Appendix B

Test-Particle Simulation, Distribution Building, and Growth-Rate Calculation Codes

Note that most of the simulation and growth rate Matlab codes were initially tested and debugged on a Core i5-4590S (4-core, 3 GHz), with 32 GB of RAM. So, not a powerhouse FLOPS-wise, but the code is pretty carefree regarding memory usage. Caveat emptor if you run this on a system with less RAM—Matlab may cry out, and forfeit.

B.1 Mirror Shards

Below are the primary test particle codes as used in this thesis. Historically, the mirror code was a single particle code provided by Dr. Wayne Scales. When moving towards parallelism, it was rewritten to be monolithic, and entirely run via the Matlab Distributed Compute Server (DCS) on GPU nodes. Then the computation was found to be entirely FLOPS-dependent, and running one or only a few particles on a single fast CPU core to be preferable to GPU massive parallelism. This requires as many CPU cores as possible, far beyond the artificially limited (due to Mathworks' prohibitive licensing fees) DCS max core number.

So, the decision was made to break the code apart into separate pieces, each of which would run be queued and run as a separate job on a PBS/TORQUE cluster compute system. The 'mirror shard' codes are so-named because they break the mirror simulation particles up among some number of 'shards', with each shard assigned a set number of calculation cores. A given core can work on one or many particles, and one or multiple shards can run on a given node—whatever makes for the best queuing setup.

The three primary codes are the distribute code, the Alice code, and the gather code.

B.1.1 Distribute

This code provides a Matlab function, `mirror_shards_distribute(n_run, n_shards)`, which generates the 'test distribution' with a given range of positions, velocities (given in eV), pitch angles, and azimuthal angles, and breaks it up among the specified number of shards. It splits the distribution up among a set of files of form `mshard-r<n_run>-<i>of<n_shards>-input.mat`, and also saves all pertinent distribution input parameters in the file '`mshards-r<n_run>-master.mat`'.

It does the splitting in an excessively lazy manner, using Matlab's built-in `distributed()` function over the Distributed Compute Server (DCS). This requires that a core for each shard, i.e. the maximum number of shards possible is equal to the maximum number of cluster cores available for use with the DCS.

This splitting is quite frankly a holdover from the rushed transition from a monolithic design to the sharded design. With some work this code could be done away with entirely—the inline distribution-building function is very fast, so it could be done within each individual job, based on the input parameters given and the job's ID number. Then the only remaining function of this code would be to build the 'master' file used to save the input parameters.

```
function ret = mirror_shards_distribute(n_run, n_shards)
% mirror_shards_distribute()
%
% Breaks a data array down into a number of shards, for use on single-node
% local worker pools, so we don't have to deal with Matlab DCE
% limitations on cores.
% Feed it a run number, and the number of shards to break the data over.
% The fundamentals of the simulation are all set here.

    q = 1;
    m = 1;
    nt = 10000;   % # timesteps
    dt = .01;     % step length
    qE = 0;
    qmt2 = q/m*dt/2;

    B0 = 50e-6; % Magnetic field base is 50 uT
    v0 = 0.00989179273; % likewise velocity base
                            % in PSL is equivalent to 25 eV
    r0 = 0.337212985; % based on Larmour radius w/ above,
                           % length base is ~0.337 m
    t0 = 7.14477319e-7; % based on B, Larmour period ~714 ns in s

    target_length = 5000;   % in km
    target_z = -target_length*1000/r0;  % negative because we're
                                            % launching upwards
    long_enough = 1000000000;
    mirror_ratio = 5;
    saved_steps = 1000;
```

```
30
31      % So Bsim=Breal/50uT, vsim=vreal/25 eV, and xsim=xreal/0.337m
32      % So a 100x100x1000 simulation extent is a 33.7x33.7x337m volume
33      % So dt ~71.4ns, and 1000 timesteps is 71us
34
35      % assumes 'end point' is z=0
36      length_factor = target_z^2/(mirror_ratio-1);
37
38      x_range = 0;
39      y_range = 0;
40      z_range = target_z;
41      v_range = [ 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, ...
42          256, 289, 324, 361, 400, 441, 484, 529, 576, 625, 676, 729, ...
43          784, 841, 900, 961, 1024, 1089, 1156, 1225 ]; % linear in v
44      t_dphi = 3*pi/256; % delta for co-latitude
45      t_domega = 0.001; % delta for solid angle in steradians
46      %p_range = 0:pi/7:pi; %0:pi/15:pi/2;
47
48      v_distrib = build_distrib(v0, x_range, y_range, z_range, v_range,
            t_dphi, t_domega);
49
50      % Re-run particles that failed due to max timestep limit in Run 4
51      load tzind.mat t_zind
52      v_distrib = v_distrib(:,t_zind)
53
54      parpool('torque_4nodes',n_shards)
55
56      N_part = size(v_distrib,2);
57      disp([ 'Distributing ' num2str(N_part) ' particles over '
            num2str(n_shards) ' node shards...' ])
58
59      v_sdivdist = distributed(v_distrib);
60
61      disp('Start')
62      tic
63
64      % spmd (single program, multiple data) is a more generalized
65      % multithreaded methodology than parfor, and allows use of
66      % distributed/codistributed functionality to split up arrays
67      spmd
68
69          v_localdist = getLocalPart(v_sdivdist);
70          N_dpart = size(v_localdist, 2);
71          chunk_inds = globalIndices(v_sdivdist,2);
72
73          disp( [ 'Shard ' num2str(labindex) ': ' num2str(N_dpart) ' particles
                ('  num2str(chunk_inds(1)) ':' num2str(chunk_inds(end)) ').' ] )
```

```
74
75          % Have to call a function to use save inside an spmd.
76          % Because...raisins.
77          % Local workspace memory separation something something.
78          save_mah_data_plz(n_run, labindex, n_shards, N_dpart, chunk_inds,
                v_localdist);
79
80      end
81
82      toc
83      disp('Done.')
84
85      save(['mshards-r' num2str(n_run) '-master.mat'], ...
86          'n_run', 'n_shards', 'N_part', 'v_distrib', ...
87          'q', 'm', 'nt', 'dt', 'qE', 'qmt2', ...
88          'B0', 'v0', 'r0', 't0', 'target_length', 'target_z', ...
89          'long_enough', 'length_factor', 'mirror_ratio', ...
90          'saved_steps', 'v_range', 't_dphi');
91
92      ret = 0;
93
94  end
95
96  function save_mah_data_plz(n_run, labindex, n_shards, N_shardpart,
        chunk_inds, v_sharddist)
97      % Just a function to save data.  Raisins.
98
99      save(['mshard-r' num2str(n_run) '-' num2str(labindex) 'of'
            num2str(n_shards) '-input.mat'], ...
100         'n_run', 'N_shardpart', 'chunk_inds', 'v_sharddist');
101
102 end
103
104 function d = build_distrib(v0, x_range, y_range, z_range, v_range, t_dphi,
        t_domega)
105     % Build particle distribution
106
107     % initial positions x y z
108     % initial velocities v theta phi (magnitude, azimuth, co-latitude)
109     %   mag 25:2000 eV, azi 0, el 0:pi/2
110     % input as [ x y z v theta phi ] columns in v_distrib_raw
111
112     t_phis = 0+t_dphi:t_dphi:pi/2-t_dphi; % range of phis, discard first
            (pole) and last (plane)
113
114     angle_list = [ 0 0 ];
115     for i=1:length(t_phis)
```

```
116         t_phi = t_phis(i);
117         angle_list = [ angle_list ; 0 t_phi ];
118     end
119
120     %angle_list = angle_list([ 1 2 3 4 19 20 21 22 38 39 40 41 ],:)
121
122     % limit angles for tests
123     %angle_list = angle_list(sin(4*angle_list(:,2)).^2 >= 0.995,:); % wedges
            in azimuthal angle
124
125     v_distrib_raw = zeros(6,length(x_range) * length(y_range) *
            length(z_range) * length(v_range) * length(angle_list));
126     vdr_ind = 1;
127     for i=1:length(x_range)
128         for j=1:length(y_range)
129             for k=1:length(z_range)
130                 for l=1:length(angle_list)
131                     for m=1:length(v_range)
132                         v_distrib_raw(:,vdr_ind) = [ x_range(i) y_range(j)
                                z_range(k) v_range(m) angle_list(l,1)
                                angle_list(l,2) ];
133                         vdr_ind = vdr_ind + 1;
134                     end
135                 end
136             end
137         end
138     end
139
140     % Transform v_mag,theta,phi to v_x, v_y, v_z
141     v_distrib = v_distrib_raw;
142     t_v = sqrt(3.913903e-6*v_distrib_raw(4,:))/v0;
143     % number is 2/(m_e*c^2) in eV^-1
144     v_distrib(4,:) = t_v .* cos(v_distrib_raw(5,:)) .*
            sin(v_distrib_raw(6,:));
145     v_distrib(5,:) = t_v .* sin(v_distrib_raw(5,:)) .*
            sin(v_distrib_raw(6,:));
146     v_distrib(6,:) = t_v .* cos(v_distrib_raw(6,:));
147
148     d = v_distrib;
149 end
```

B.1.2 Alice

The Alice code provides `mirror_shards_alice(n_shard, n_cores, master_file)`, the primary worker function of the system. It must be fed its shard number and the assigned

number of cores by its calling PBS script. An incorrect core number will result in baffling failures to run. It loads its distribution from the input file corresponding to its `n_shard`, performs the processing until particles reach their target altitude, then saves its results as codistributed arrays to '`mshard-r<n_run>-<n_shard>of<n_shards>-output.mat`'.

```
function ret = mirror_shards_alice(n_shard, n_cores, master_file)
% mirror_shards_alice()
%
% Does the actual simulation work in the mirror_shards_* system.

    p_g = load(master_file,'n_run','n_shards', ...
        'dt','qE','qmt2','r0', 'long_enough', ...
        'target_length','mirror_ratio', 'saved_steps');

    target_z = -p_g.target_length*1000/p_g.r0;
    % negative because we're launching upwards
    length_factor = target_z^2/(p_g.mirror_ratio-1);
    % assumes 'end point' is z=0

    % load values from p_g
    n_run = p_g.n_run; n_shards = p_g.n_shards; dt = p_g.dt; qE = p_g.qE;
    qmt2 = p_g.qmt2; long_enough = p_g.long_enough; saved_steps =
        p_g.saved_steps;

    % n_run, N_shardpart, chunk_inds, v_sharddist;
    p_s = load(['mshard-r' num2str(n_run) '-' num2str(n_shard) 'of'
        num2str(n_shards) '-input.mat'], ...
        'N_shardpart','v_sharddist');

    N_shardpart = p_s.N_shardpart;
    v_sharddist = p_s.v_sharddist;

    parpool('local', n_cores);

    % This is local to the shard now, but we'll just redefine below for the
    % part of the distribution that's local to each worker.
    v_sdivdist = distributed(v_sharddist);

    disp([ 'Simulating ' num2str(N_shardpart) ' particles over INFINITE
        timesteps...' ])
    tic
    disp('Start')

    % spmd (single program, multiple data) is a more generalized
    % multithreaded methodology than parfor, and allows use of
    % distributed/codistributed functionality to split up arrays
    spmd

