Forces between magnets and multipole arrays of magnets

Will Robertson

November 23, 2009

magnetforces

1. About this file. This is a 'literate programming' approach to writing Matlab code using MATLABWEB[1]. 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 ones work habits, this may be more or less of limiting factor to using 'literate programming' in this way.

2. 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.)

[1] http://tug.ctan.org/pkg/matlabweb

3. 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.

⟨`magnetforces.m` 3⟩ ≡

```
function [varargout] = magnetforces(magnet_fixed, magnet_float, displ, varargin)
  ⟨Matlab help text 34⟩
  ⟨Parse calculation args 8⟩
  ⟨Initialise main variables 5⟩
  ⟨Precompute rotation matrices 30⟩
  ⟨Decompose orthogonal superpositions 6⟩
  ⟨Calculate everything 11⟩
  ⟨Combine results and exit 9⟩
  ⟨Functions for calculating forces and stiffnesses 18⟩
  end
```

4. Variables and data structures.

5. 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 ($-\pi/2 \leq \phi \leq \pi/2$) and θ 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:

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

⟨ Initialise main variables 5 ⟩ ≡

```
size1 = reshape(magnet_fixed.dim/2, [3 1]);
size2 = reshape(magnet_float.dim/2, [3 1]);
J1r = magnet_fixed.magn;
J2r = magnet_float.magn;
J1t = magnet_fixed.magdir(1);
J2t = magnet_float.magdir(1);
J1p = magnet_fixed.magdir(2);
J2p = magnet_float.magdir(2);
```

See also sections 17 and 27.
This code is used in section 3.

6. Superposition is used to turn an arbitrary magnetisation angle into a set of orthogonal magnetisations.

Each magnet can potentially have three components, which can result in up to nine force calculations for a single magnet.

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 rather than symbolically).

⟨ Decompose orthogonal superpositions 6 ⟩ ≡

```
displ = reshape(displ, [3 1]);        % column vector
J1 = [J1r*cosd(J1p)*cosd(J1t); ...
  J1r*cosd(J1p)*sind(J1t); ...
  J1r*sind(J1p)];
J2 = [J2r*cosd(J2p)*cosd(J2t); ...
  J2r*cosd(J2p)*sind(J2t); ...
  J2r*sind(J2p)];
```

This code is used in section 3.

7. Wrangling user input and output.

8. We now have a choice of calculations to take based on the user input. Take the opportunity to bail out in case the user has requested more calculations than provided as outputs to the function.

⟨ Parse calculation args 8 ⟩ ≡

```
Nvargin = length(varargin);
if ( Nvargin ≠ 0 ∧ ∧ Nvargin ≠ nargout )
error('Must␣have␣as␣many␣outputs␣as␣calculations␣requested.')
end
calc_force_bool = false;
calc_stiffness_bool = false;
if Nvargin ≡ 0
  calc_force_bool = true;
else
  for ii = 1 : Nvargin
    switch varargin{ii}
    case 'force'
      calc_force_bool = true;
    case 'stiffness'
      calc_stiffness_bool = true;
    otherwise
      error(['Unknown␣calculation␣option␣''', varargin{ii}, ''''])
    end
  end
end
```

This code is used in section 3.

9. After all of the calculations have occured, they're placed back into `varargout`.

⟨ Combine results and exit 9 ⟩ ≡

```
if Nvargin ≡ 0
  varargout{1} = forces_out;
else
  for ii = 1 : Nvargin
    switch varargin{ii}
    case 'force'
      varargout{ii} = forces_out;
    case 'stiffness'
      varargout{ii} = stiffnesses_out;
    end
  end
end
```

This code is used in section 3.

10. The actual mechanics.

11. The expressions we have to calculate the forces assume a fixed magnet with positive z magnetisation only. Secondly, magnetisation direction of the floating magnet may only be in the positive z- or y-directions.

The parallel forces are more easily visualised; if *J1z* is negative, then transform the coordinate system so that up is down and down is up. Then proceed as usual and reverse the vertical forces in the last step.

The orthogonal forces require reflection and/or rotation to get the displacements in a form suitable for calculation.

Initialise a 9×3 array to store each force component in each direction, and then fill 'er up.

⟨ Calculate everything 11 ⟩ ≡

```
⟨ Print diagnostics 12 ⟩
⟨ Calculate x 14 ⟩
⟨ Calculate y 15 ⟩
⟨ Calculate z 13 ⟩
⟨ Combine calculations 16 ⟩
```

This code is used in section 3.

