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 to some code to calculate the forces (and perhaps torques) between two cuboid-shaped magnets with arbitary displacement and magnetisation direction.

If this code works then I'll look at calculating the forces for magnets with rotation as well.

3. The main function is called *magnetforces*, which takes three arguments: *magnet_fixed*, *magnet_float*, and *displ*. These will be described below.

⟨`magnetforces.m` 3⟩ ≡

```
function [forces_out] = magnetforces(magnet_fixed, magnet_float, displ)
  ⟨ Matlab help text 20 ⟩
  ⟨ Extract input variables 4 ⟩
  ⟨ Precompute rotation matrices 19 ⟩
  ⟨ Decompose orthogonal superpositions 5 ⟩
  ⟨ Calculate all forces 6 ⟩
  ⟨ Functions for calculating forces and stiffnesses 11 ⟩
  end
```

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

4. 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.

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:

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

⟨ Extract input variables 4 ⟩ ≡

```
a1 = 0.5*magnet_fixed.dim(1);
b1 = 0.5*magnet_fixed.dim(2);
c1 = 0.5*magnet_fixed.dim(3);
size1 = [a1;  b1;  c1];
a2 = 0.5*magnet_float.dim(1);
b2 = 0.5*magnet_float.dim(2);
c2 = 0.5*magnet_float.dim(3);
size2 = [a2;  b2;  c2];
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);
if ( J1r < 0 OR J2r < 0 )
error('By␣convention,␣magnetisation␣must␣be␣positive;␣change␣the␣angle␣to␣reverse␣direction
end
```

This code is used in section 3.

5. 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 5 ⟩ ≡

```
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.

6. 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 $3 \times 3 \times 3$ array to store each force component in each direction, and fill it up by calculating

⟨ Calculate all forces 6 ⟩ ≡

```
force_components = repmat(NaN, [9 3]);
⟨ Print diagnostics 7 ⟩
⟨ Calculate forces x 9 ⟩
⟨ Calculate forces y 10 ⟩
⟨ Calculate forces z 8 ⟩
forces_out = sum(force_components);
```

This code is used in section 3.

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

⟨ Print diagnostics 7 ⟩ ≡

```
disp('␣␣')
disp('CALCULATING␣FORCES')
disp('==================')
disp('Displacement:')
disp(displ')
disp('Magnetisations:')
disp(J1')
disp(J2')
```

This code is used in section 6.

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

⟨ Calculate forces z 8 ⟩ ≡

```
disp('Forces␣z-z:')
forces_z_z = forces_calc_z_z(size1, size2, displ, J1, J2);
force_components(7, :) = forces_z_z;
disp('Forces␣z-y:')
forces_z_y = forces_calc_z_y(size1, size2, displ, J1, J2);
force_components(8, :) = forces_z_y;
disp('Forces␣z-x:')
forces_z_x = forces_calc_z_x(size1, size2, displ, J1, J2);
force_components(9, :) = forces_z_x;
```

This code is used in section 6.

9. 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 forces x 9⟩ ≡

```
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);
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);
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);
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);
```

This code is used in section 6.

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

⟨Calculate forces y 10⟩ ≡

```
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);
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);
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);
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);
```

This code is used in section 6.

11. 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 11 $\rangle \equiv$

  $\langle$ Parallel magnets force calculation 12 $\rangle$
  $\langle$ Orthogonal magnets force calculation 14 $\rangle$

This code is used in section 3.

12. 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 12 ⟩ ≡