```

```
41          v_localdist = getLocalPart(v_sdivdist);
42          N_dpart = size(v_localdist, 2);
43          chunk_inds = globalIndices(v_sdivdist,2);
44
45          disp( [ 'Running ' num2str(N_dpart) ' particles ('
                num2str(chunk_inds(1)) ':' num2str(chunk_inds(end)) ') in Lab '
                num2str(labindex) '.' ] )
46
47          % Pre-allocate result arrays
48          gm_X = zeros([ 3, saved_steps, N_dpart ], 'double'); % x,y,z
49          gm_V = zeros([ 3, saved_steps, N_dpart ], 'double'); % vx,vy,vz
50          gm_Bv = zeros([ 3, saved_steps, N_dpart ], 'double'); % Bx,By,Bz
51
52          % redundant array of results, seven 3-vectors containing
53          % 1,2 position and velocity at target-z (z_t)
54          % 3, # of timestep before z_t, after, and actual calculated crossing
                time
55          % 4,5 position and velocity of pre-z_t timestep
56          % 6,7 position and velocity of post-z_t timestep
57          gm_result = zeros([ 3, 7, N_dpart ], 'double');
58
59          active_indices = 1:N_dpart;
60
61          % Get B at initial positions
62          % permute() lets us slot a (3,N) data peg into a (3,M,N) hole
63          gm_X(:,end-1,:) = permute(v_localdist(1:3,:),[1 3 2]);
64          gm_V(:,end-1,:) = permute(v_localdist(4:6,:),[1 3 2]);
65
66          % Recall all arrays are (dimension, timestep, particles)
67          gm_B_x = squeeze(-gm_X(1,end-1,active_indices) .*
                gm_X(3,end-1,active_indices) / length_factor);
68          gm_B_y = squeeze(-gm_X(2,end-1,active_indices) .*
                gm_X(3,end-1,active_indices) / length_factor);
69          gm_B_z = squeeze(1+gm_X(3,end-1,active_indices).^2 / length_factor);
70
71          % Calculate 2nd position with Boris Mover
72          gm_v_mh = squeeze(gm_V(:,end-1,:));
73          gm_v_minus = gm_v_mh + qE;
74
75          gm_B = [ gm_B_x gm_B_y gm_B_z ].';
76          gm_Bv(:,end,:) = gm_B;
77          gm_t_vec = qmt2*gm_B;
78          gm_s_vec = 2*gm_t_vec./(1+gm_t_vec.^2);
79          gm_v_prime = gm_v_minus + cross(gm_v_minus,gm_t_vec,1);
80          gm_v_plus = gm_v_minus + cross(gm_v_prime,gm_s_vec,1);
81
82          gm_V(:,end,:) = 0.5 .* (gm_v_mh + gm_v_plus + qE);
```

```
83          gm_X(:,end,:) = gm_X(:,end-1,:) + gm_V(:,end-1,:) .* dt;
84
85          tstep = 1;
86          % Loop until all particles are done.
87          while ~isempty(active_indices)
88              tstep = tstep + 1;
89
90              % shift saved-data matrices down one row
91              gm_X(:,1:end-1,active_indices) = gm_X(:,2:end,active_indices);
92              gm_V(:,1:end-1,active_indices) = gm_V(:,2:end,active_indices);
93              gm_Bv(:,1:end-1,active_indices) = gm_Bv(:,2:end,active_indices);
94
95              if labindex == 1 && mod(tstep,10000) == 0
96                  display(['Step ' num2str(tstep) ', '
                        num2str(length(active_indices)) ...
97                       ' particles active, min/max z = '
                            num2str(min(gm_X(3,end,active_indices))) '/'
                            num2str(max(gm_X(3,end,active_indices))) '.'])
98              end
99
100             % Recall all arrays are (dimension, timestep, particles)
101             gm_B_x = squeeze(-gm_X(1,end-1,active_indices) .*
                    gm_X(3,end-1,active_indices) / length_factor);
102             gm_B_y = squeeze(-gm_X(2,end-1,active_indices) .*
                    gm_X(3,end-1,active_indices) / length_factor);
103             gm_B_z = squeeze(1+gm_X(3,end-1,active_indices).^2 /
                    length_factor);
104
105             % half-step due to E-field
106             gm_v_minus = squeeze(gm_V(:,end-1,active_indices) + qE);
107
108             gm_B = [ gm_B_x gm_B_y gm_B_z ].';
109             gm_Bv(:,end,active_indices) = gm_B;
110             gm_t_vec = qmt2*gm_B;
111             gm_s_vec = 2*gm_t_vec./(1+gm_t_vec.^2);
112             % these calculate the B-field effects
113             gm_v_prime = gm_v_minus + cross(gm_v_minus,gm_t_vec);
114             gm_v_plus = gm_v_minus + cross(gm_v_prime,gm_s_vec);
115
116             % second half-step from E-field
117             gm_V(:,end,active_indices) = gm_v_plus + qE;
118
119             gm_X(:,end,active_indices) = gm_X(:,end-1,active_indices) +
                    gm_V(:,end-1,active_indices) .* dt;
120
121             % check if next z-pos passes the target plane z=0
122             strike_indices = active_indices(gm_X(3, end, active_indices) >=
```

```
            0);
123         if ~isempty(strike_indices)
124             %display([ 'Timestep ' num2str(tstep) ': '
                    num2str(length(strike_indices)) ' strikes.' ]);
125             % interpolate absolute strike XVT
126             % NB: assumes 'target z' is z=0 plane
127             t_nStrikes = length(strike_indices);
128             t_V0 = squeeze(gm_V(:,end-1,strike_indices)); % init and
129             t_V1 = squeeze(gm_V(:,end,strike_indices)); % final vel
130             t_X0 = squeeze(gm_X(:,end-1,strike_indices)); % init and
131             t_X1 = squeeze(gm_X(:,end,strike_indices)); % final pos
132
133             % acceleration from x_pre to x_post
134             t_a01 = ( t_V1-t_V0 )/dt;
135
136             % time to z=0
137             t_t0t = ( -t_V0(3,:) + sqrt(t_V0(3,:).^2 -
                    2*t_a01(3,:).*t_X0(3,:)) )./t_a01(3,:);
138             % velocity at z=0
139             t_Vt = t_V0 + bsxfun(@times,t_a01,t_t0t);
140             % complete pos at z=0
141             t_Xt = t_X0 + bsxfun(@times,t_V0,t_t0t) +
                    0.5*bsxfun(@times,t_a01,t_t0t.^2);
142
143             % target x, target v, times, x0, v0, x1, v1
144             gm_result(:,1,strike_indices) = squeeze(t_Xt);
145             gm_result(:,2,strike_indices) = squeeze(t_Vt);
146
147             t_t = [ (tstep+t_t0t)*dt ; repmat(tstep,1,t_nStrikes) ;
                    t_t0t ];
148
149             gm_result(:,3,strike_indices) = t_t;
150
151             gm_result(:,4,strike_indices) = squeeze(t_X0);
152             gm_result(:,5,strike_indices) = squeeze(t_V0);
153
154             gm_result(:,6,strike_indices) = squeeze(t_X1);
155             gm_result(:,7,strike_indices) = squeeze(t_V1);
156
157             active_indices = active_indices( ~ismember(active_indices,
                    strike_indices) );
158         end
159
160     end
161
162     display(['Final timesteps: ' num2str(tstep) '.'])
163
```

```
164         t_codist_result = codistributor1d(3, codistributor1d.unsetPartition,
                [3, 7, N_shardpart]);
165         t_codist_saved = codistributor1d(3, codistributor1d.unsetPartition,
                [3, saved_steps, N_shardpart]);
166
167         % build codist arrays
168         r_divres = codistributed.build(gm_result, t_codist_result,
                'noCommunication');
169         r_divsavX = codistributed.build(gm_X, t_codist_saved,
                'noCommunication');
170         r_divsavV = codistributed.build(gm_V, t_codist_saved,
                'noCommunication');
171         r_divsavB = codistributed.build(gm_Bv, t_codist_saved,
                'noCommunication');
172
173     end % spmd
174
175     % gather() to recombine distributed arrays
176     r_shard_res = gather(r_divres);
177     r_shard_X = gather(r_divsavX);
178     r_shard_V = gather(r_divsavV);
179     r_shard_B = gather(r_divsavB);
180     r_shard_dist = gather(v_sdivdist);
181
182     save(['mshard-r' num2str(n_run) '-' num2str(n_shard) 'of'
            num2str(n_shards) '-output.mat'], ...
183          'N_shardpart', 'r_shard_dist', ...
184          'r_shard_res', 'r_shard_X', 'r_shard_V', 'r_shard_B');
185
186     disp('End')
187     toc
188
189     ret = 0;
190
191 end
```

B.1.3 Gather

Our final code provides `mirror_shards_gather(master_file)`, which requires only the 'master' file generated by the Distribute function. It loads all applicable output files, runs `gather()` to join the `distributed()` matrix, and saves the final results to '`mshards-r<n_run>-final.mat`'.

```
1 function mirror_shards_gather(master_file)
2 % mirror_shards_gather()
3 %
4 % Compiles the output produced by per-node mirror shards that ran on input
```

```
% constructed by mirror_shards_distribute().

    % load the things we care about
    p_g = load(master_file, ...
        'n_run', 'n_shards', 'N_part', 'v_distrib', 'saved_steps');


    n_run = p_g.n_run; n_shards = p_g.n_shards; saved_steps =
       p_g.saved_steps;
    N_part = p_g.N_part; v_distrib = p_g.v_distrib;

    parpool('torque_4nodes',n_shards);

    disp([ 'Loading ' num2str(n_shards) ' shard outputs...' ])

    spmd

        [ v_sharddist, gm_result, gm_X, gm_V, gm_B ] =
           load_mah_data_plz(n_run, labindex, n_shards);

        t_codist_distrib = codistributor1d(2,
           codistributor1d.unsetPartition, [6, N_part]);
        t_codist_result = codistributor1d(3, codistributor1d.unsetPartition,
           [3, 7, N_part]);
        t_codist_saved = codistributor1d(3, codistributor1d.unsetPartition,
           [3, saved_steps, N_part]);

        % gather() to copy from GPU RAM to Main Memory
        % ...or just to combine sharded data...
        r_divdist = codistributed.build(v_sharddist, t_codist_distrib,
           'noCommunication');
        r_divres = codistributed.build(gm_result, t_codist_result,
           'noCommunication');
        r_divsavX = codistributed.build(gm_X, t_codist_saved,
           'noCommunication');
        r_divsavV = codistributed.build(gm_V, t_codist_saved,
           'noCommunication');
        r_divsavB = codistributed.build(gm_B, t_codist_saved,
           'noCommunication');

    end

    disp('Done, saving...')

    r_dist = gather(r_divdist);
    r_res = gather(r_divres);
    r_savX = gather(r_divsavX);
```

```
42      r_savV = gather(r_divsavV);
43      r_savB = gather(r_divsavB);
44
45      if ~isequal(r_dist, v_distrib)
46          disp('Rebuilt distribution does not equal OG distribution from
                master file!')
47      end
48
49      save([ 'mshards-r' num2str(n_run) '-final.mat' ], ...
50          'n_run', 'n_shards', 'N_part', 'v_distrib', ...
51          'r_dist', 'r_res', 'r_savX', 'r_savV', 'r_savB');
52
53      disp('...great success?')
54
55  end
56
57  function [ l_dist, l_res, l_gm_X, l_gm_V, l_gm_B ] =
        load_mah_data_plz(i_n_run,labindex,n_shards)
58
59      p_d = load([ 'mshard-r' num2str(i_n_run) '-' num2str(labindex) 'of'
            num2str(n_shards) '-output.mat' ], ...
60          'r_shard_dist', 'r_shard_res', 'r_shard_X', 'r_shard_V',
                'r_shard_B');
61
62      l_dist = p_d.r_shard_dist;
63      l_res  = p_d.r_shard_res;
64      l_gm_X = p_d.r_shard_X;
65      l_gm_V = p_d.r_shard_V;
66      l_gm_B = p_d.r_shard_B;
67
68  end
```

B.1.4 Bonus Code: GPU-Node Support

This is the final version of mirror code which runs on GPUs. Note that this is old, and there may be bug fixes and stuff in the Shards code that were not implemented here.

Algorithmically the code is essentially the same as Mirror Shards, but allocates its arrays with the 'gpuArray' parameter, and makes heavy use of `arrayfun()` on included functions, as this is faster on GPUs. As far as I can tell, when Matlab first sees an `arrayfun()` working on data stored in the GPU's RAM, it builds a CUDA kernel for that function, so future runs of the same type (as in a for loop) are GPU accelerated/parallelized as best as possible.