12. 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 12 ⟩ ≡

```
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 section 11.

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

⟨ Calculate z 13 ⟩ ≡

```
if calc_force_bool
  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);
end
if calc_stiffness_bool
  debug_disp('z-z␣stiffness:')
  stiffness_components(9, :) = stiffnesses_calc_z_z(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-x␣stiffness:')
  stiffness_components(7, :) = stiffnesses_calc_z_x(size1, size2, displ, J1,
      J2);
end
```

This code is used in section 11.

14. 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 14⟩ ≡

```
size1_rot = swap_x_z(size1);
size2_rot = swap_x_z(size2);
d_rot = rotate_x_to_z(displ);
J1_rot = rotate_x_to_z(J1);
J2_rot = rotate_x_to_z(J2);
if calc_force_bool
  debug_disp('Forces x-x:')
  forces_x_x = forces_calc_z_z(size1_rot, size2_rot, d_rot, J1_rot, J2_rot);
  force_components(1, :) = rotate_z_to_x(forces_x_x);
  debug_disp('Forces x-y:')
  forces_x_y = forces_calc_z_y(size1_rot, size2_rot, d_rot, J1_rot, J2_rot);
  force_components(2, :) = rotate_z_to_x(forces_x_y);
  debug_disp('Forces x-z:')
  forces_x_z = forces_calc_z_y(size1_rot, size2_rot, d_rot, J1_rot, J2_rot);
  force_components(3, :) = rotate_z_to_x(forces_x_z);
end
if calc_stiffness_bool
  debug_disp('x-z stiffness:')
  stiffness_components(3, :) = rotate_z_to_x(stiffnesses_calc_z_x(size1_rot,
      size2_rot, d_rot, J1_rot, J2_rot));
  debug_disp('x-y stiffness:')
  stiffness_components(2, :) = rotate_z_to_x(stiffnesses_calc_z_y(size1_rot,
      size2_rot, d_rot, J1_rot, J2_rot));
  debug_disp('x-x stiffness:')
  stiffness_components(1, :) = rotate_z_to_x(stiffnesses_calc_z_z(size1_rot,
      size2_rot, d_rot, J1_rot, J2_rot));
end
```

This code is used in section 11.

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

⟨ Calculate y 15 ⟩ ≡

```
size1_rot = swap_y_z(size1);
size2_rot = swap_y_z(size2);
d_rot = rotate_y_to_z(displ);
J1_rot = rotate_y_to_z(J1);
J2_rot = rotate_y_to_z(J2);
if calc_force_bool
  debug_disp('Forces y-x:')
  forces_y_x = forces_calc_z_x(size1_rot, size2_rot, d_rot, J1_rot, J2_rot);
  force_components(4, :) = rotate_z_to_y(forces_y_x);
  debug_disp('Forces y-y:')
  forces_y_y = forces_calc_z_z(size1_rot, size2_rot, d_rot, J1_rot, J2_rot);
  force_components(5, :) = rotate_z_to_y(forces_y_y);
  debug_disp('Forces y-z:')
  forces_y_z = forces_calc_z_y(size1_rot, size2_rot, d_rot, J1_rot, J2_rot);
  force_components(6, :) = rotate_z_to_y(forces_y_z);
end
if calc_stiffness_bool
  debug_disp('y-z stiffness:')
  stiffness_components(6, :) = rotate_z_to_y(stiffnesses_calc_z_y(size1_rot,
      size2_rot, d_rot, J1_rot, J2_rot));
  debug_disp('y-y stiffness:')
  stiffness_components(5, :) = rotate_z_to_y(stiffnesses_calc_z_z(size1_rot,
      size2_rot, d_rot, J1_rot, J2_rot));
  debug_disp('y-x stiffness:')
  stiffness_components(4, :) = rotate_z_to_y(stiffnesses_calc_z_x(size1_rot,
      size2_rot, d_rot, J1_rot, J2_rot));
end
```

This code is used in section 11.

16. Finally sum all the components in each direction to get the total forces.

⟨ Combine calculations 16 ⟩ ≡

```
if calc_force_bool
  forces_out = sum(force_components);
end
if calc_stiffness_bool
  stiffnesses_out = sum(stiffness_components);
end
```

This code is used in section 11.

17. You might have noticed that the initialisation of the *force_components* (and other) variables has not yet been listed. That's because the code is boring.

⟨ Initialise main variables 5 ⟩ +≡