```
function forces_xyz = forces_calc_z_z(size1, size2, offset, J1, J2)
    % You probably want to call
    %   warning off MATLAB:divideByZero
    %   warning off MATLAB:log:logOfZero
  J1 = J1(3);
  J2 = J2(3);
  if ( J1 ≡ 0 OR J2 ≡ 0 )
  disp('Zero magnetisation.')
  forces_xyz = [0; 0; 0];
  return;
  end
  ⟨ Forces initialise variables 17 ⟩
  f_x = ...
  +0.5*(v .^ 2 − w .^ 2) .* log(r − u) ...
  +u .* v .* log(r − v) ...
  +v .* w .* atan(u .* v ./ r ./ w) ...
  +0.5*r .* u;
  f_y = ...
  +0.5*(u .^ 2 − w .^ 2) .* log(r − v) ...
  +u .* v .* log(r − u) ...
  +u .* w .* atan(u .* v ./ r ./ w) ...
  +0.5*r .* v;
  f_z = ...
  −u .* w .* log(r − u) ...
  −v .* w .* log(r − v) ...
  +u .* v .* atan(u .* v ./ r ./ w) ...
  −r .* w;
  fx = index_sum .* f_x;
  fy = index_sum .* f_y;
  fz = index_sum .* f_z;
  magconst = J1*J2/(4*π*(4*π*1 · 10^−7));
  forces_xyz = magconst .* [sum(fx(:)); sum(fy(:)); sum(fz(:))];
  disp(forces_xyz′)
  end
```

This code is used in section 11.

13. Orthogonal magnets forces given by Yonnet and Allag [2]. Magnetisation of the fixed magnet *J1* is in the positive z-direction, and the magnetisation of the floating magnet *J2* is in the positive y-direction. This means we need to perform coordinate system transformation if either or both of *J1* and *J2* are negative.

I don't use a general solution here because there's only a small, fixed number of possibilities. The general solution is more calculatorily complex.

14. There is a singularity at a displacement of $(0, 0, \pm z)$.

⟨ Orthogonal magnets force calculation 14 ⟩ ≡

```
function forces_xyz = forces_calc_z_y(size1, size2, offset, J1, J2)
    J1m = J1(3);
    J2m = J2(2);
    if ( J1m ≡ 0 OR J2m ≡ 0 )
    disp('Zero␣magnetisation.')
    forces_xyz = [0; 0; 0];
    return;
    end
    if ( J1m > 0 ∧ ∧ J2m > 0 )
    rotate_transform = rotate_none;
    elseif ( J1m < 0 ∧ ∧ J2m > 0 )
    rotate_transform = rotate_round_y;
    elseif ( J1m > 0 ∧ ∧ J2m < 0 )
    rotate_transform = rotate_round_z;
    elseif ( J1m < 0 ∧ ∧ J2m < 0 )
    rotate_transform = rotate_round_x;
    end
    forces_tmp = forces_calc_z_y_plusplus(...
      size1, size2, ...
      rotate_transform(offset), ...
      rotate_transform(J1), ...
      rotate_transform(J2) ...
      );
    forces_xyz = rotate_transform(forces_tmp);
    disp(forces_xyz')
    end
```

See also sections 15 and 16.

This code is used in section 11.

15. Don't bother with rotation matrices for the z–x case; just reflect the coordinate system by swapping the components.

⟨ Orthogonal magnets force calculation 14 ⟩ +≡

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

16. This is what it all boils down to.

⟨ Orthogonal magnets force calculation 14 ⟩ +≡

```
function forces_xyz = forces_calc_z_y_plusplus(size1, size2, offset, J1, J2)
    J1 = J1(3);
    J2 = J2(2);
    disp('Offset')
    disp(offset')
    disp('Magnetisations')
    disp([J1 J2])
    ⟨ Forces initialise variables 17 ⟩
    if ( J1 < 0 OR J2 < 0 )
    error('Positive␣magnetisations␣only!')
    end
    f_x = ...
    -v .* w .* log(r - u) ...
    +v .* u .* log(r + w) ...
    +u .* w .* log(r + v) ...
    -0.5*u .^ 2 .* atan(v .* w ./ (u .* r)) ...
    -0.5*v .^ 2 .* atan(u .* w ./ (v .* r)) ...
    -0.5*w .^ 2 .* atan(u .* v ./ (w .* r));
    f_y = ...
    0.5*(u .^ 2 - v .^ 2) .* log(r + w) ...
    -u .* w .* log(r - u) ...
    -u .* v .* atan(u .* w ./ (v .* r)) ...
    -0.5*w .* r;
    f_z = ...
    0.5*(u .^ 2 - w .^ 2) .* log(r + v) ...
    -u .* v .* log(r - u) ...
    -u .* w .* atan(u .* v ./ (w .* r)) ...
    -0.5*v .* r;
    fx = index_sum .* f_x;
    fy = index_sum .* f_y;
    fz = index_sum .* f_z;
    magconst = J1*J2/(4*π*(4*π*1·10^-7));
    forces_xyz = magconst .* [sum(fx(:)); sum(fy(:)); sum(fz(:))];
    end
```

17. 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 'chopped'; i.e., that we don't need to check for *J1* or *J2* less than $1 \cdot 10^{-12}$ or whatever.

(I'm using the Mathematica definition of *chop* here; in Matlab it means truncate to a certain number of significant figures.)

⟨ Forces initialise variables 17 ⟩ ≡