There is some turning point, dependent on number of particles and the necessary timesteps for the desired simulation length, between which either CPU sharding or GPU parallelism is the best choice. Of course, it also depends on what GPUs and CPUs are available.

mirror_gpu.m

```
function [ dist, resXVT, savedX, savedV ] = mirror_rtp()
% mirror_gpu_scriptable()
% Externally-scriptable version of test-particles-in-a-mirror-B-field
% simulation.  Comes with functions (below) to build distribution and
% construct the field, as well as various support functions.
% By default, will fall back to CPU processing if compatible GPUs
% are not present.

    q = 1;
    m = 1;
    nt = 10000;   % # timesteps
    dt = .1;     % step length
    qE = 0;
    qmt2 = q/m*dt/2;

    B0 = 1; % Magnetic field base is 50 uT
    v0 = 0.00989179273; % likewise velocity base in
                        % PSL is equivalent to 25 eV
    r0 = 0.337212985; % based on Larmour radius w/
                      % above, length base is ~0.337 m
    t0 = 0.714477319; % based on B, Larmour period ~714 ns

    target_length = 5000;   % in km
    target_z = -target_length/r0;  % negative because
                                   % we're launching upwards
    long_enough = 500;
    mirror_ratio = 5;
    saved_steps = 500;

    % So Bsim=Breal/50uT, vsim=vreal/25 eV, and xsim=xreal/0.337m
    % So a 100x100x1000 simulation extent is a 33.7x33.7x337m volume
    % So dt ~71.4ns, and 1000 timesteps is 71us

    % assumes 'end point' is z=0
    length_factor = target_z^2/(mirror_ratio-1);

    x_range = 0;
    y_range = 0;
    z_range = target_z;
    v_range = [25 484 1125];%[25, 36, 49, 64, 81, 100, 121, 144, ...
    % 169, 196, 225, 256, 289, 324, 361, 400, 441, 484, 529, ...
    % 576, 625, 676, 729, 784, 841, 900, 961, 1024, 1089, 1156, ...
    % 1225]; % linear in v
    t_dtheta = 3*pi/256; % delta for co-latitude
    t_domega = 0.001; % delta for solid angle in steradians
    %p_range = 0:pi/7:pi; %0:pi/15:pi/2;
```

```
47
48      v_distrib = build_distrib(v0, x_range, y_range, z_range, v_range,
            t_dtheta, t_domega);
49
50      N_part = size(v_distrib,2);
51      [ 'Simulating ' num2str(N_part) ' particles over maximum '
            num2str(long_enough) ' timesteps...' ]
52      N_ts = nt+2;
53
54      % distributed() is dumb, and requires the
55      % chunking dimension to be the last one.
56      v_sdivdist = distributed(v_distrib);
57
58      disp('Start')
59      tic
60
61      % spmd (single program, multiple data) is a more generalized
62      % multithreaded methodology than parfor, and allows use of
63      % distributed/codistributed functionality to split up arrays
64      spmd
65
66          v_localdist = getLocalPart(v_sdivdist);
67          N_dpart = size(v_localdist, 2);
68          chunk_inds = globalIndices(v_sdivdist,2);
69
70          d = gpuDevice();
71          disp( [ 'Running ' num2str(N_dpart) ' particles ('
                num2str(chunk_inds(1)) ':' num2str(chunk_inds(end)) ') in Lab '
                num2str(labindex) ' on GPU ' num2str(d.Index) '.' ] )
72
73          % Pre-allocate result arrays on GPU
74          gm_X = nan([ 3, saved_steps, N_dpart ], 'double', 'gpuArray');
75          gm_V = nan([ 3, saved_steps, N_dpart ], 'double', 'gpuArray');
76
77          % result is x,y,z,vx,vy,vz,t,ts1,ts2
78          gm_result = zeros([ 3, 3, N_dpart ], 'double', 'gpuArray');
79
80          d = gpuDevice();
81          t_tmem = d.TotalMemory;
82          t_umem = t_tmem-d.AvailableMemory;
83          disp( [ 'Memory Used: ' num2str(t_umem/1e9) '/' num2str(t_tmem/1e9)
                'GB (' num2str(t_umem/t_tmem*100) '%)' ]);
84
85          active_indices = 1:N_dpart;
86          length(active_indices)
87
88          % Get B at initial positions
```

```
89          % permute() lets us slot a (3,N) data peg into a (3,M,N) hole
90          gm_X(:,end-1,:) = permute(v_localdist(1:3,:),[1 3 2]);
91          gm_V(:,end-1,:) = permute(v_localdist(4:6,:),[1 3 2]);
92
93          % Recall all arrays are (dimension, timestep, particles)
94          gm_B_x = squeeze(arrayfun(@bxcalc,gm_X(1,end-1,:), gm_X(3,end-1,:),
                length_factor));
95          gm_B_y = squeeze(arrayfun(@bycalc,gm_X(2,end-1,:), gm_X(3,end-1,:),
                length_factor));
96          gm_B_z = squeeze(arrayfun(@bzcalc,gm_X(3,end-1,:), length_factor));
97
98          % Calculate 2nd position with Boris Mover
99          gm_v_mh = squeeze(gm_V(:,end-1,:));
100         gm_v_minus = gm_v_mh + qE;
101
102         gm_t_vec = tcalc(gm_B_x,gm_B_y,gm_B_z,qmt2);
103         gm_s_vec = scalc(gm_t_vec);
104         size(gm_v_minus)
105         size(gm_t_vec)
106         gm_v_prime = gm_v_minus + cross(gm_v_minus,gm_t_vec,1);
107         gm_v_plus = gm_v_minus + cross(gm_v_prime,gm_s_vec,1);
108
109         gm_V(:,end,:) = 0.5 .* (gm_v_mh + gm_v_plus + qE);
110         gm_X(:,end,:) = gm_X(:,end-1,:) + gm_V(:,end-1,:) .* dt;
111
112         tstep = 1;
113         % Loop until all particles are done, or we've
114         % done an absurd number of timesteps.
115         while ~isempty(active_indices) && (tstep <= long_enough)
116             tstep = tstep + 1;
117
118             % shift saved-data matrices down one row
119             gm_X(:,1:end-1,:) = gm_X(:,2:end,:);
120             gm_V(:,1:end-1,:) = gm_V(:,2:end,:);
121
122             if labindex == 1 && mod(tstep,100) == 0
123                 display(['Step ' num2str(tstep) ', '
                        num2str(length(active_indices)) ...
124                      ' particles active, min/max z = '
                           num2str(min(gm_X(3,end-1,active_indices))) '/'
                           num2str(max(gm_X(3,end-1,active_indices))) '.'])
125             end
126
127             % Recall all arrays are (dimension, timestep, particles)
128             gm_B_x = squeeze(arrayfun(@bxcalc, gm_X(1,end-1,active_indices),
                    gm_X(3,end-1,active_indices), length_factor));
129             gm_B_y = squeeze(arrayfun(@bycalc, gm_X(2,end-1,active_indices),
```

```
            gm_X(3,end-1,active_indices), length_factor));
130         gm_B_z = squeeze(arrayfun(@bzcalc, gm_X(3,end-1,active_indices),
                length_factor));
131
132         % half-step due to E-field
133         gm_v_minus = squeeze(gm_V(:,end-1,active_indices) + qE);
134
135         gm_t_vec = tcalc(gm_B_x,gm_B_y,gm_B_z,qmt2);
136         gm_s_vec = scalc(gm_t_vec);
137         % these calculate the B-field effects
138         gm_v_prime = gm_v_minus + cross(gm_v_minus,gm_t_vec);
139         gm_v_plus = gm_v_minus + cross(gm_v_prime,gm_s_vec);
140
141         % second half-step from E-field
142         gm_V(:,end,active_indices) = gm_v_plus + qE;
143         gm_X(:,end,active_indices) = gm_X(:,end-1,active_indices) +
                gm_V(:,end-1,active_indices) .* dt;
144
145         % check if next z-pos passes the target plane
146         strike_indices = active_indices(gm_X(3,end,active_indices) > 0);
147         if ~isempty(strike_indices)
148             display([ 'Timestep ' num2str(tstep) ': '
                    num2str(length(strike_indices)) ' strikes.' ]);
149             % interpolate absolute strike time?
150             gm_result(:,1,strike_indices) =
                    squeeze(gm_X(:,end,strike_indices));
151             gm_result(:,2,strike_indices) =
                    squeeze(gm_V(:,end,strike_indices));
152             gm_result(:,3,strike_indices) = repmat([ tstep-1 ; tstep ;
                    tstep*dt*t0 ],[1 length(strike_indices)]);
153             active_indices = active_indices( ~ismember(active_indices,
                    strike_indices) );
154         end
155     end
156
157     t_codist_result = codistributor1d(3, codistributor1d.unsetPartition,
            [3, 3, N_part]);
158     t_codist_saved = codistributor1d(3, codistributor1d.unsetPartition,
            [3, saved_steps, N_part]);
159
160     % gather() to copy from GPU RAM to Main Memory
161     r_divres = codistributed.build(gather(gm_result), t_codist_result,
            'noCommunication');
162     r_divsavX = codistributed.build(gather(gm_X), t_codist_saved,
            'noCommunication');
163     r_divsavV = codistributed.build(gather(gm_V), t_codist_saved,
            'noCommunication');
```

```
164
165     end % spmd
166
167     % gather() again to recombine distributed arrays
168     r_result = gather(r_divres);
169     r_savX = gather(r_divsavX);
170     r_savV = gather(r_divsavV);
171
172     toc
173     'Stop'
174
175     % results to output variables
176     dist = v_distrib;
177     resXVT = r_result;
178     savedX = r_savX;
179     savedV = r_savV;
180
181 end
182
183 function n = bxcalc(x,z,L_z2)
184     n = -x*z/L_z2;
185 end
186
187 function n = bycalc(y,z,L_z2)
188     n = -y*z/L_z2;
189 end
190
191 function n = bzcalc(z,L_z2)
192     n = (1+z^2/L_z2);
193 end
194
195 function n = threenorm(x,y,z)
196
197     n = sqrt(x^2+y^2+z^2);
198
199 end
200
201 function n = tcalc(bx,by,bz,c)
202
203     n = c*[ bx by bz ].';
204
205 end
206
207 function n = scalc(t)
208
209     n = 2*t./(1 + t.^2);
210
```

```
211 end
212
213 function d = build_distrib(v0, x_range, y_range, z_range, v_range, t_dtheta,
        t_domega)
214     % Build particle distribution
215
216     % initial positions x y z
217     % initial velocities v theta phi (magnitude, azimuth, elevation)
218     %   mag 25:2000 eV, azi 0:pi, el 0:pi/2
219     % input as [ x y z v theta phi ] columns in v_distrib_raw
220
221     % range of thetas, discard first (pole) and last (plane)
222     t_range = 0+t_dtheta:t_dtheta:pi/2-t_dtheta;
223
224     angle_list = [ 0 0 ];
225     for i=1:length(t_range)
226         theta = t_range(i);
227         for omega=0:2*pi/round(2*pi*sin(theta)*t_dtheta/t_domega):2*pi
228             angle_list = [ angle_list ; theta omega ];
229         end
230     end
231
232     v_distrib_raw = zeros(6,length(x_range) * length(y_range) *
            length(z_range) * length(v_range) * length(angle_list));
233     vdr_ind = 1;
234     for i=1:length(x_range)
235         for j=1:length(y_range)
236             for k=1:length(z_range)
237                 for l=1:length(angle_list)
238                     for m=1:length(v_range)
239                         v_distrib_raw(:,vdr_ind) = [ x_range(i) y_range(j)
                                z_range(k) v_range(m) angle_list(l,1)
                                angle_list(l,2) ];
240                         vdr_ind = vdr_ind + 1;
241                     end
242                 end
243             end
244         end
245     end
246
247     % Transform v_mag,theta,phi to v_x, v_y, v_z
248     v_distrib = v_distrib_raw;
249     % number is 2/(m_e*c^2) in eV^-1
250     t_v = sqrt(3.913903e-6*v_distrib_raw(4,:))/v0;
251     v_distrib(4,:) = t_v .* cos(v_distrib_raw(6,:)) .*
            cos(v_distrib_raw(5,:));
252     v_distrib(5,:) = t_v .* cos(v_distrib_raw(6,:)) .*
```

```
        sin(v_distrib_raw(5,:));
    v_distrib(6,:) = t_v .* sin(v_distrib_raw(6,:));

    d = v_distrib;
end

function d = test_distrib()

    test_array = [  1 3 8 ;
                    4 7 2 ;
                    9 1 7 ;
                    4 6 2 ;
                    1 1 1 ];

    test_v = [ 0 2 1.22 ;
               0 1.9 1.22 ;
               0 2.1 1.22 ;
               0 2 1.12 ;
               0 2 1.32 ];

    d = shiftdim([ test_array test_v ],1);
end

function [ Xg, Yg, Zg, Bx, By, Bz ] = build_field()
    % Field Initialization
    L_xyz = 100;
    n_xyz = 100;
    k = pi/L_xyz;
    x0 = 51; y0 = 51; z0 = 51;

    % Generate the grid of the magnetic field
    [X, Y, Z] = meshgrid(1:100, 1:100, 1:1000);
    B_xgrid = zeros(100,100,1000); B_ygrid = zeros(100,100,1000); B_zgrid =
        zeros(100,100,1000);
    %B_maggrid = zeros(100,100,100);
    for ii = 1:100
        for jj = 1:100
            for kk = 1:1000
    %             B_xgrid(ii,jj,kk) = 0;
    %             B_ygrid(ii,jj,kk) = 0;
    %             B_zgrid(ii,jj,kk) = -1;
                % 2.85966 factor normalizes field so max magnitude is 1
                B_xgrid(ii,jj,kk) = -(5/8) * k * sin(k*(kk-z0)) * (ii - x0)
                    / 2.85966;
                B_ygrid(ii,jj,kk) = -(5/8) * k * sin(k*(kk-z0)) * (jj - y0)
                    / 2.85966;
                B_zgrid(ii,jj,kk) = 5 * (1 - .5*(1 + 1/8*k^2 * ((ii - x0)^2
```

```
                    + (jj - y0)^2)) * cos(k*(kk-z0))) / 2.85966;
296             %   B_maggrid(ii,jj,kk) = sqrt(B_xgrid(ii,jj,kk)^2 +
                    B_ygrid(ii,jj,kk)^2 + B_zgrid(ii,jj,kk)^2);
297             end
298         end
299     end
300
301     Xg = X; Yg = Y; Zg = Z;
302     Bx = B_xgrid; By = B_ygrid; Bz = B_zgrid;
303 end
304
305 function ok = selectGPUDeviceForLab()
306
307     persistent hasGPU;
308
309     if isempty( hasGPU )
310         devIdx = mod(labindex-1,gpuDeviceCount())+1;
311         try
312             dev = gpuDevice( devIdx );
313             hasGPU = dev.DeviceSupported;
314         catch %#ok
315             hasGPU = false;
316         end
317     end
318     ok = hasGPU;
319
320 end
```

B.1.5 Support Scripts

This is a simple python script which takes the desired number of shards and cores per shard, as well as cell, wall time per core, and a run-identification number. It runs the Distribute function to create the master and shard-input files, and creates a PBS script using the template file (below), and accompanying submission script.

