.mcount 3]);
[sindex_x sindex_y sindex_z] = ...
meshgrid(1 : array.mcount(1), 1 : array.mcount(2), 1 : array.mcount(3));
    %also.
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) = ...
  $array.mlength(\mathrm{mod}(ii - 1, 2) + 1)$;
  $array.msize(sindex\_x(ii),\ sindex\_y(ii),\ sindex\_z(ii),$
      $facing\_index) = \ldots$
  $array.height$;
  $array.msize(sindex\_x(ii),\ sindex\_y(ii),\ sindex\_z(ii),$
      $width\_index) = \ldots$
  $array.width$;
end
```

This code is used in section 61.

66. For now it's a bit more messy to do the planar array variables.

⟨ Infer planar array variables 66 ⟩ ≡

```
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;
```

This code is used in section 61.

67. The other two planar arrays are less complicated than the planar Halbach array above. Still lots of annoying variable-wrangling, though.

⟨ Infer quasi-Halbach array variables 67 ⟩ ≡

```
  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
```

This code is used in section 61.

68. Basically the same for the patchwork array but without worrying about wavelengths.

⟨ Infer patchwork array variables 68 ⟩ ≡

```
if isfield(array, 'mcount')
  if numel(array.mcount) $\neq 3$
    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
```

This code is used in section 61.

69. Sizes.

⟨ Array sizes 69 ⟩ ≡

```
array.total = prod(array.mcount);
if NOT 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 NOT 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 NOT isfield(array, 'mgap')
  array.mgap = [0;  0;  0];
elseif length(array.mgap) ≡ 1
  array.mgap = repmat(array.mgap, [3 1]);
end
```

This code is used in section 61.

70. Magnetisation strength of each magnet.

⟨ Array magnetisation strengths 70 ⟩ ≡

```
if NOT isfield(array, 'magn')
  array.magn = 1;
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
```

This code is used in section 61.

71. Magnetisation direction of each magnet.

⟨ Array magnetisation directions 71 ⟩ ≡

```
if NOT isfield(array, 'magdir_fn')
  if NOT 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 NOT 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
```

This code is used in section 61.

72. Sub-functions.

⟨ Extrapolate variables from input 72 ⟩ ≡

```
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 = repmat(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 = NOT isnan(variables);
    var_known = var_matrix(:, idx)∗variables(idx);
    var_calc = var_matrix(:, NOT idx)\(var_results − var_known);
    variables(NOT 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

This code is used in section 57.

73. When users type `help multipoleforces` this is what they see.

⟨ Matlab help text (multipole) 73 ⟩ ≡

```
%% MULTIPOLEFORCES  Calculate forces between two multipole arrays of magnets
%
% Finish this off later.  Please read the PDF documentation instead for now.
%
```

This code is used in section 56.

74. Test files for multipole arrays. Not much here yet.

⟨testall.m 36⟩ +≡

```
multiforce_test002a
multiforce_test002b
multiforce_test002c
multiforce_test002d
multiforce_test003a
```

75. First test just to check the numbers aren't changing.

⟨multiforce_test002a.m 75⟩ ≡

```
disp('=================')
fprintf('TEST␣002a:␣')
fixed_array = ...
struct(...
  'type', 'linear', ...
  'align', 'x', ...
  'face', 'up', ...
  'length', 0.01, ...
  'width', 0.01, ...
  'height', 0.01, ...
  'Nmag_per_wave', 4, ...
  'Nwaves', 1, ...
  'magn', 1, ...
  'magdir_first', 90...
  );
float_array = fixed_array;
float_array.face = 'down';
float_array.magdir_first = −90;
displ = [0 0 0.02];
f_total = multipoleforces(fixed_array, float_array, displ);
assert(chop(f_total(3), 5) ≡ 0.13909, 'Regression␣shouldn''t␣fail');
fprintf('passed\n')
disp('=================')
```

