1 Robotics.m

```
%Bouncing on a discretized surface and reorientation parallel to the
%tangent of the boundary

clear variables; close all; tic;

%This final version of the code makes use of bitmaps created in subfolders
%located in the "Bitmaps_n" folders (from local path)
%All units are in so called "pixels"

%Notes:
%The spatial separation is 1 pixel
%The time seperation dt should be as small as possible thus it is logical
%to make it equal to dl. However, this is not strictly speaking required
%here (the value for dt can be changed)

%Conventions used in namings:
%No capital letter: simple (1x1) variable or 1D vector
%First letter capitalized: 2D matrix
%All capitalized: 3D matrix

%Exceptions to the above are listed below:
%c_sol is (*,5) 2D matrix | notation is consistent with that of t_sol


%% Part one: Loading the necessary data into matrices, from the folders
%Select main folder:
Bitmaps_directories = dir('*Bitmaps*');
str = {Bitmaps_directories.name};
prompt = {'Select main folder to load surface, walls, and/or workspace from'};
[selection_main,OK] = listdlg('PromptString',prompt,...
                              'SelectionMode','single',...
                              'ListSize',[500 150],...
                              'ListString',str);
if(~OK); return; end; %Make sure something was selected
Bitmaps_dir = str(selection_main);

f = fopen([char(Bitmaps_dir) '/n.txt'],'r'); n = fscanf(f,'%i'); fclose(f);   ... %size of the workspace (in pixels)
f = fopen([char(Bitmaps_dir) '/rho.txt'],'r'); r = fscanf(f,'%i'); fclose(f); ... %radius of the sphere/robot (in pixels)
mainDirectory = dir(char(Bitmaps_dir));
subDirectories = find(vertcat(mainDirectory.isdir));
str = {mainDirectory(subDirectories).name};
%Remove the '.', '..' and 'Sphere_robot directories
good_str_index = (~strcmp(str,'.') & ~strcmp(str,'..') & ... ~strcmp(str,'Sphere_robot'));
str = str(good_str_index);
%Choose to enter the surface and two walls separately or all at once in a
%workspace:
```

```
choice = {'Enter surface, and walls separately', 'Enter full workspace'};
[selection,OK] = listdlg('PromptString','Make a selection',...
                         'SelectionMode','single',...
                         'ListSize',[500 100],...
                         'ListString',choice);
if(~OK); return; end; %Make sure something was selected
if(selection == 1)
    % Getting the surface matrix:
    [selection_S,OK] = listdlg('PromptString','Select a surface:',...
                         'SelectionMode','single',...
                         'ListSize',[500 500],...
                         'ListString',str);
    if(~OK); return; end; %Make sure something was selected
    surf_str = str(selection_S); %Store the name of the selected surface
    SURF=false(n,n,n);
    for i=1:n
        filename=[char(Bitmaps_dir) '/' char(surf_str) '/' int2str(i) '.png'];
        SURF(:,:,i)=imread(filename);
    end
    WORKSPACE = (SURF);
    clear('SURF');
    % Getting the 1st wall matrix:
    [selection_W1,OK] = listdlg('PromptString','Select the first wall:',...
                         'SelectionMode','single',...
                         'ListSize',[500 500],...
                         'ListString',str);
    if(~OK); return; end; %Make sure something was selected
    wall1_str = str(selection_W1); %Store the name of the selected wall 1
    WALL1=false(n,n,n);
    for i=1:n
        filename=[char(Bitmaps_dir) '/' char(wall1_str) '/' int2str(i) '.png'];
        WALL1(:,:,i)=imread(filename);
    end
    WORKSPACE = (WORKSPACE | WALL1);
    clear('WALL1');
    % Getting the 2nd wall matrix:
    [selection_W2,OK] = listdlg('PromptString','Select the second wall:',...
                         'SelectionMode','single',...
                         'ListSize',[500 500],...