There is one problem with this, related to the fact that the script does not know how many total particles the Distribute function will be creating. If you're going for a one-particle-per-core scenario, and your particles are not evenly divided by your number of cores per shard, then you can end up with a scenario where on some shards you have more cores than particles, and end up with Matlab workers crashing and messing things up (and crashed workers don't give very nice feedback).

Fixing this requires knowing how Matlab's distributed() function split up the array. The script will try to figure that out by looking at input file sizes, but you can specify it manually with the `-m` option. Either way, we'll set up two PBS scripts, and a two-stage submission

script to switch between the two. To disable this behavior (e.g. if not aiming for a 1:1 particle:core ratio) , use `-m -1`.

Finally, there's a simple script to run the Gather function.

`mss.py`:

```
#!/usr/bin/env python

from optparse import OptionParser
import subprocess
import sys

parser = OptionParser("Usage: %prog -r <run number> [options]")
parser.add_option("-r","--run",dest="run",type="int",default=0,
    help="Run number [required].")
parser.add_option("-c","--cell",dest="cell",type="string",default="j",
    help="Cell to use [default: %default].")
parser.add_option("-s","--shards",dest="shards",type="int",default=20,
    help="Number of shards to break distribution into [%default].")
parser.add_option("-n","--cores",dest="cores",type="int",default=24,
    help="Number of cores per shard [%default].")
parser.add_option("-w","--wall",dest="wall",type="int",default=50,
    help="Wall time per core [%default].")
parser.add_option("-m","--modulo",dest="modulo",type="int",default=0,
    help="Manually specify modulo point.  The first M shards will be given N
        cores per shard, the remainder will be given N-1.  Set to -1 to
        disable auto-detect.")

(opt, args) = parser.parse_args()

if opt.run==0:
    print("Must provide a run number with -r.")
    sys.exit()

# break up distribution
subprocess.call(["matlab","-nodisplay","-r",
    "try; mirror_shards_distribute({0},{1}); catch; end;
        quit".format(opt.run,opt.shards)])

def ms_size(i,t):
    # returns size of an mshard file
    fname = "mshard-r4-{0}of{1}-input.mat".format(i,t)
    return os.path.getsize(fname)

if opt.modulo==0:
    # try to figure out where Distribute put the modulo point
    size = ms_size(1,opt.shards)
    for i in range(2,opt.shards+1):
```

```
40          if size != ms_size(i,opt.shards):
41              opt.modulo = i-1
42              break
43          if i==opt.shards:
44              opt.modulo = opt.shards
45
46  elif opt.modulo == -1:
47      # auto-detect disabled
48      opt.modulo = opt.shards
49
50  def mss_files(cores,wall,cell,run,mtag):
51      # function to create the PBS scripts
52      pbsfn = "mss_PBS-r{0}-m{1}.sh".format(run,mtag)
53      pbsfile = open(pbsfn,"w")
54
55      subprocess.call(["sed",
56          "s/@@PPN@@/{0}/; s/@@WALL@@/{1}/; s/@@CELL@@/cell{2}/;
               s/@@RUN@@/{3}/;".format(cores,wall,cell,run),
57              "./mirror_shards.PBStemplate"],
58              stdout=pbsfile)
59      pbsfile.close()
60
61  # create primary PBS and job submission scripts
62  mss_files(opt.cores,opt.wall,opt.cell,opt.run,0)
63
64  subfile = open("mss_submit-r{0}.sh".format(opt.run),"w")
65  subfile.write("#!/bin/bash\n\n")
66  subfile.write("for ((i=1 ; i<={0} ; i++)); do\n".format(opt.modulo))
67  subfile.write("\tqsub -t $i mss_PBS-r{0}-m{1}.sh\n".format(opt.run,0))
68  subfile.write("\tsleep 7\n")
69  subfile.write("done\n")
70
71  if opt.modulo != opt.shards:
72      # if modulo, create second PBS script,
73      # add second part to submission script
74      mss_files(opt.cores-1,opt.wall,opt.cell,opt.run,1)
75
76      subfile.write("\nfor ((i={0} ; i<={1} ; i++));
           do\n".format(opt.modulo+1,opt.shards))
77      subfile.write("\tqsub -t $i mss_PBS-r{0}-m{1}.sh\n".format(opt.run,1))
78      subfile.write("\tsleep 7\n")
79      subfile.write("done\n")
80
81  subfile.close()
82
83  # clean up output from Distribute
84  subprocess.call("rm -rf Job1*",shell=True)
```

mirror_shards.PBStemplate:

```
#!/bin/bash -l
# declare a name for this job to be sample_job
#PBS -N mirror_shard

# request the default queue for this job
#PBS -q default

#PBS -l nodes=1:ppn=@@PPN@@
#PBS -l walltime=@@WALL@@:00:00
#PBS -l feature='@@CELL@@'

# mail is sent to you when the job starts
# and when it terminates or aborts
#PBS -m bea

# specify your email address
#PBS -M micah.p.dombrowski.gr@dartmouth.edu

#change to the directory where you submitted the job
cd $PBS_O_WORKDIR

# include the relative path to the name of your MPI program
matlab -nodisplay -r "try; mirror_shards_alice($PBS_ARRAYID,@@PPN@@,
    'mshards-r@@RUN@@-master.mat'); catch; end; quit"
```

msf.py:

```
#!/usr/bin/env python

from optparse import OptionParser
import subprocess, sys, os

parser = OptionParser("Usage: %prog -r <run number> [options]")
parser.add_option("-r","--run",dest="run",type="int",default=0,
    help="Run number [required].")

(opt, args) = parser.parse_args()

if opt.run==0:
    print("Must provide a run number with -r.")
    sys.exit()

# gather distribution
subprocess.call(["matlab","-nodisplay","-r",
    "try; mirror_shards_gather('mshards-r{0}-master.mat'); catch; end;
       quit".format(opt.run)])
```

B.2 Result Reformation

These functions and codes take the raw output from the Mirror Shards system, and turn it into something neatly packaged and usable in the later stages. The data Mirror Shards returns is: three matrices, of the final 1,000 points for each particle, of position, velocity, and magnetic field, and a 'result' array of data regarding travel times.

B.2.1 Gyro-Interpolation

Takes the raw data for each particle (the last 1,000 positions and velocities) and fits an interpolating gyro-orbit function to it, then uses that to interpolate to the actual strike values at the target plane. Uses the `HyperSVD()` algebraic circle-fitting function by Nikolai Chernov (http://people.cas.uab.edu/~mosya/cl/), included below.

`gyroterpolate.m`

```
function [ t_tz_time, t_Xf, t_Vf, t_mphi, t_circ ] = gyroterpolate(t_X, t_V,
    t_B, target_z, dt, t_d)
% Takes 3xTxN input vectors, where T = some number of saved timesteps,
% and N = some number of particles which have crossed the target
% z-level.  Fits a gyro-orbit to each particle's track, then
% interpolates the actual strike XVT.

    Ns = size(t_X, 2);
    Np = size(t_X, 3);
    t_tz_time = zeros(Np,1);
    t_Xf = zeros(Np,3);
    t_Vf = zeros(Np,3);
    t_mphi = zeros(Np,1);
    t_circ = zeros(Np,3);

    options = optimoptions('fminimax');
    options.Display = 'none';

    parfor part=t_d%1:Np

        t_pX = squeeze(t_X(:,:,part));
        t_pV = squeeze(t_V(:,:,part));

        tz_center = HyperSVD(squeeze(t_pX(1:2,:)).');

        if tz_center(3) ~= 0 % gyropath
```

```
26
27              t_circ(part,:) = tz_center;
28
29              % We know the equations of motion that the particle should
30              % be following; the only thing we don't know is the phase.
31              omega = sqrt(sum(squeeze(t_B(:,:,part)).^2,1)); % Bmag
32              vperp = sqrt( squeeze(t_pV(1,:)).^2 + squeeze(t_pV(2,:)).^2 );
33              vpar = t_pV(3,:);
34              ttime = (-(Ns-2):1)*dt;
35
36              t_Xc = [ t_pX(1,:) ; t_pX(2,:) ; t_pX(3,:) ];
37              t_x0 = t_pX(1,end-1);
38              t_y0 = t_pX(2,end-1);
39              t_z0 = t_pX(3,end-1);
40              deltx = t_Xc(1,:);
41              delty = t_Xc(2,:);
42              deltz = t_Xc(3,:)-t_z0;
43              Cx = -vperp./omega;
44              Cy =  vperp./omega;
45              Cz = vpar.*ttime;
46              tau = omega.*ttime;
47
48              % geometric error
49              ferrorphi = @(phi) ferrorphifunc(phi, deltx, delty, deltz, tau,
                    Cx, Cy, Cz);
50
51              [ t_mphi(part), ~ ] = fminimax(ferrorphi, pi, [], [], [], [], 0,
                    2*pi, [], options);
52
53              % X, V before target crossing
54              tz_x0 = t_pX(1,end-1); tz_vx0 = t_pV(1,end-1);
55              tz_y0 = t_pX(2,end-1); tz_vy0 = t_pV(2,end-1);
56              tz_z0 = t_pX(3,end-1); tz_vz0 = t_pV(3,end-1);
57              % travel time to target crossing
58              t_tz_time(part) = (target_z-tz_z0)/vpar(end-1);
59
60              % now just use gryo equations to get final interp. results
61              tz_time = t_tz_time(part);
62              tz_vperp = sqrt(tz_vx0^2 + tz_vy0^2);
63              tz_omega = omega(end-1);
64              tz_tau = tz_omega*(ttime(end-1)+tz_time);
65              tz_phi = t_mphi(part);
66
67              tz_xf = -tz_vperp/tz_omega*cos(tz_tau + tz_phi) + tz_center(1);
68              tz_yf =  tz_vperp/tz_omega*sin(tz_tau + tz_phi) + tz_center(2);
69              tz_zf =  tz_vz0*tz_time + tz_z0;
70
```

```
            tz_vxf = tz_vperp*sin(tz_tau + tz_phi);
            tz_vyf = tz_vperp*cos(tz_tau + tz_phi);
            tz_vzf = tz_vz0;

            [ tz_x0 tz_y0 tz_z0 ];
            t_Xf(part,:) = [ tz_xf tz_yf tz_zf ];
            [ tz_vx0 tz_vy0 tz_vz0 ];
            t_Vf(part,:) = [ tz_vxf tz_vyf tz_vzf ];

        else % straight line

            % travel time to target crossing
            t_tz_time(part) = (target_z-t_pX(3,end-1))/t_pV(3,end-1);

            % x, y, and velocities don't change, just set z = target_z
            t_Xf(part,:) = [ t_pX(1,end-1) t_pX(2,end-1) target_z ].';
            t_Vf(part,:) = t_pV(:,end-1).';
            t_mphi(part) = NaN;
            t_circ(part,:) = [ t_pX(1,end-1) t_pX(2,end-1) 0 ];

        end



    end

    %fdx = fdx + t_circ(1);
    %fdy = fdy + t_circ(2);

    %    phi = fminbnd(ferrorphi, 0, 2*pi);

end

function err = ferrorphifunc(phi,deltx,delty,deltz,tau,Cx,Cy,Cz)

    err = sqrt( ...
        (deltx - Cx.*cos(tau + phi)).^2 + ...
        (delty - Cy.*sin(tau + phi)).^2 + ...
        (deltz - Cz).^2);

end

function vp = vel_upd(t, v, B, phi)
    v_perp = sqrt(v(1).^2 + v(2).^2);
    omega_g = 2*pi*B;

    vp(1) = v_perp.*sin(omega_g.*t + phi);
```

```
    vp(2) = v_perp.*sin(omega_g.*t + phi);
    vp(3) = v(3);
end

function xp = pos_upd(t, x, v, B, phi)
    v_perp = sqrt(v(1).^2 + v(2).^2);
    omega_g = 2*pi*B;

    xp(1) = x(1) + -v_perp/omega_g.*cos(omega_g.*t + phi);
    xp(2) = x(2) +  v_perp/omega_g.*sin(omega_g.*t + phi);
    xp(3) = x(3) + v(3).*t;
end
```