76. Test against single magnet.

⟨multiforce_test002b.m 76⟩ ≡

```
disp('=================')
fprintf('TEST␣002b:␣')
fixed_array = ...
struct(...
  'type', 'linear', ...
  'align', 'x', ...
  'face', 'up', ...
  'length', 0.01, ...
  'width', 0.01, ...
  'height', 0.01, ...
  'Nmag_per_wave', 1, ...
  'Nwaves', 1, ...
  'magn', 1, ...
  'magdir_first', 90 ...
  );
float_array = fixed_array;
float_array.face = 'down';
float_array.magdir_first = −90;
displ = [0 0 0.02];
f_total = multipoleforces(fixed_array, float_array, displ);
fixed_mag = struct('dim', [0.01 0.01 0.01], 'magn', 1, 'magdir', [0 90]);
float_mag = struct('dim', [0.01 0.01 0.01], 'magn', 1, 'magdir', [0 − 90]);
f_mag = magnetforces(fixed_mag, float_mag, displ);
assert(chop(f_total(3), 6) ≡ chop(f_mag(3), 6));
fprintf('passed\n')
disp('=================')
```

77. Test that linear arrays give consistent results regardless of orientation.

⟨multiforce_test002c.m 77⟩ ≡

```
disp('=================')
fprintf('TEST␣002c:␣')
      % Fixed parameters
fixed_array = ...
struct(...
  'length', 0.10, ...
  'width', 0.01, ...
  'height', 0.01, ...
  'Nmag_per_wave', 4, ...
  'Nwaves', 1, ...
  'magn', 1, ...
  'magdir_first', 90 ...
  );
float_array = fixed_array;
float_array.magdir_first = −90;
f = repmat(NaN, [3 0]);
      % The varying calculations
fixed_array.type = 'linear';
float_array.type = fixed_array.type;
fixed_array.align = 'x';
float_array.align = fixed_array.align;
fixed_array.face = 'up';
float_array.face = 'down';
displ = [0 0 0.02];
f( : , end +1 ) = multipoleforces(fixed_array, float_array, displ);
fixed_array.type = 'linear';
float_array.type = fixed_array.type;
fixed_array.align = 'x';
float_array.align = fixed_array.align;
fixed_array.face = '+y';
float_array.face = '-y';
displ = [0 0.02 0];
f( : , end +1 ) = multipoleforces(fixed_array, float_array, displ);
fixed_array.type = 'linear';
float_array.type = fixed_array.type;
fixed_array.align = 'y';
float_array.align = fixed_array.align;
fixed_array.face = 'up';
float_array.face = 'down';
displ = [0 0 0.02];
f( : , end +1 ) = multipoleforces(fixed_array, float_array, displ);
```

```
fixed_array.type = 'linear';
float_array.type = fixed_array.type;
fixed_array.align = 'y';
float_array.align = fixed_array.align;
fixed_array.face = '+x';
float_array.face = '-x';
displ = [0.02 0 0];
f( : , end +1 ) = multipoleforces(fixed_array, float_array, displ);
fixed_array.type = 'linear';
float_array.type = fixed_array.type;
fixed_array.align = 'z';
float_array.align = fixed_array.align;
fixed_array.face = '+x';
float_array.face = '-x';
displ = [0.02 0 0];
f( : , end +1 ) = multipoleforces(fixed_array, float_array, displ);
fixed_array.type = 'linear';
float_array.type = fixed_array.type;
fixed_array.align = 'z';
float_array.align = fixed_array.align;
fixed_array.face = '+y';
float_array.face = '-y';
displ = [0 0.02 0];
f( : , end +1 ) = multipoleforces(fixed_array, float_array, displ);
assert(all(chop(sum(f), 4) ≡ 37.31), ...
  'Arrays␣aligned␣in␣different␣directions␣should␣produce␣consistent␣results.');
fprintf('passed\n')
disp('=================')
```