                         'ListString',str);
    if(~OK); return; end; %Make sure something was selected
    wall2_str = str(selection_W2); %Store the name of the selected wall 2
    WALL2=false(n,n,n);
    for i=1:n
        filename=[char(Bitmaps_dir) '/' char(wall2_str) '/' int2str(i) '.png'];
        WALL2(:,:,i)=imread(filename);
    end
    % Creating the complete WORKSPACE matrix which includes both walls and the
    % surface.
    % Note: such a matrix could potentially have a corresponding image that
    % would be loaded (similarly to the above), in which case the above 3
    % matrices could be ignored
    WORKSPACE = (WORKSPACE | WALL2);
    clear('WALL2'); %Cleaning up for memory space and speed
else
```

```
    % Getting the full workspace matrix:
    [selection_W,OK] = listdlg('PromptString','Select a full workspace:',...
                                'SelectionMode','single',...
                                'ListSize',[500 500],...
                                'ListString',str);
    if(~OK); return; end; %Make sure something was selected
    workspace_str = str(selection_W); %Store the name of the selected wall 2
    WORKSPACE=false(n,n,n);
    for i=1:n
        filename=[char(Bitmaps_dir) '/' char(workspace_str) '/' int2str(i) '.png'];
        WORKSPACE(:,:,i)=imread(filename);
    end
end

% Getting the robot matrix!
ROBOT=false(n,n,n);
for i=1:n
    filename=[char(Bitmaps_dir) '/' 'Sphere_robot/' int2str(i) '.png'];
    ROBOT(:,:,i)=imread(filename);
end

% "Positionning" the robot on the surface
% This step is not necessary if the loaded ROBOT matrix is properly
% positioned. However, in our case we use a single sphere always located at
% the bottom center of the workspace, which can then easily be shifted around
% i.e. the default position of the bottom of the sphere with respect to the
% workspace is [n/2,n/2,0]
if(strcmp(surf_str,'Flat_surf'))
    xshift = 0; yshift = 0; zshift = n/2;
elseif(strcmp(surf_str,'Flat_Surf'))
    xshift = 0; yshift = 0; zshift = n/2;
elseif(strcmp(surf_str,'Tilted_x_Surf'));
    alpha = pi/4; h = floor(r*((1/cos(alpha))-1));
    xshift = 0; yshift = 0; zshift = n/2+h;
elseif(strcmp(surf_str,'Tilted_y_Surf'));
    alpha = pi/4; h = floor(r*((1/cos(alpha))-1));
    xshift = 0; yshift = 0; zshift = n/2+h;
elseif(strcmp(surf_str,'Tilted_xy_Surf'));
    alpha = pi/4; h = floor(r*((1/cos(alpha))-1));
    xshift = 0; yshift = 0; zshift = n/2+h;
elseif(strcmp(surf_str,'Sine_x_Surf'));
    xshift = round(pi*r*2-n/2)-r/5; yshift = 0; zshift = 2*r;
elseif(strcmp(surf_str,'Sine_y_Surf'));
    xshift = 0; yshift = round(pi*r*2-n/2); zshift = 2*r;
elseif(strcmp(surf_str,'Sine_xy_Surf'));
    xshift = round(pi*r*2-n/2); yshift = round(pi*r*2-n/2); zshift = 2*r;
elseif(strcmp(surf_str,'Gaussian_dome_Surf'));
    xshift = 0; yshift = 0; zshift = ...