```
if calc_force_bool
  force_components = repmat(NaN, [9 3]);
end
if calc_stiffness_bool
  stiffness_components = repmat(NaN, [9 3]);
end
```

18. 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.

$\langle$ Functions for calculating forces and stiffnesses 18 $\rangle \equiv$

$\langle$ Parallel magnets force calculation 19 $\rangle$
$\langle$ Orthogonal magnets force calculation 20 $\rangle$
$\langle$ Parallel magnets stiffness calculation 23 $\rangle$
$\langle$ Orthogonal magnets stiffness calculation 24 $\rangle$
$\langle$ Helper functions 31 $\rangle$

This code is used in section 3.

19. 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

$\langle$ Parallel magnets force calculation 19 $\rangle \equiv$

```
function calc_out = forces_calc_z_z(size1, size2, offset, J1, J2)
    J1 = J1(3);
    J2 = J2(3);
    ⟨ Initialise subfunction variables 26 ⟩
    component_x = ...
    +0.5*(v .^ 2 - w .^ 2) .* log(r - u) ...
    +u .* v .* log(r - v) ...
    +v .* w .* atan2(u .* v, r .* w) ...
    +0.5*r .* u;
    component_y = ...
    +0.5*(u .^ 2 - w .^ 2) .* log(r - v) ...
    +u .* v .* log(r - u) ...
    +u .* w .* atan2(u .* v, r .* w) ...
    +0.5*r .* v;
    component_z = ...
    -u .* w .* log(r - u) ...
    -v .* w .* log(r - v) ...
    +u .* v .* atan2(u .* v, r .* w) ...
    -r .* w;
    ⟨ Finish up 28 ⟩
```

This code is used in section 18.

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

$\langle$ Orthogonal magnets force calculation 20 $\rangle \equiv$

```
function calc_out = forces_calc_z_y(size1, size2, offset, J1, J2)
    J1 = J1(3);
    J2 = J2(2);
    ⟨ Initialise subfunction variables 26 ⟩
    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_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_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;
    allag_correction = -1;
    component_x = allag_correction*component_x;
    component_y = allag_correction*component_y;
    component_z = allag_correction*component_z;
    if 0
        ⟨ Test against Janssen results 21 ⟩
    end
    ⟨ Finish up 28 ⟩
```

See also section 22.

This code is used in section 18.

21. This is the same calculation with Janssen's equations instead.

$\langle$ Test against Janssen results 21 $\rangle \equiv$

```
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)...
;
if 0
   xx = component_x(:);
   xx_ii = component_x_ii(:);
   assert(all(xx ≡ xx_ii))
end
if 0
   yy = component_y(:);
   yy_ii = component_y_ii(:);
   assert(all(abs(abs(yy) - abs(yy_ii)) < 1 · 10^-4))
end
if 0
   zz = component_z(:);
   zz_ii = component_z_ii(:);
   assert(all(abs(abs(zz) - abs(zz_ii)) < 1 · 10^-4))
end
```

```
if 1
  component_x = component_x_ii;
  component_y = component_y_ii;
  component_z = component_z_ii;
end
```

This code is used in section 20.

22. Don't need to swap $J1$ because it should only contain z components anyway. (This is assumption isn't tested because it it's wrong we're in more trouble anyway; this should all be taken care of earlier when the magnetisation components were separated out.)

⟨ Orthogonal magnets force calculation 20 ⟩ +≡

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

23. Stiffness calculations are derived[2] from the forces.

⟨ Parallel magnets stiffness calculation 23 ⟩ ≡

```
function calc_out = stiffnesses_calc_z_z(size1, size2, offset, J1, J2)
    J1 = J1(3);
    J2 = J2(3);
    ⟨ Initialise subfunction variables 26 ⟩
    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 28 ⟩
```

This code is used in section 18.

---

[2] Literally.

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

⟨ Orthogonal magnets stiffness calculation 24 ⟩ ≡

function $calc\_out = stiffnesses\_calc\_z\_y(size1, size2, offset, J1, J2)$

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

⟨ Initialise subfunction variables 26 ⟩

$component\_x = -((u.\hat{}2.*v)./(u.\hat{}2+v.\hat{}2)) - (u.\hat{}2.*w)./(u.\hat{}2+w.\hat{}2)\ldots$
$+u .* atan1(v .* w, r .* u) - multiply\_x\_log\_y(w, r + v) + \ldots$
$-multiply\_x\_log\_y(v, r + w);$

$component\_y = v/2 - (u.\hat{}2.*v)./(u.\hat{}2+v.\hat{}2) + (u.*v.*w)./(v.\hat{}2+w.\hat{}2)\ldots$
$+u .* atan1(u .* w, r .* v) + multiply\_x\_log\_y(v, r + w);$

$component\_z = -component\_x - component\_y;$

$allag\_correction = -1;$
$component\_x = allag\_correction * component\_x;$
$component\_y = allag\_correction * component\_y;$
$component\_z = allag\_correction * component\_z;$

⟨ Finish up 28 ⟩

See also section 25.
This code is used in section 18.

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

⟨ Orthogonal magnets stiffness calculation 24 ⟩ +≡

function $calc\_out = stiffnesses\_calc\_z\_x(size1, size2, offset, J1, J2)$

$stiffnesses\_xyz = stiffnesses\_calc\_z\_y(\ldots$
$rotate\_x\_to\_y(size1), rotate\_x\_to\_y(size2), rotate\_x\_to\_y(offset), \ldots$
$J1, rotate\_x\_to\_y(J2));$

$calc\_out = rotate\_y\_to\_x(stiffnesses\_xyz);$

end

26. 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 26 ⟩ ≡