HyperSVD.m

```
function Par = HyperSVD(XY)
%-------------------------------------------------------------------
%
%     Algebraic circle fit with "hyperaccuracy"
%     (with zero essential bias)
%
%     Input:  XY(n,2) is the array of coordinates
%             of n points x(i)=XY(i,1), y(i)=XY(i,2)
%
%     Output: Par = [a b R] is the fitting circle:
%             center (a,b) and radius R
%
%     Note: this is a version optimized for stability, not for speed
%
%-------------------------------------------------------------------

centroid = mean(XY);   % the centroid of the data set

X = XY(:,1) - centroid(1);  %  centering data
Y = XY(:,2) - centroid(2);  %  centering data
Z = X.*X + Y.*Y;
ZXY1 = [Z X Y ones(length(Z),1)];
[U,S,V]=svd(ZXY1,0);
if (S(4,4)/S(1,1) < 1e-12)   %  singular case
    A = V(:,4);
else                         %  regular case
    R = mean(ZXY1);
    N = [8*R(1) 4*R(2) 4*R(3) 2; 4*R(2) 1 0 0; 4*R(3) 0 1 0; 2 0 0 0];
    W = V*S*V';
    [E,D] = eig(W*inv(N)*W);
    [Dsort,ID] = sort(diag(D));
    Astar = E(:,ID(2));
```

```
    A = W\Astar;
end

Par = [-(A(2:3))'/A(1)/2+centroid ,
    sqrt(A(2)*A(2)+A(3)*A(3)-4*A(1)*A(4))/abs(A(1))/2];

end   %  HyperSVD
```

B.2.2 Hemispherical Filling

Takes particles which were launched at one azimuthal angle, and rotates the whole system to get a gyrotropic set with a constant solid angle subtended.

hemi_fill.m

```
function [ n, n_azi, rel ] = hemi_fill(xvt,r_dist,t_dphi,t_domega,varargin)
% hemi_fill: function to populate a constant-solid-angle hemisphere,
% given a single stripe of co-latitude positions and velocities, the
% corresponding co-latitudes, and the solid angle value in steradians.
%
% xv should be a 3x2xN vector, where N=(2pi/dphi)-2,
% positions are in (:,1,:), and velocities in (:,2,:)

    opt = struct('cell',false,'phistop',2*pi);
    opt = optParse(opt,varargin{:});

    % range of thetas, discard first (pole) and last (equator)
    t_phis = 0+t_dphi:t_dphi:pi/2-t_dphi;
    n_phis = length(t_phis);
    t_alpha = sqrt(r_dist(4,:).^2 + r_dist(5,:).^2)./r_dist(6,:);
    n_En = length(find(t_alpha == 0));

    nc_Xt = cell(n_phis,n_En,1);
    nc_Vt = cell(n_phis,n_En,1);
    nc_Xb = cell(n_phis,n_En,1);
    nc_Vb = cell(n_phis,n_En,1);
    nc_t = cell(n_phis,n_En,1);
    n_azi = zeros(n_phis,1);
    nc_rel = cell(n_phis,n_En,1);
    for i=1:n_phis
        t_phi = t_phis(i);

        n_azi(i) = round(2*pi*sin(t_phi)*t_dphi/t_domega);
        l_thetas = 0:2*pi/n_azi(i):opt.phistop;
        n_az = length(l_thetas);
        [ n_az n_azi(i) ];
```

```
32          if n_az ~= n_azi(i)
33          %    display('fuuu');
34              n_azi(i) = n_az;
35          end
36          for j=1:n_En
37              part = (i-1)*n_En + j;
38              display(['fnh ' num2str(part) ' lsjdf ' num2str(size(xvt))])
39              t_x = squeeze(xvt(:,1,part));
40              t_v = squeeze(xvt(:,2,part));
41              b_x = r_dist(1:3,part);
42              b_v = r_dist(4:6,part);
43              t_t = squeeze(xvt(:,3,part));
44
45              n_xt = zeros(3,n_az);
46              n_vt = zeros(3,n_az);
47              n_xb = zeros(3,n_az);
48              n_vb = zeros(3,n_az);
49              for k=1:n_az
50
51                  t_theta = l_thetas(k);
52                  t_rot = [ cos(t_theta) sin(t_theta) 0 ; -sin(t_theta)
                        cos(t_theta) 0 ; 0 0 1 ];
53
54                  n_xt(:,k) = t_rot*t_x;
55                  n_vt(:,k) = t_rot*t_v;
56                  n_xb(:,k) = t_rot*b_x;
57                  n_vb(:,k) = t_rot*b_v;
58              end
59
60              nc_Xt{i,j} = n_xt;
61              nc_Vt{i,j} = n_vt;
62              nc_Xb{i,j} = n_xb;
63              nc_Vb{i,j} = n_vb;
64              nc_t{i,j} = t_t(3);
65              nc_rel{i,j} = part;
66          end
67
68      end
69
70      n_vec = sum(n_azi)*n_En;
71      if opt.cell
72          n = cell(n_phis,5);
73          n(:,1) = nc_Xt;
74          n(:,2) = nc_Vt;
75          n(:,3) = nc_Xb;
76          n(:,4) = nc_Vb;
77          n(:,5) = nc_t;
```

```
78      else
79          n = zeros(13,n_vec);
80          rel = zeros(1,n_vec);
81          i_n = 0;
82          for i=1:n_phis
83              for j=1:n_En
84                  for k=1:n_azi(i)
85                      i_n = i_n + 1;
86                          n(:,i_n) = [ nc_Xt{i,j}(:,k) ; nc_Vt{i,j}(:,k) ;
                                nc_Xb{i,j}(:,k) ; nc_Vb{i,j}(:,k) ; nc_t{i,j} ];
87                      rel(i_n) = nc_rel{i,j};
88                  end
89              end
90          end
91      end
92
93  end
94
95  function optstr = optParse(options, varargin)
96
97      %# read the acceptable names
98      optionNames = fieldnames(options);
99
100     %# count arguments
101     nArgs = length(varargin);
102     if round(nArgs/2)~=nArgs/2
103        error('hemi_fill needs propertyName/propertyValue pairs')
104     end
105
106     for pair = reshape(varargin,2,[]) %# pair is {propName;propValue}
107        inpName = lower(pair{1}); %# make case insensitive
108
109        if any(strcmp(inpName,optionNames))
110           %# Overwrite options. If you want you can test for the right
111           %# class here.  Also, if you find out that there is an option
112           %# you keep getting wrong, you can use "if strcmp(inpName,
113           %# 'problemOption'),testMore,end"-statements
114           options.(inpName) = pair{2};
115        else
116           error('%s is not a recognized parameter name',inpName)
117        end
118     end
119
120     optstr = options;
121 end
```

B.2.3 Data-Processing Utility Script

This script runs `gryoterpolate()` and `hemi_fill()`, fiddling with the data in between and after to yield a nicely structured result matrix for use in distribution building.

```
%% Mirror data raw output manipulation

% Important constants
Np = size(r_savX, 3);
B0 = 50e-6; % Magnetic field base is 50 uT
v0 = 0.00989179273; % likewise velocity base in
                    % PSL is equivalent to 25 eV
r0 = 0.337212985; % based on Larmour radius w/ above,
                  % length base is ~0.337 m
t0 = 7.14477319e-7; % based on B, Larmour period ~714 ns in s
dt = 0.01;

%% Gyro-interpolation

[ t_tz_time, t_Xf, t_Vf, t_mphi, t_circ ] =
    gyroterpolate(r_savX,r_savV,r_savB,0,dt,1:Np);

%% Build new results matrix

r_interp = zeros(3, 3, Np);

% new layout:
% 1,2 position and velocity at target-z
% 3,  number of timesteps before target (unitless, # of timesteps),
%     fractional timestep to actually reach target (unitless, simulation
    time [timesteps*dt]),
%     total time (in ns, timesteps*dt*t0)

r_interp(:,1,:) = t_Xf.';
r_interp(:,2,:) = t_Vf.';

t_base_time = squeeze(r_res(2,3,:))-1;
t_time_ns = (t_base_time*dt+t_tz_time)*t0;

r_interp(:,3,:) = [ squeeze(r_res(2,3,:))-1 t_tz_time t_time_ns ].';

%% Hemi_fill to build a gyrotropic distribution.

t_dphi = 3*pi/256; % delta for co-latitude
t_domega = 0.001; % delta for solid angle in steradians
t_phis = 0+t_dphi:t_dphi:pi/2-t_dphi; % range of phis, discard
                                      % first (pole) and last (plane)

```

```
% final data is stored flat in a 13xN matrix
% top x y z top vx vy vz bottom x y z bottom vx vy vz time (ns)

[ r_hemiterp, ~, r_hemirel ] = hemi_fill(r_interp, r_dist, t_dphi, t_domega);

%% Save intermediate data

save J:\Data' Core'\particles\sim\run4-hemi.mat t_domega t_dphi t_phis B0 r0
    t0 v0 target_length target_z N_part dt mirror_ratio length_factor r_dist
    r_res r_interp r_hemiterp

%% Convert to units and reverse all velocities

Np = size(r_hemiterp,2);
eVconst = 3.913903e-6;

t_X_t = r_hemiterp(1:3,:)*r0; % Distances in m

t_vx_t = -r_hemiterp(4,:);
t_vy_t = -r_hemiterp(5,:);
t_vz_t = -r_hemiterp(6,:);
t_vmag_t = sqrt(t_vx_t.^2 + t_vy_t.^2 + t_vz_t.^2); % v, unitless
t_vpar_t = t_vz_t; % vpar, unitless
t_vper_t = sqrt(t_vx_t.^2 + t_vy_t.^2); % vper, unitless

t_alpha_t = atan2(t_vper_t,-t_vpar_t);%*180/pi;
t_theta_t = 2*pi-atan2(t_vy_t,t_vx_t); % math to convert from atan2 output
    range to standard
t_theta_t(t_theta_t>=2*pi) = t_theta_t(t_theta_t>=2*pi)-2*pi; % 0to2pi
    clockwise from +x angle
%t_theta_t = t_theta_t * 180/pi;

t_X_b = r_hemiterp(7:9,:)*r0; % Distances in m

t_En_t = (t_vmag_t*v0).^2/eVconst; % Energy, eV
t_vmag_t_mps = t_vmag_t*v0*299792458; % convert to m/s
t_vpar_t_mps = t_vpar_t*v0*299792458;
t_vper_t_mps = t_vper_t*v0*299792458;

t_vx_b = -r_hemiterp(10,:);
t_vy_b = -r_hemiterp(11,:);
t_vz_b = -r_hemiterp(12,:);
t_vmag_b = sqrt(t_vx_b.^2 + t_vy_b.^2 + t_vz_b.^2); % v, unitless
t_vpar_b = t_vz_b; % vpar, unitless
t_vper_b = sqrt(t_vx_b.^2 + t_vy_b.^2); % vper, unitless

t_alpha_b = atan2(t_vper_b,-t_vpar_b);%*180/pi;
```

```
t_theta_b = 2*pi-atan2(t_vy_b,t_vx_b); % math to convert from atan2 output
    range to standard
t_theta_b(t_theta_b>=2*pi) = t_theta_b(t_theta_b>=2*pi)-2*pi; % 0to2pi
    clockwise from +x angle
%t_theta_b = t_theta_b * 180/pi;

t_En_b = (t_vmag_t*v0).^2/eVconst; % Energy, eV
t_vmag_b_mps = t_vmag_b*v0*299792458; % convert to m/s
t_vpar_b_mps = t_vpar_b*v0*299792458;
t_vper_b_mps = t_vper_b*v0*299792458;

% 17xN full-results matrix
% (x,y,z, vmag, vpar, vper, pa, azi)_top
% (x,y,z, vmag, vpar, vper, pa, azi)_bottom
% time

%r_mirror_XVPA = [ ...
%    t_X_t(1,:) ; t_X_t(2,:) ; t_X_t(3,:) ; ...
%    t_vmag_t_mps ; t_vpar_t_mps ; t_vper_t_mps ; ...
%    t_alpha_t ; t_theta_t ; ...
%    t_X_b(1,:) ; t_X_b(2,:) ; t_X_b(3,:) ; ...
%    t_vmag_b_mps ; t_vpar_b_mps ; t_vper_b_mps ; ...
%    t_alpha_b ; t_theta_b ; ...
%    r_hemiterp(13,:) ...
%    ];

% Discard X (assume homogeneity), re-add Energies
% 13xN matrix
% (En, vmag, vpar, vper, pa, azi)_top
% (En, vmag, vpar, vper, pa, azi)_bottom
% time

r_mirror_EVPA = [ ...
    t_En_t ; ...
    t_vmag_t_mps ; t_vpar_t_mps ; t_vper_t_mps ; ...
    t_alpha_t ; t_theta_t ; ...
    t_En_b ; ...
    t_vmag_b_mps ; t_vpar_b_mps ; t_vper_b_mps ; ...
    t_alpha_b ; t_theta_b ; ...
    r_hemiterp(13,:) ...
    ];

% Build a map structure, to keep track of what's what.

smap_EVPA.top.En = 1;
smap_EVPA.top.v.mag = 2;
smap_EVPA.top.v.para = 3;
```

```
smap_EVPA.top.v.perp = 4;
smap_EVPA.top.alpha = 5;
smap_EVPA.top.theta = 6;
smap_EVPA.bot.En = 7;
smap_EVPA.bot.v.mag = 8;
smap_EVPA.bot.v.para = 9;
smap_EVPA.bot.v.perp = 10;
smap_EVPA.bot.alpha = 11;
smap_EVPA.bot.theta = 12;
smap_EVPA.time = 13;
```