        floor(r^1.5)-1-ceil((r*(sqrt(r)-7.8)+abs(r*(sqrt(r)-7.8)))/2);
elseif(strcmp(surf_str,'PhotoShop_FlatSurf'));
    xshift = 0; yshift = 0; zshift = 0;
else
    fprintf('ERROR: The selection is not a known surface\n');
    fprintf('The code might need to be updated\n');
    return
```

```
end
ROBOT=circshift(ROBOT,[xshift,yshift,zshift]);
%So, we now have 2 matching (in size) 3D matrices: ROBOT and WORKSPACE

%% Part 2: Setting up the initial conditions:
%The ROBOT has it's own coordinate system which does not necessarily match
%that of the WORKSPACE. This allows us to keep the initial conditions for
%the robot fixed, and not have to worry about strange behaviors when
%rotating since those do occur especially when x0_robot is not close to
%-pi/2
x0_robot = -pi/2 + 1e-4;
y0_robot = 0;
psi0 = 0;
%Definition of the robot/sphere in it's own reference frame
robot = @(x,y) r*[cos(x).*cos(y); cos(x).*sin(y); sin(x)+1];

%The other initial conditions:
%Everything is pixelized (even time)
dl = 1;                 %Spatial separation (1 pixel)
dt = input('Enter dt (in units of 1/r: ');
dt = dt/r;              %Time step: normalized by r so that shift=dl when rolling
t_start = 0;            %Initial time for the robot to begin moving
t_final = input('Enter final time: ');
%Adjust t_final to make sure it is an integer value time dt:
delta = (t_final-t_start)*r-floor((t_final-t_start)*r);
t_final = t_final - delta/r;
current_time = t_start; %The current time
bounces = 0;            %Number of bounces
break_val = 0;          %Used to break out of loops

%The velocity vector
vx = 0; vy = 0; vz = 0; wz = 0;
fprintf('Enter initial rotation velocities:\n');
fprintf('Note that the velocities will be automatically normalized\n');
wx = input('Enter initial rotation velocity wx = ');
wy = input('Enter initial rotation velocity wy = ');
%Normalize the velocity vector
normalization = sqrt(wx*wx + wy*wy);
wx = wx/normalization; wy = wy/normalization;
V = [vx;vy;vz;wx;wy;wz];

%The progress bar:
handle = waitbar(0,'1','Name','Running...',...
            'CreateCancelBtn',...
            'setappdata(gcbf,''canceling'',1)');
setappdata(handle,'canceling',0);

t_sol = zeros(1,round((t_final-t_start)/dt+1));
t_sol(1) = current_time;  %Initialize total time vector

%Initializing the matrix of contact points (with the surface)
Contact_pts = zeros(length(t_sol),3);
%Getting the first contact points:
%Before even starting there is not any previous contact point, and there
%should not be any hit point (otherwise an error will be caused)
```

```
prev_contact_pt = 'NaN';
guess = 'NaN';
if getappdata(handle,'canceling'); delete(handle); return; end
[contact_pt,hit_pt,error] = ...
    Get_hit_cont_pts(ROBOT,WORKSPACE,prev_contact_pt,r,guess);
if(error); return; end %Checking for the error boolean value
Contact_pts(1,:) = contact_pt';

%The initial surface parameters:
x0_surf = contact_pt(1);
y0_surf = contact_pt(2);
z0_surf = contact_pt(3);

if getappdata(handle,'canceling'); delete(handle); return; end
%Getting the slopes z_x and z_y (del_z/del_x and del_z/del_y):
z_x_old = 0; z_y_old = 0; %Values to use if "inf" is reached
[z_x,z_y] = Get_slopes(WORKSPACE,contact_pt,r,z_x_old,z_y_old);
z_x_old = z_x; z_y_old = z_y; % update the "old" values.