```
if ( J1 ≡ 0 OR 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);
```

This code is used in sections 19, 20, 23, and 24.

27. Here are some variables used above that only needs to be computed once (and is slow). The idea here is to vectorise instead of use **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]);
[index2_j, index2_l, index2_q] = ndgrid([0 1]);
index_sum = (−1).^(index_i+index_j+index_k+index_l+index_p+index_q);
```

28. And some shared finishing code.

⟨ Finish up 28 ⟩ ≡

```
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 19, 20, 23, and 24.

29. Setup code.

30. When the forces are rotated we use these rotation matrices to avoid having to think too hard. Use degrees in order to compute $\sin(\pi/2)$ exactly!

⟨Precompute rotation matrices 30⟩ ≡

```
swap_x_y =@(vec) vec([2 1 3]);
swap_x_z =@(vec) vec([3 2 1]);
swap_y_z =@(vec) vec([1 3 2]);
Rx =@(θ) [1 0 0; 0 cosd(θ) − sind(θ); 0 sind(θ) cosd(θ)];
Ry =@(θ) [cosd(θ) 0 sind(θ); 0 1 0; −sind(θ) 0 cosd(θ)];
Rz =@(θ) [cosd(θ) − sind(θ) 0; sind(θ) cosd(θ) 0; 0 0 1];
Rx_180 = Rx(180);
Rx_090 = Rx(90);
Rx_270 = Rx(−90);
Ry_180 = Ry(180);
Ry_090 = Ry(90);
Ry_270 = Ry(−90);
Rz_180 = Rz(180);
Rz_090 = Rz(90);
Rz_270 = Rz(−90);
identity_function =@(inp) inp;
rotate_round_x =@(vec) Rx_180∗vec;
rotate_round_y =@(vec) Ry_180∗vec;
rotate_round_z =@(vec) Rz_180∗vec;
rotate_none = identity_function;
rotate_z_to_x =@(vec) Ry_090∗vec;
rotate_x_to_z =@(vec) Ry_270∗vec;
rotate_z_to_y =@(vec) Rx_090∗vec;
rotate_y_to_z =@(vec) Rx_270∗vec;
rotate_x_to_y =@(vec) Rz_090∗vec;
rotate_y_to_x =@(vec) Rz_270∗vec;
```

This code is used in section 3.

31. The equations contain some odd 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.

⟨ Helper functions 31 ⟩ ≡

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

See also sections 32 and 33.

This code is used in section 18.

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. This function is for easy debugging; in normal use it gobbles its argument but will print diagnostics when required.

⟨ Helper functions 31 ⟩ +≡

```
function debug_disp(str)
      %disp(str)
  end
```

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

⟨ Matlab help text 34 ⟩ ≡

```
%% MAGNETFORCES  Calculate forces between two cuboid magnets
%
% Finish this off later.
%
```

This code is used in section 3.

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;
unix('~/bin/mtangle magnetforces');
magforce_test001a
magforce_test001b
magforce_test001c
magforce_test001d
```

37. Force testing.

38. 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 38⟩ ≡

```
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 39⟩
⟨Assert magnetisations tests 47⟩
⟨Test x–x magnetisations 40⟩
⟨Assert magnetisations tests 47⟩
⟨Test y–y magnetisations 41⟩
⟨Assert magnetisations tests 47⟩
fprintf('passed\n')
disp('=================')
```

39. Testing vertical forces.

⟨Test z–z magnetisations 39⟩ ≡

```
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 38.

40. Testing horizontal x forces.

⟨ Test x–x magnetisations 40 ⟩ ≡

```
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 38.