B.3 Distribution Building and Reduction, Growth Rates

B.3.1 Maxwell-Boltzmann Distribution

Takes parameters, and arrays that tell it where its sample points in energy pitch-angle phase space are, and builds a Maxwellian distribution.

maxwellian.m

```
function [ maxw_vel ] = maxwellian( temp, shift_eV, PAcenter, PAwidth,
    in_velocities, in_angles)
%maxwellian(temp, PA center, PA width, input eneriges, input angles)
% Returns a discretely sampled, joint probability distribution
% function, based on the input parameters, sampled at the provided
% energies and angles.  Temp in K, shift in eV, PA in degrees, leave
% PAcenter empty [] to use a flat pitch-angle distribution.

    v_th = sqrt(3*temp*15156333.1);     % Convert input temp. to
                                        % v_th = (3kT/m)^(1/2)
    eVconst = 3.913903e-6; % 2/(m_e*c^2) in eV^-1,
                           % i.e. conversion from eV to PSL
    v0 = 0.00989179273; % velocity base in PSL is equivalent to 25 eV

    shift = sqrt(shift_eV*eVconst)*299792458; % conv shift eV to m/s

    % Maxwell-Boltzmann in velocity
    % maxw_vel = (temp/pi)^(3/2) * 4*pi * (in_velocities).^2 .*
        exp(-temp*(in_velocities-shift).^2);
    maxw_exp = exp(-(in_velocities-shift).^2/(2*v_th^2));
    maxw_vel = (2*pi)^(-3/2)*v_th^-3 .* maxw_exp;

    if ~isempty(PAcenter)
```

```
22          % Gaussian in pitch angle
23          maxw_PA = 1/(PAcenter*sqrt(2*pi)) *
                exp(-(in_angles-PAwidth).^2/(2*PAcenter^2));
24          maxw_vel = maxw_vel.*maxw_PA;
25      end
26
27  end
```

B.3.2 Background/Beam Definition Structure Builder

This takes a launch period, sampling period, and structures that define the ionospheric background, secondary background, and beams, and builds a parent structure encompassing all of that. It performs a couple of simple sanity checks, and builds a vector with the start times for each segment of the beam structure, as well as the 'end' time.

`build_dyn_struct()`

```
1   function s_dyn = build_dyn_struct( launch_dt, sample_dt, s_iono, s_bg,
        s_beams )
2   %build_dyn_struct Builds a dynamic distribution definition structure.
3   % Builds a structure for feeding to dynamic_distribution().
4   % Beyond using struct(), the main function is to build the vector
5   % s_dyn.times, which has the start time of each beam, so you can do a
6   % simple find(s_dyn.times >= time & s_dyn.times < time) to figure out
7   % what beam def is active at a given time.
8
9   n_beams = length(s_beams);
10  v_times = zeros(n_beams+1,1);
11  for i=2:n_beams+1
12      v_times(i) = v_times(i-1) + s_beams{i-1}.dwell_time;
13  end
14
15  if sample_dt < 10*launch_dt
16      display('Sampling time should really be at least 10 times launch time!')
17  end
18
19  if ~isempty(find(diff(v_times) < launch_dt, 1))
20      display('Dwell time lower than launch time!  Is this really what you
            want?')
21  end
22
23  s_dyn = struct('launch_dt', launch_dt, 'sample_dt', sample_dt, ...
24      'iono', s_iono, 'bg', s_bg, 'times', v_times);
25  s_dyn.beams = s_beams;
26
27  end
```

B.3.3 Dynamic Distribution Timeslice Calculator

Takes the a time, the input data from the test particle simulation and its map structure, and a definition structure made by `build_dyn_struct()`, and returns the Maxwellian for the given time.

`dynamic_distribution.m`

```
function [ dist, sdist, n_beam ] = dynamic_distribution(time, in_dist,
    in_map, dist_def)
% Returns a distribution at a given time, for a provided 2xN list of
% particles/distribution function element centers, with velocities
% in (1,:) and pitch angles in (2,:).  Returns an N-element list which
% is values of f(vmag,pa) for each particle.

    in_vmag = in_dist(in_map.bot.v.mag,:);
    in_PA = in_dist(in_map.bot.alpha,:);

    % ionosphere parameters
    iono_def = dist_def.iono;

    % Create ionospheric distribution
    iono_dist = maxwellian(iono_def.temp, iono_def.shift, ...
        iono_def.PAcenter, iono_def.PAwidth, in_vmag, in_PA);
    iono_part = iono_dist*iono_def.n;

    % background parameters
    bg_def = dist_def.bg;

    % Create background distribution
    bg_dist = maxwellian(bg_def.temp, bg_def.shift, ...
        bg_def.PAcenter, bg_def.PAwidth, in_vmag, in_PA);
    bg_part = bg_dist*bg_def.n;

    % We'll be using segment time /. dwell_time
    i_beam = find(dist_def.times <= time, 1, 'last');
    beam_def = dist_def.beams{i_beam};
    n_beam = beam_def.n;

    if beam_def.n == 0 % BG-only case

        dist = bg_part+iono_part;
        sdist = { iono_part, bg_part, zeros(size(iono_part)) };

    else % BG + beam

        beam_dist = maxwellian(beam_def.temp, beam_def.shift, ...
            beam_def.PAcenter, beam_def.PAwidth, in_vmag, in_PA);
```

```
        beam_part = beam_dist*beam_def.n;

        dist = iono_part+bg_part+beam_part;
        sdist = { iono_part, bg_part, beam_part };

    end

end
```

B.3.4 Azimuthal Summation

The first step of reduction is to undo all that hard work `hemi_fill()` did. In gyrotropic cases the azimuthal angle has no effect on time of flight, so we can do this before we deal with any time summation issues.

This function will be run many times, and the `uniquetol()` required to get the indices of unique $v_\perp$ and $v_\parallel$ is quite slow. The result is also identical for a given set of input velocity vectors, so we can save the result, and reuse it for every `azi_sum()` in a given run. This is what `azi_sum_stash()`, with its friend `array_checksum()`, both included below, accomplish: compare input vs. stored checksums, and if it checks out, just pass back saved results. Saves and checksums are stored as persistent variables in the function-local context.

Finally, also included is a simple intermediate utility function `time_azi_sum_chain()`, which chains `dynamic_distribution()` to `azi_sum()`, for great justice.

`azi_sum.m`

```
function [ m_fperppara, out_map, m_intersect, s_uniques ] = azi_sum(in_dist,
    in_map, t_dalpha, t_domega)
% Takes a 14xN list of cells in a distribution in 3-D perp/para/azi
% space, and sums over Azimuthseseses to return a 12xM list of 2-D
% reduced distribution functions in v_perp-v_para space.  Optionally
% returns its intersection list and a structure of the unique perp
% and para values.

    % Input 14xN matrix
    % (En, vmag, vpar, vper, pa, azi)_top
    % (En, vmag, vpar, vper, pa, azi)_bottom
    % time, distN

    % Returns 12xM
    % (En, vmag, vpar, vper, pa)_top
    % (En, vmag, vpar, vper, pa)_bottom
    % time, N

    v_perp = in_dist(in_map.bot.v.perp,:);
```

```
    v_para = in_dist(in_map.bot.v.para,:);

    % get cell of tuple-matches
    [ m_intersect, s_uniques ] = azi_sum_stash(v_perp, v_para);
    n_cells = length(m_intersect);

    m_fperppara = zeros(12,n_cells);
    for i=1:n_cells
        v_indices = m_intersect{i};

        v_distN = in_dist(in_map.dist,v_indices);

        % Since these are limited to a single vperp,vpara, they all
        % have the same alpha, i.e. they're in an azimuthal ring.
        % Because that's exactly how azimuths were defined.  Thus,
        % dtheta is just 2pi/(# of points).  We can just factor that.
        t_dtheta = 2*pi/length(v_indices);
        t_sumN = sum(v_distN*t_dtheta);

        m_fperppara(:,i) = [ in_dist([ ...
            in_map.top.En in_map.top.v.mag ...
            in_map.top.v.perp in_map.top.v.para in_map.top.alpha ...
            in_map.bot.En in_map.bot.v.mag ...
            in_map.bot.v.perp in_map.bot.v.para in_map.bot.alpha ...
            in_map.time],v_indices(1)) ; t_sumN ];
            % The values from the input should be
            % identical for all v_indices()

    end

    % create new output field map
    out_map.top.En = 1; out_map.top.v.mag = 2;
    out_map.top.v.perp = 3; out_map.top.v.para = 4;
    out_map.top.alpha = 5;
    out_map.bot.En = 6; out_map.bot.v.mag = 7;
    out_map.bot.v.perp = 8; out_map.bot.v.para = 9;
    out_map.bot.alpha = 10;
    out_map.time = 11; out_map.dist = 12;

end
```

azi_sum_stash.m

```
function [ m_intersect, s_uniques ] = azi_sum_stash(v_perp,v_para)
% Finds unique (v_perp,v_para) tuples and returns the indices from the
% data that hit those tuples, i.e. a cell of arrays of data points with
% the same (v_perp,v_para), but different azimuths.
```

```
% Will cache this search for inputs which match checksums, because the
% uniquetol()s and intersections are rather slow.

    % uniquetol() and the set intersection stuff are very time
    % consuming, so we'll cache those results and a checksum.
    persistent perpcs paracs sm_intersect ss_uniques

    % First check if we've got an accurate cache.
    newperpcs = array_checksum(v_perp); % checksum of perp velocities
    newparacs = array_checksum(v_para); % checksum of para velocities

    if ~isequal(perpcs,newperpcs) || ~isequal(paracs,newparacs) ||
        isempty(sm_intersect)
        display('Rerunning uniquetol() & intersections.')
        % no stored copy or checksums were bad, must run uniquetol()s

        perpcs = newperpcs; % store checksums
        paracs = newparacs;

        [ v_vperpvals, v_vperpinds ] =
            uniquetol(v_perp,0.000001,'OutputAllIndices',true);
        [ v_vparavals, v_vparainds ] =
            uniquetol(v_para,0.000001,'OutputAllIndices',true);

        ss_uniques.v_vperpvals = v_vperpvals; ss_uniques.v_vperpinds =
            v_vperpinds;
        ss_uniques.v_vparavals = v_vparavals; ss_uniques.v_vparainds =
            v_vparainds;

        n_vperp = length(v_vperpvals);
        n_vpara = length(v_vparavals);

        % Make a grid for all possible (v_perp,v_para) tuples
        m_intersect = cell(n_vperp,n_vpara);
        m_interlen = zeros(n_vperp,n_vpara);
        for i=1:n_vperp
            parfor j=1:n_vpara

                % v_indices = intersect(v_vperpinds{i},v_vparainds{j});
                % using ismember() is faster, but still pretty slow
                m_intersect{i,j} = v_vperpinds{i}(ismember(v_vperpinds{i},
                    v_vparainds{j}));
                m_interlen(i,j) = length(m_intersect{i,j});

            end
        end