78. Test that planar arrays give consistent results regardless of orientation.

⟨multiforce_test002d.m 78⟩ ≡

```
disp('================')
fprintf('TEST␣002d:␣')
      % Fixed parameters
fixed_array = ...
struct(...
  'length', [0.10 0.10], ...
  'width', 0.10, ...
  'height', 0.01, ...
  'Nmag_per_wave', [4 4], ...
  'Nwaves', [1 1], ...
  'magn', 1, ...
  'magdir_first', [90 90] ...
  );
float_array = fixed_array;
float_array.magdir_first = [−90 − 90];
f = repmat(NaN, [3 0]);
      % The varying calculations
fixed_array.type = 'planar';
float_array.type = fixed_array.type;
fixed_array.align = 'xy';
float_array.align = fixed_array.align;
fixed_array.face = 'up';
float_array.face = 'down';
displ = [0 0 0.02];
f( : , end +1 ) = multipoleforces(fixed_array, float_array, displ);
fixed_array.type = 'planar';
float_array.type = fixed_array.type;
fixed_array.align = 'yz';
float_array.align = fixed_array.align;
fixed_array.face = '+x';
float_array.face = '-x';
displ = [0.02 0 0];
f( : , end +1 ) = multipoleforces(fixed_array, float_array, displ);
fixed_array.type = 'planar';
float_array.type = fixed_array.type;
fixed_array.align = 'xz';
float_array.align = fixed_array.align;
fixed_array.face = '+y';
float_array.face = '-y';
displ = [0 0.02 0];
f( : , end +1 ) = multipoleforces(fixed_array, float_array, displ);
ind = [3 4 8];
```

```
assert(all(round(f(ind)*100)/100 ≡ 589.05), ...
  'Arrays␣aligned␣in␣different␣directions␣should␣produce␣consistent␣results.');
assert(all(f(NOT ind) < 1 · 10^{-10}), ...
  'These␣forces␣should␣all␣be␣(essentially)␣zero.');
fprintf('passed\n')
disp('=================')
```

79. Check that the *linear − quasi* array gives same output as *linear* array for equivalent parameters.

⟨multiforce_test003a.m 79⟩ ≡

```
disp('=================')
fprintf('TEST␣003a:␣')
displ = [0.02 0.02 0.02];
    % Test against Halbach array with four magnets per wavelength
fixed_array = struct(...
  'type', 'linear', ...
  'align', 'x', ...
  'face', 'up', ...
  'length', 0.05, ...
  'width', 0.01, ...
  'height', 0.01, ...
  'Nmag_per_wave', 4, ...
  'Nwaves', 1...
  );
float_array = fixed_array;
float_array.face = 'down';
f1 = multipoleforces(fixed_array, float_array, displ);
fixed_array = struct(...
  'type', 'linear-quasi', ...
  'align', 'x', ...
  'face', 'up', ...
  'length', 0.05, ...
  'width', 0.01, ...
  'height', 0.01, ...
  'Nwaves', 1, ...
  'ratio', 1...
  );
float_array = fixed_array;
float_array.face = 'down';
f2 = multipoleforces(fixed_array, float_array, displ);
assert(all(chop(f1, 6) ≡ chop(f2, 6)), ...
  'linear␣(4mag)␣and␣linear-quasi␣should␣be␣equivalent');
    % Test against Halbach array with two magnets per wavelength
fixed_array = struct(...
  'type', 'linear', ...
  'align', 'x', ...
  'face', 'up', ...
  'length', 0.03, ...
  'width', 0.01, ...
  'height', 0.01, ...
```

```
    'Nmag_per_wave', 2, ...
    'Nwaves', 1 ...
    );
  float_array = fixed_array;
  float_array.face = 'down';
  f3 = multipoleforces(fixed_array, float_array, displ);
  fixed_array = struct(...
    'type', 'linear-quasi', ...
    'align', 'x', ...
    'face', 'up', ...
    'length', 0.03, ...
    'width', 0.01, ...
    'height', 0.01, ...
    'Nwaves', 1, ...
    'ratio', 0 ...
    );
  float_array = fixed_array;
  float_array.face = 'down';
  f4 = multipoleforces(fixed_array, float_array, displ);
  assert(all(chop(f3, 6) ≡ chop(f4, 6)), ...
    'linear␣(2mag)␣and␣linear-quasi␣should␣be␣equivalent');
  fprintf('passed\n')
  disp('=================')
```

80. These are MATLABWEB declarations to improve the formatting of this document. Ignore unless you're editing `magnetforces.web`.