```
dx = offset(1);
dy = offset(2);
dz = offset(3);
a = size1(1);
b = size1(2);
c = size1(3);
A = size2(1);
B = size2(2);
C = size2(3);
[index_h, index_j, index_k, index_l, index_p, index_q] = ndgrid([0 1]);
index_sum = (-1) .^ (index_h + index_j + index_k + index_l + index_p +
    index_q);
    % (Using this vectorised method is actually less efficient than using six for
    % loops over [0, 1]. To be addressed.)
u = dx + A*(-1) .^ index_j - a*(-1) .^ index_h;
v = dy + B*(-1) .^ index_l - b*(-1) .^ index_k;
w = dz + C*(-1) .^ index_q - c*(-1) .^ index_p;
r = sqrt(u .^ 2 + v .^ 2 + w .^ 2);
```

This code is used in sections 12 and 16.

18. Setup code.

19. 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 19 ⟩ ≡

```
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);
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;
```

This code is used in section 3.

20. When users type `help magnetforces` this is what they see. This is designed to be displayed in a fixed-width font so the output here will be fairly ugly.

⟨ Matlab help text 20 ⟩ ≡

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

This code is used in section 3.

21. 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.

22. Because I'm lazy, just run the tests manually for now:

⟨ `testall.m` 22 ⟩ ≡

```
try
  dbquit
end
! NOT / bin/mtangle magnetforces
magforce_test001a
magforce_test001b
```

23. 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` 23 ⟩ ≡

```
clc;
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 24 ⟩
⟨ Assert parallel magnetisations tests 27 ⟩
⟨ Test x–x magnetisations 25 ⟩
⟨ Assert parallel magnetisations tests 27 ⟩
⟨ Test y–y magnetisations 26 ⟩
⟨ Assert parallel magnetisations tests 27 ⟩
disp('============')
disp('Tests␣passed')
disp('============')
```

24. Testing vertical forces.

⟨ Test z–z magnetisations 24 ⟩ ≡

```
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 23.

25. Testing horizontal x forces.

⟨ Test x–x magnetisations 25 ⟩ ≡

```
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 23.

26. Testing horizontal y forces.

⟨ Test y–y magnetisations 26 ⟩ ≡

```
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 23.

27. The checks, common between directions. Use the subjunctive case to both describe the assertions and to output the failed tests.

⟨ Assert parallel magnetisations tests 27 ⟩ ≡

```
assert(...
  all(all(abs(otherforces) < 1·10^-11)), ...
  'Horizontal 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 section 23.

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

⟨`magforce_test001b.m` 28⟩ ≡

```
clc;
f = [];
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;
magnet_fixed.magdir = [0 90];        % z
for ii = [1, -1]
  for jj = [1, -1]
    magnet_float.magdir = ii*[90 0];        % ±y
    displ = jj*[1·10^-12 1·10^-12 0.1];        % ±z
    f( : , end +1 ) = magnetforces(magnet_fixed, magnet_float, displ);
    pause
    end
    end
        % chop:
    f(abs(f) < 1·10^-10) = 0;
```

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`.

[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.