```

```
46          % flatten
47          m_intersect = reshape(m_intersect,1,[]);
48          m_interlen = reshape(m_interlen,1,[]);
49
50          % keep only points with matching cells
51          m_intersect = m_intersect(m_interlen ~= 0);
52
53          sm_intersect = m_intersect;
54      end
55
56      s_uniques = ss_uniques;
57      m_intersect = sm_intersect;
58
59  end
```

array_checksum.m

```
1   function cs = array_checksum(in)
2
3       flatsum = sum(in);
4       cs = flatsum/sum((in/flatsum).^2);
5
6   end
```

time_azi_sum_chain.m

```
1   function [ m_EVPN, smap_EVPN ] = time_azi_sum_chain(in_time, in_dist,
        in_map, dist_def)
2
3       t_dphi = 3*pi/256; % delta for co-latitude
4       t_domega = 0.001; % delta for solid angle in steradians
5
6       % generate dist at top-time t
7       t_dist = dynamic_distribution(in_time, in_dist, in_map, dist_def);
8       m_EVPAN = [ in_dist ; t_dist ]; % Tack distribution on to the rest
9       smap_EVPAN = in_map;
10      smap_EVPAN.dist = 14;
11
12      % azi_sum
13      [ m_EVPN, smap_EVPN ] = azi_sum(m_EVPAN, smap_EVPAN, t_dphi, t_domega);
14
15  end
```

B.3.5 Perpendicular Summation

Now we sum over the perpendicular velocities, to get a parallel reduced distribution function. This just straight up returns the RDF, no more structy stuff since we're combining things. I wonder how setting the bin centers arbitrarily might change things in the results...

perp_sum.m

```
function [ m_rdf, paravals ] = perp_sum(in_dist, in_map, paravals)
% Takes a 12xN distribution of cells in 2-D perp/para space,
% and sums over perp values to return a 10xM 1-D distribution.

    % Input 12xM
    % (En, vmag, vpar, vper, pa)_top
    % (En, vmag, vpar, vper, pa)_bottom
    % time, N

    v_perp = in_dist(in_map.bot.v.perp,:);
    v_para = in_dist(in_map.bot.v.para,:);
    v_N = in_dist(in_map.dist,:);

    n_para = length(paravals);

    [ para_widths, para_deltas ] = half_deltas(paravals);

    m_rdf = zeros(n_para,1);
    parfor i=1:n_para
        para = paravals(i);
        deltas = para_deltas(i:i+1);

        parainds = find(v_para >= para-deltas(1) & v_para < para+deltas(2));
        [ t_perpvals, t_perpinds ] = uniquetol(v_perp(parainds), ...
            0.00001, 'OutputAllIndices', true);
        n_perp = length(t_perpvals);

        if n_perp > 1
            % flatten the lists, making a list of all vals,
            % and a list of indices
            perpvals = [];
            for k=1:n_perp
                perpvals = [ perpvals
                    repmat(t_perpvals(k),1,length(t_perpinds{k})) ];
            end
            perpinds = vertcat(t_perpinds{:});

            [ s_perpvals, si_perpvals ] = sort(perpvals);

            % indices within this batch of parainds
```

```
            si_perpinds = perpinds(si_perpvals);
            % values of f(perp,para)
            s_distN = v_N(parainds(si_perpinds));

            % trapezoidal rule function,
            % 1/2 sum( (v_{i+1}-v_i)*(f(v_{i+1})+f(v_i))*v_i )
            delta_v = diff(s_perpvals);
            f_sums = s_distN(1:end-1) + s_distN(2:end);
            f = 0.5*sum( delta_v .* f_sums .* s_perpvals(1:end-1) );

        elseif n_perp == 1
            if length(t_perpinds) > 1
                display('Only one perp value, but multiple indices.  This
                    really shouldn''t happen!')
            end
            f = sum(v_N(parainds(t_perpinds{1})));

        else
            f = 0;

        end

        m_rdf(i) = f;
    end

end % parper_rdf()
```

B.3.6 Growth Rate Utility Script

Aaand the remainder is done in another sectioned script. The memory usage of this gets untenable for small timesteps: potentially hundreds of GB. Recoding it to not keep all data (only that which affects the current detector timeslice or whatever) would help. Making it cluster-deployable would be better, but that would take a good bit of work.

```
% Full distribution to growth rate code stack

%% Define the distribution

% Fiddle with topside distribution
fiddle = false;
if fiddle
    % Find unique energies for fiddling
    [ t_en, t_en_ind ] = uniquetol(r_mirror_EVPA(smap_EVPA.top.En,:), 0.001,
        'outputallindices', true);

```

```
    % Remove half of the energies
    t_en_find = vertcat( t_en_ind{1:2:end} );
    GR_dist = r_mirror_EVPA(:,t_en_find);
else
    GR_dist = r_mirror_EVPA;
end

GR_smap = smap_EVPA;

shortest_travel_time = min(GR_dist(GR_smap.time,:));
longest_travel_time = max(GR_dist(GR_smap.time,:));

density_const = 0.000314207783; % m_e*epsilon_0/e^2

% ionospheric background parameters
f_pe = 400000;
omega_pe = f_pe * 2*pi;
n_e = omega_pe^2*density_const;
iono_temperature = 2000; %in Kelvin

s_dyn_iono = struct('n', n_e, 'temp', iono_temperature, 'shift', 0, ...
    'PAcenter', [], 'PAwidth', []);

% secondary background parameters
maxw_temperature = 200000; % in Kelvin
t_temp_eV = maxw_temperature/11604.505;
% linearly interpolate n(T) from Table 1b in Lotko & Maggs 1981
if t_temp_eV < 137.1
    t_n_lotkomaggs = (0.44 - 0.83)/(137.1 - 62.4)*(t_temp_eV-62.4) + 0.83;
else
    t_n_lotkomaggs = (0.42 - 0.44)/(220.1 - 137.1)*(t_temp_eV-137.1) + 0.44;
end
maxw_particles = t_n_lotkomaggs * 1000000; % cm^-3 -> m^-3
maxw_shift = 0;
maxw_PAcenter = [];
maxw_PAwidth = [];

s_dyn_bg = struct('n', maxw_particles, 'temp', maxw_temperature, 'shift',
    maxw_shift, ...
    'PAcenter', maxw_PAcenter, 'PAwidth', maxw_PAwidth);

% beam parameter sets
% run 4 longest travel time is <14s
s_dyn_beams = {
    struct('n', 0, 'dwell_time', 5), ...
    struct('n', maxw_particles/50, 'dwell_time', 0.100, 'temp',
        maxw_temperature/5, 'shift', 400, 'PAcenter', [], 'PAwidth', []), ...
```

```
...%    struct('n', maxw_particles/5, 'temp', maxw_temp/16, 'shift', 300,
    'PAcenter', [], 'PAwidth', []), ...
    struct('n', 0, 'dwell_time', 5)
    };

% Builder function eats launch time, sample time,
% and the three dist structures.
s_dyn_dist = build_dyn_struct(0.001, 0.010, s_dyn_iono, s_dyn_bg,
    s_dyn_beams);

%% Sanity-check plots

java_numFmt = java.text.DecimalFormat;

% sort by energy for plotting, but we feed
% dynamic_distribution velocities and PAs
[ s_En, si_En ] = sort(GR_dist(GR_smap.bot.En,:));

h = figure(7777);
clf(h)

set(h,'position',[ 10 500 300*n_beams 400]);
suptitle('Electron Distribution Functions')

n_beams = length(s_dyn_beams);
for i=1:n_beams

    [ t_dist, s_dist ] = dynamic_distribution(s_dyn_dist.times(i), GR_dist,
        GR_smap, s_dyn_dist);

    t_width = 0.90/n_beams;
    subplot('position',[ 0.05+t_width*(i-1) 0.17 t_width 0.70 ])

    plot(s_En,t_dist(si_En),'k', s_En,s_dist{1}(si_En),'g.', ...
        s_En,s_dist{2}(si_En),'b*', s_En,s_dist{3}(si_En),'rx');
    set(gca,'fontsize',12)

    xlabel('Energy [eV]')
    foo = get(gca,'xticklabel'); foo{end}=''; set(gca,'xticklabel',foo);
    if i==1
        ylabel('$f(|v|)$','interpreter','latex');
        t_xlim = xlim; t_ylim = ylim;
    else
        set(gca,'yticklabel',[])
        xlim(t_xlim); ylim(t_ylim);
    end