define end ≡ **end**
format *END* *TeX*

Index of `magnetforces`

List of Refinements in `magnetforces`

⟨ `magforce_test001a.m` 37 ⟩
⟨ `magforce_test001b.m` 41 ⟩
⟨ `magforce_test001c.m` 47 ⟩
⟨ `magforce_test001d.m` 51 ⟩
⟨ `magnetforces.m` 4 ⟩
⟨ `multiforce_test002a.m` 75 ⟩
⟨ `multiforce_test002b.m` 76 ⟩
⟨ `multiforce_test002c.m` 77 ⟩
⟨ `multiforce_test002d.m` 78 ⟩
⟨ `multiforce_test003a.m` 79 ⟩
⟨ `multipoleforces.m` 56 ⟩
⟨ `testall.m` 36, 74 ⟩
⟨ Array magnetisation directions 71 ⟩ Used in section 61.
⟨ Array magnetisation strengths 70 ⟩ Used in section 61.
⟨ Array sizes 69 ⟩ Used in section 61.
⟨ Assert combinations tests 50 ⟩ Used in section 47.
⟨ Assert magnetisations tests 46 ⟩ Used in sections 37 and 41.
⟨ Assert superposition 55 ⟩ Used in section 51.
⟨ Calculate array forces 59 ⟩ Used in section 56.
⟨ Calculate for each displacement 9 ⟩ Used in section 4.
⟨ Calculate stiffnesses 15 ⟩ Used in section 11.
⟨ Calculate x force 13 ⟩ Used in section 10.
⟨ Calculate y force 14 ⟩ Used in section 10.
⟨ Calculate z force 12 ⟩ Used in section 10.
⟨ Create arrays from input variables 61 ⟩ Used in section 57.
⟨ Extrapolate variables from input 72 ⟩ Used in section 57.
⟨ Fill in array structures 62 ⟩ Used in section 61.
⟨ Finish up 26 ⟩ Used in sections 17, 18, 21, and 22.
⟨ Function for resolving magnetisations 28 ⟩ Used in section 4.
⟨ Function for single force calculation 10 ⟩ Used in section 4.
⟨ Function for single stiffness calculation 11 ⟩ Used in section 4.
⟨ Functions for calculating forces and stiffnesses 16 ⟩ Used in section 4.
⟨ Helper functions 31, 32 ⟩ Used in section 16.
⟨ Infer linear array variables 64 ⟩ Used in section 61.
⟨ Infer linear-quasi array variables 65 ⟩ Used in section 61.
⟨ Infer patchwork array variables 68 ⟩ Used in section 61.
⟨ Infer planar array variables 66 ⟩ Used in section 61.
⟨ Infer quasi-Halbach array variables 67 ⟩ Used in section 61.
⟨ Initialise main variables 5, 25 ⟩ Used in section 4.
⟨ Initialise multipole variables 60 ⟩ Used in section 56.
⟨ Initialise subfunction variables 24 ⟩ Used in sections 17, 18, 21, and 22.
⟨ Matlab help text (forces) 34 ⟩ Used in section 4.
⟨ Matlab help text (multipole) 73 ⟩ Used in section 56.
⟨ Multipole sub-functions 57 ⟩ Used in section 56.

⟨ Organise input displacements 6 ⟩ Used in sections 4 and 56.
⟨ Orthogonal magnets force calculation 18, 20 ⟩ Used in section 16.
⟨ Orthogonal magnets stiffness calculation 22, 23 ⟩ Used in section 16.
⟨ Parallel magnets force calculation 17 ⟩ Used in section 16.
⟨ Parallel magnets stiffness calculation 21 ⟩ Used in section 16.
⟨ Parse calculation args 7 ⟩ Used in sections 4 and 56.
⟨ Precompute displacement rotations 30 ⟩ Used in sections 10 and 11.
⟨ Precompute rotations 29 ⟩ Used in section 4.
⟨ Print diagnostics 33 ⟩ Used in sections 10 and 11.
⟨ Return all results 8 ⟩ Used in sections 4 and 56.
⟨ Set alignment/facing directions 63 ⟩ Used in section 61.
⟨ Test x–x magnetisations 39 ⟩ Used in section 37.
⟨ Test y–y magnetisations 40 ⟩ Used in section 37.
⟨ Test z–z magnetisations 38 ⟩ Used in section 37.
⟨ Test XY superposition 52 ⟩ Used in section 51.
⟨ Test XZ superposition 53 ⟩ Used in section 51.
⟨ Test ZXX 45 ⟩ Used in section 41.
⟨ Test ZXZ 43 ⟩ Used in section 41.
⟨ Test ZYY 44 ⟩ Used in section 41.
⟨ Test ZYZ 42 ⟩ Used in section 41.
⟨ Test against Janssen results 19 ⟩
⟨ Test combinations ZY 49 ⟩ Used in section 47.
⟨ Test combinations ZZ 48 ⟩ Used in section 47.
⟨ Test planar superposition 54 ⟩ Used in section 51.

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.

[2] Jean-Paul Yonnet and Hicham Allag. “Analytical Calculation of CuboÃŕdal Magnet Interactions in 3D”. In: *The 7th International Symposium on Linear Drives for Industry Application*. 2009.