41. Testing horizontal y forces.

⟨ Test y–y magnetisations 41 ⟩ ≡

```
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 38.

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

⟨ magforce_test001b.m 42 ⟩ ≡

```
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 43 ⟩
⟨ Assert magnetisations tests 47 ⟩
⟨ Test ZXZ 44 ⟩
⟨ Assert magnetisations tests 47 ⟩
⟨ Test ZXX 46 ⟩
⟨ Assert magnetisations tests 47 ⟩
⟨ Test ZYY 45 ⟩
⟨ Assert magnetisations tests 47 ⟩
fprintf('passed\n')
disp('=================')
```

43. z–y magnetisations, z displacement.

⟨ Test ZYZ 43 ⟩ ≡

```
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 42.

44. z–x magnetisations, z displacement.

⟨ Test ZXZ 44 ⟩ ≡

```
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 42.

45. z–y magnetisations, y displacement.

⟨ Test ZYY 45 ⟩ ≡

```
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 42.

46. z–x magnetisations, x displacement.

⟨ Test ZXX 46 ⟩ ≡

```
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 42.

47. The assertions, common between directions.

⟨ Assert magnetisations tests 47 ⟩ ≡

```
assert(...
  all(abs(otherforces(:)) < 1e-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 38 and 42.

48. Now try combinations of displacements.

⟨ magforce_test001c.m 48 ⟩ ≡

```
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 49 ⟩
⟨ Assert combinations tests 51 ⟩
⟨ Test combinations ZY 50 ⟩
⟨ Assert combinations tests 51 ⟩
fprintf('passed\n')
disp('=================')
```

49. Tests.

⟨ Test combinations ZZ 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 = [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 48.

50. Tests.

⟨ Test combinations ZY 50 ⟩ ≡

```
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 48.

51. Shared tests, again.

⟨ Assert combinations tests 51 ⟩ ≡

```
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 48.

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

⟨ magforce_test001d.m 52 ⟩ ≡

```
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 53 ⟩
⟨ Assert superposition 56 ⟩
⟨ Test XZ superposition 54 ⟩
⟨ Assert superposition 56 ⟩
⟨ Test planar superposition 55 ⟩
⟨ Assert superposition 56 ⟩
fprintf('passed\n')
disp('=================')
```

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

⟨ Test XY superposition 53 ⟩ ≡

```
magnet_float.magdir = [45 0];        % e_x + e_y
f1 = magnetforces(magnet_fixed, magnet_float, displ);
    % Components:
magnet_float.magdir = [0 0];         % e_x
fc1 = magnetforces(magnet_fixed, magnet_float, displ);
magnet_float.magdir = [90 0];        % e_y
fc2 = magnetforces(magnet_fixed, magnet_float, displ);
f2 = (fc1 + fc2)/sqrt(2);
```

This code is used in section 52.

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

⟨ Test XZ superposition 54 ⟩ ≡

```
magnet_float.magdir = [0 45];       % e_y + e_z
f1 = magnetforces(magnet_fixed, magnet_float, displ);
        % Components:
magnet_float.magdir = [0 0];        % e_x
fc1 = magnetforces(magnet_fixed, magnet_float, displ);
magnet_float.magdir = [0 90];       % e_z
fc2 = magnetforces(magnet_fixed, magnet_float, displ);
f2 = (fc1 + fc2)/sqrt(2);
```

This code is used in section 52.

55. 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 55 ⟩ ≡

```
[t p r] = cart2sph(1/sqrt(3), 1/sqrt(3), 1/sqrt(3));
magnet_float.magdir = [t p]*180/π;  % e_y + e_z + e_z
f1 = magnetforces(magnet_fixed, magnet_float, displ);
        % Components:
magnet_float.magdir = [0 0];        % e_x
fc1 = magnetforces(magnet_fixed, magnet_float, displ);
magnet_float.magdir = [90 0];       % e_y
fc2 = magnetforces(magnet_fixed, magnet_float, displ);
magnet_float.magdir = [0 90];       % e_z
fc3 = magnetforces(magnet_fixed, magnet_float, displ);
f2 = (fc1 + fc2 + fc3)/sqrt(3);
```

This code is used in section 52.

56. The assertion is the same each time.

⟨ Assert superposition 56 ⟩ ≡

```
assert(...
  isequal(chop(f1, 5), chop(f2, 5)), ...
  'Components␣should␣sum␣due␣to␣superposition' ...
  )
```

This code is used in section 52.

57. 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

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. See p. 11.

[2] Jean-Paul Yonnet and Hicham Allag. "Analytical Calculation of Cubodal Magnet Interactions in 3D". In: *The 7th International Symposium on Linear Drives for Industry Application*. 2009. See pp. 12, 15.