```

```
100     if s_dyn_dist.beams{i}.n == 0
101         legend('Combined', [ char(java_numFmt.format(s_dyn_dist.iono.temp))
                ' K ionospheric BG' ], ...
102             [ char(java_numFmt.format(s_dyn_dist.bg.temp)) ' K secondary BG'
                    ])
103     else
104         legend('Combined', [ char(java_numFmt.format(s_dyn_dist.iono.temp))
                ' K ionospheric BG' ], ...
105             [ char(java_numFmt.format(s_dyn_dist.bg.temp)) ' K secondary BG'
                    ], ...
106             [ char(java_numFmt.format(s_dyn_dist.beams{i}.temp)) ' K, ' ...
107                 char(java_numFmt.format(s_dyn_dist.beams{i}.shift)) '
                        eV-shifted beam'])
108     end
109 end
110
111 %print('-dpng',[file_outdir '\topdist.png'])
112
113 %% azi_sum all top timesteps
114 launch_time = s_dyn_dist.launch_dt;
115 n_beams = length(s_dyn_dist.beams);
116
117 tic
118 dt_c_EVPN = cell(n_launchsteps,2);
119
120 v_launchsteps = 0:launch_time:s_dyn_dist.times(end)-launch_time;
121 n_launchsteps = length(v_launchsteps);
122
123 for i=1:n_launchsteps
124     t_time = v_launchsteps(i);
125
126     [ m_EVPN, smap_EVPN ] = time_azi_sum_chain (t_time, GR_dist, GR_smap,
            s_dyn_dist);
127
128     t_strike = m_EVPN(smap_EVPN.time, :) + t_time;
129
130     dt_c_EVPN{i,1} = m_EVPN;
131     dt_c_EVPN{i,2} = t_strike;
132
133 end
134
135 dt_m_EVPN = [dt_c_EVPN{:,1}];
136 dt_v_EVPNt = [dt_c_EVPN{:,2}];
137 toc
138
139 %% Set up for perp_sum
140 % Now we go through and filter, for bottom time t-deltat to t
```

```
141
142 sample_time = s_dyn_dist.sample_dt;
143 v_timesteps = shortest_travel_time:sample_time:(n_beams*t_dwell)-launch_time;
144 n_timesteps = length(v_timesteps);
145
146 % Find the field-aligned velocities, for use as the
147 % center points in the reduced distribution function.
148 display('Uniquetol...')
149 v_paravels = uniquetol(dt_m_EVPN(smap_EVPN.bot.v.para,
        dt_m_EVPN(smap_EVPN.bot.v.perp,:)==0));
150 display('...done.')
151
152 % Reverse to smallest magnitude first
153 v_paravels = sortmag(v_paravels);
154
155 % extend these to zero
156 n_para = length(v_paravels);
157 d_vpar = median(diff(v_paravels));
158 d_extrap = v_paravels(1):-d_vpar:0;
159
160 v_paravelx = [ flip(d_extrap(2:end)) v_paravels ];
161 n_paravelx = length(v_paravelx);
162
163 %% perp_sum all top timesteps
164
165 display('Running perp_sum()s...')
166 tic
167 dt_m_rdf = zeros(n_timesteps,n_paravelx);
168 dt_v_nrdf = zeros(n_timesteps,1);
169 for i=1:n_timesteps
170
171     t_time = v_timesteps(i);
172     % search for particles that have been 'detected' in this timeslice
173     v_timeinds = find(dt_v_EVPNt <= t_time & dt_v_EVPNt >
            t_time-sample_time);
174     m_particles = dt_m_EVPN(:,v_timeinds);
175     dt_v_nrdf(i) = size(m_particles,2);
176     m_particles(smap_EVPN.dist,:) = m_particles(smap_EVPN.dist,:) *
            launch_time/sample_time;
177
178     if length(m_particles) < 1
179         continue
180     end
181
182 %    display(['Time ' num2str(t_time) ' found ' num2str(length(m_particles))
        ' particles.'])
183
```

```
184     % reduce to parallel
185     dt_m_rdf(i,:) = perp_sum(m_particles, smap_EVPN, v_paravelx);
186 
187     % growth rate
188 %    v_gRate = para_gRate(m_rdf);
189 
190 end
191 toc
192 
193 %%
194 
195 for i=490:800 %1:n_timesteps
196 
197     t_time = v_timesteps(i);
198     % search for particles that have been 'detected' in this timeslice
199     [ t_time t_time-sample_time ]
200     v_timeinds = find(dt_v_EVPNt <= t_time & dt_v_EVPNt >
        t_time-sample_time);
201     length(v_timeinds)
202 end
203 
204 %% Full gamma vs k & time plot
205 
206 t_temp = s_dyn_dist.beams{2}.temp;
207 t_shift = s_dyn_dist.beams{2}.shift;
208 t_bg = 2000; % background ionospheric cold electron temperature [K]
209 v_bg = sqrt(3*t_bg*15156333.1);
210 
211 f_pe = 400000; % 500 kHz plasma freq.
212 omega_pe = f_pe * 2*pi;
213 t_test_temp = t_shift + t_temp/11604; % approximate beam speed
214 t_test_vel = eV2mps(t_test_temp);
215 [ ~, i_test ] = min(abs(-v_paravelx-t_test_vel))
216 v_omega_test = (1.00001:0.00001:1.01)*omega_pe;
217 v_k_test = sqrt(2/3*(v_omega_test-omega_pe)*omega_pe/v_bg^2);
218 %v_k_test = logspace(-6,20,1000);
219 v_k_test = 0.1:0.001:0.5;
220 v_omega_test = v_k_test.^2*3/2*v_bg^2/omega_pe + omega_pe;
221 n_test = length(v_k_test)
222 
223 m_gamma = zeros(n_timesteps,n_test); % timestep,kind,val/omegaind
224 m_vtest = zeros(n_timesteps,n_test,2);
225 m_kmag2 = zeros(n_timesteps,n_test,1);
226 m_df1 = zeros(n_timesteps,n_test,1);
227 m_omega_test = zeros(n_timesteps,n_test,1);
228 m_n_e = zeros(n_timesteps,n_test,1);
229 
```

```
parfor i=1:n_timesteps
    for j=1:n_test
        t_kpara = v_k_test(j);
%        t_omega_test = v_omega_test(j);

        [ m_gamma(i,j), m_vtest(i,j,:), m_kmag2(i,j), m_df1(i,j),
           m_omega_test(i,j), m_n_e(i,j) ] = growth_rate(dt_m_rdf(i,:),
           -v_paravelx, [ t_kpara 0 ], omega_pe, v_bg, t_test_vel);

    end
end

%% Launch timestep n summation, for 'this is where the beam was' plot.

dt_v_fbg = zeros(n_launchsteps,1);
dt_v_fbeam = zeros(n_launchsteps,1);
parfor i=1:n_launchsteps
    t_time = v_launchsteps(i);

    [ ~, s_dist ] = dynamic_distribution(t_time, GR_dist, GR_smap,
       s_dyn_dist);

    dt_v_fbg(i) = sum(s_dist{1}) + sum(s_dist{2});
    dt_v_fbeam(i) = sum(s_dist{3});
end


%% r/b gamma vs k,time plot

t_azi = 0;
t_el = 0;

v_upsteps = find(v_timesteps > 7.2 & v_timesteps < 8);
v_dnsteps = find(v_timesteps >= 6 & v_timesteps < 12);

%v_upsteps = find(v_timesteps);
%v_dnsteps = find(v_timesteps);

h = figure(7805);
clf
set(h, 'position', [100 50 1200 900])

p_plbase = 0.08;
p_plleft = 0.08;
p_plwidt = 0.40;
p_lpheig = 0.10;
p_vspace = 0.03;
```

```
274 p_spheig = 0.32;
275 p_hspace = 0.04;
276
277 hT = suptitle([ 'Growth Rates, $\Delta t_S =' num2str(launch_time) '$ s,
        $\Delta t_D =' num2str(sample_time) '$ s' ]);
278 set(hT,'interpreter','latex');
279
280 % -- n vs t --
281
282 nax = subplot('Position',[ ...
283     p_plleft ...
284     p_plbase+2*p_spheig+p_vspace ...
285     2*p_plwidt+p_hspace ...
286     p_lpheig ...
287 ]); ax = [ ax nax ];
288 plot(v_launchsteps,dt_v_fbeam./dt_v_fbg)
289 xlabel('Time [s]'); set(gca, 'fontsize', 12); grid on; ylim([-0.25 0.75]);
        set(gca,'ytick',[0 0.2 0.4 0.6])
290 set(gca,'XAxisLocation','top');
        ylabel('$n_{beam}/n_{bg}$','interpreter','latex')
291 set(gca,'xtick',1:15)
292
293 % -- gamma vs t --
294
295 ax = [];
296 nax = subplot('Position',[ ...
297     p_plleft ...
298     p_plbase+p_spheig ...
299     p_plwidt ...
300     p_spheig ...
301 ]); ax = [ ax nax ];
302 surf(v_timesteps(v_upsteps), v_k_test, m_gamma(v_upsteps,:).', 'edgecolor',
        'none'); colormap(rwbmap); box on;  set(gca, 'layer', 'top')
303 %caxis([-t_crange t_crange])
304 view(0,0); %set(gca,'zscale', 'log');  % ylim([min(v_k_test) 0.5])
305 %zlim([-10e3 10e3])
306 set(gca, 'fontsize', 12, 'xticklabel', []); ylabel('k'); zlabel('\gamma');
307 t_tick = get(gca,'ztick'); t_tick = t_tick(2:end); set(gca, 'ztick',t_tick);
308
309 nax = subplot('Position',[ ...
310     p_plleft+p_hspace+p_plwidt ...
311     p_plbase+p_spheig ...
312     p_plwidt ...
313     p_spheig ...
314 ]); ax = [ ax nax ];
315 surf(v_timesteps(v_dnsteps), v_k_test, m_gamma(v_dnsteps,:).', 'edgecolor',
        'none'); colormap(rwbmap); box on; set(gca, 'layer', 'top')
```

```
316 t_crange = max([caxis(ax(1)) caxis(ax(2))]);
317 caxis(ax(1),[-t_crange t_crange]); caxis(ax(2),[-t_crange t_crange])
318 view(0,0); set(gca, 'ydir', 'reverse'); %set(gca, 'zscale', 'log');%
        ylim([min(v_k_test) 0.5])
319 zlim([-10e3 10e3])
320 set(gca, 'fontsize', 12, 'xticklabel', []); %ylabel('k'); zlabel('\gamma');
321 t_tick = get(gca,'ztick'); t_tick = t_tick(2:end); set(gca, 'ztick',t_tick);
322
323 % -- k vs t --
324
325 nax = subplot('Position',[ ...
326     p_plleft ...
327     p_plbase ...
328     p_plwidt ...
329     p_spheig ...
330 ]); ax = [ ax nax ];
331 surf(v_timesteps(v_upsteps), v_k_test, m_gamma(v_upsteps,:).', 'edgecolor',
        'none'); colormap(rwbmap); box on; set(gca, 'layer', 'top')
332 %caxis([-t_crange t_crange])
333 view(0,90); ylim([min(v_k_test) 0.5])
334 %ylim([0 3e-3])
335 set(gca, 'fontsize', 12,'xtick',get(ax(1),'xtick'))
336 ylabel('k');
337 t_tick = get(gca,'yticklabel'); t_tick{end}=''; set(gca,
        'yticklabel',t_tick);
338 xlabel('Time [s]');
339
340 nax = subplot('Position',[ ...
341     p_plleft+p_hspace+p_plwidt ...
342     p_plbase ...
343     p_plwidt ...
344     p_spheig ...
345 ]); ax = [ ax nax ];
346 surf(v_timesteps(v_dnsteps), v_k_test, m_gamma(v_dnsteps,:).', 'edgecolor',
        'none'); colormap(rwbmap); box on; set(gca, 'layer', 'top')
347 t_crange = max([caxis(ax(3)) caxis(ax(4))]);
348 caxis(ax(3),[-t_crange t_crange])
349 caxis(ax(4),[-t_crange t_crange])
350 view(0,-90); set(gca, 'ydir', 'reverse'); ylim([min(v_k_test) 0.5])
351 %ylim([0 3e-3])
352 set(gca, 'fontsize', 12);  %ylabel('k')
353 t_tick = get(gca,'yticklabel'); t_tick{end}=''; set(gca,
        'yticklabel',t_tick);
354 xlabel('Time [s]');
355
356 foo = annotation('line',[0.08 0.473],[0.72 0.749]);
357 set(foo,'color','red')
```

```
uistack(foo,'bottom')
foo = annotation('line',[0.48 0.528],[0.72 0.749]);
set(foo,'color','red')
uistack(foo,'bottom')

foo = annotation('line',[0.521 0.752],[0.72 0.749]);
set(foo,'color','red');
uistack(foo,'bottom')
foo = annotation('line',[0.92 0.808],[0.72 0.749]);
set(foo,'color','red');
uistack(foo,'bottom')

foo = annotation('textbox',[0.45 0.325 0.1 0.1], ...
    'string', '...', 'horizontalalignment', 'center', 'linestyle',
       'none','fontsize', 16, 'fontweight', 'bold');

% print looks awful, use manual export
%print('-opengl','-dpng', [file_outdir '\gr.png'])

%% r/b gamma vs k,time plot, single zoom

t_azi = 0;
t_el = 0;

v_steps = find(v_timesteps > 7.2 & v_timesteps < 8);

h = figure(7805);
clf
set(h, 'position', [100 50 1200 900])

p_plbase = 0.08;
p_plleft = 0.08;
p_plwidt = 0.84;
p_lpheig = 0.10;
p_vspace = 0.03;
p_spheig = 0.32;

hT = suptitle([ 'Growth Rates, $\Delta t_S =' num2str(launch_time) '$ s,
    $\Delta t_D =' num2str(sample_time) '$ s, 100 ms Beam' ]);
set(hT,'interpreter','latex');

% -- n vs t --

nax = subplot('Position',[ ...
    p_plleft ...
    p_plbase+2*p_spheig+p_vspace ...
    p_plwidt ...
```

```
403     p_lpheig ...
404 ]); ax = [ ax nax ];
405 plot(v_launchsteps,dt_v_fbeam./dt_v_fbg)
406 xlabel('Time [s]'); set(gca, 'fontsize', 12); grid on; ylim([-0.25 0.75]);
        set(gca,'ytick',[0 0.2 0.4 0.6])
407 set(gca,'XAxisLocation','top');
        ylabel('$n_{beam}/n_{bg}$','interpreter','latex')
408 set(gca,'xtick',[ 1:7 8:10]); xlim(v_launchsteps([1 end]));
409
410 % -- gamma vs t --
411
412 ax = [];
413 nax = subplot('Position',[ ...
414     p_plleft ...
415     p_plbase+p_spheig ...
416     p_plwidt ...
417     p_spheig ...
418 ]); ax = [ ax nax ];
419 surf(v_timesteps(v_upsteps), v_k_test, m_gamma(v_upsteps,:).', 'edgecolor',
        'none'); colormap(rwbmap); box on;  set(gca, 'layer', 'top')
420 t_crange = max([abs(caxis(ax(1)))]);
421 caxis([-t_crange t_crange])
422 view(0,0); %set(gca,'zscale', 'log');  % ylim([min(v_k_test) 0.5])
423 %zlim([-10e3 10e3])
424 set(gca, 'fontsize', 12, 'xticklabel', []); ylabel('k'); zlabel('\gamma');
425 t_tick = get(gca,'ztick'); t_tick = t_tick(2:end); set(gca, 'ztick',t_tick);
426
427 % -- k vs t --
428
429 nax = subplot('Position',[ ...
430     p_plleft ...
431     p_plbase ...
432     p_plwidt ...
433     p_spheig ...
434 ]); ax = [ ax nax ];
435 surf(v_timesteps(v_upsteps), v_k_test, m_gamma(v_upsteps,:).', 'edgecolor',
        'none'); colormap(rwbmap); box on; set(gca, 'layer', 'top')
436 %t_crange = max([abs(caxis(ax(2)))]);
437 caxis([-t_crange t_crange])
438 view(0,90); ylim([min(v_k_test) 0.5])
439 %ylim([0 3e-3])
440 set(gca, 'fontsize', 12,'xtick',get(ax(1),'xtick'))
441 ylabel('k');
442 t_tick = get(gca,'yticklabel'); t_tick{end}=''; set(gca,
        'yticklabel',t_tick);
443 xlabel('Time [s]');
444
```

```
445 foo = annotation('line',[0.08 0.6805],[0.72 0.749]);
446 set(foo,'color','red')
447 uistack(foo,'bottom')
448 foo = annotation('line',[0.92 0.745],[0.72 0.749]);
449 set(foo,'color','red')
450 uistack(foo,'bottom')
451
452 foo = annotation('textbox',[0.45 0.325 0.1 0.1], ...
453     'string', '...', 'horizontalalignment', 'center', 'linestyle',
        'none','fontsize', 16, 'fontweight', 'bold');
```