%Initial contact coordinates all into the initial contacts vector:
contacts0 = [x0_robot, y0_robot, x0_surf, y0_surf, psi0];
c_sol = zeros(length(t_sol),5);
c_sol(1,:) = contacts0; %Initialize total contact coordinate vector
%% Part 3: Setting up the fixed parameters:
%Some parameters never need to be updated and should be taken care of now
%Locally, the surface is a plane:
K_surf = false(2,2); %Curvature tensor
T_surf = false(1,2); %Torsion form

%The robot is known to be a sphere:
M_robot = [r, 0; 0, cos(x0_robot)];  %Metric tensor
K_robot = [-1/r, 0; 0, -1/r];        %Curvature tensor
T_robot = [0, -1/r * tan(x0_robot)]; %Torsion form

%A few 'constants' that only need to be computed once:
%(Refer to the book for notations)
R = @(psi) [cos(psi), -sin(psi); -sin(psi), -cos(psi)];
K = false(2,2);
R_cocf = @(psi) [R(psi), [0;0]; [0,0],1];
p = [0;0;0];
p_hat = [ 0 -p(3) p(2); p(3) 0 -p(1); -p(2) p(1) 0 ];

fprintf('\n\t The program is about to loop around. \n')
fprintf('All parameters have been initialized\n');
fprintf('And the current time is: %g \n', current_time);

if getappdata(handle,'canceling'); delete(handle); return; end
toc
tic
%%------------------------------------------------------------------------%%
%% The above are only about setting things up                             %%
%% Below is the part where things move                                    %%
%%------------------------------------------------------------------------%%

%%%%%%%%%%%%%%%%%%%%%%%%Now we are looping around%%%%%%%%%%%%%%%%%%%%%%%%
%**Loop #1: Loop until final allowed time:
```

```
index1 = 1; %Index for the main general loop (the "master" index)
            %Note: this index is used throughout the loops to update the
            %contacts and time matrices
while current_time < t_final
    %1
    %First, the robot is sent rolling straight into the x-direction
    %The robot only ever rolls about the wy-axis unless it is spinning to
    %reorientate, in which case it is rotating about the wz-axis
    %The orientation is only specified at the end of this comming loop by
    %the bouncing (after loop #2)

    %%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%% Part four: Move-on and rotate until the orientation needs to    %%%
    %%%                          be reset                               %%%
    % I need to loop around: after each small step the sphere coordinates
    % need to be reset, but not those of the surface

    hit = false; %The sphere has not yet hit the boundary
    index2 = index1;  %Index of loop 2

    while(~hit) %Loop #2: Loop until a boundary is hit
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        %%% Part five: Getting bigZ and bigC                           %%%
        % To avoid passing heavy functions to the function it is easier to
        % directly compute p_hat here and pass it into the function
        % Again, R_psi is computed here and passed to the function
        % Same thing for R_cocf
        R_psi = R(psi0);
        R_cocf_psi = R_cocf(psi0);
        [bigZ, bigC] = Get_bigZ_bigC(z_x, z_y, ...
            x0_robot, y0_robot, p_hat, M_robot, K_robot, ...
            T_robot, R_psi, R_cocf_psi, V);
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        %%% Part six: Now that we have bigZ and bigC we can roll:      %%%
        % The time and contacts needs to be updated
        t_sol(index2+1) = current_time + dt;
        c_sol(index2+1,:) = ((bigZ^(-1))*dt*bigC + contacts0')';
        contacts0(3) = c_sol(index2+1,3);
        contacts0(4) = c_sol(index2+1,4);
        % Do not update (so reset) the values for x0_robot, y0_robot, and
        % psi0:
        c_sol(index2+1,1) = contacts0(1);
        c_sol(index2+1,2) = contacts0(2);
        c_sol(index2+1,5) = contacts0(5);
        %Shift the robot to it's new position:
        prev_contact_pt = Contact_pts(index2,:);
        shift_x = round(c_sol(index2+1,3)) - round(prev_contact_pt(1));
        shift_y = round(c_sol(index2+1,4)) - round(prev_contact_pt(2));
        %Locally, everything is a plane. So we can use the equation of a
        %plane: z = z_x*x + z_y*y + c => dz = z_x*dx + z_y*dy
        shift_z = round(z_x*shift_x + z_y*shift_y);
        ROBOT = circshift(ROBOT,[shift_x, shift_y, shift_z]);
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        %%% Part seven: check whether a boundary was hit:              %%%
```

```
        guess = ...
            round([c_sol(index2+1,3),c_sol(index2+1,4),prev_contact_pt(3)+shift_z]);
        if getappdata(handle,'canceling'); delete(handle); return; end
        percentage = round(current_time/t_final*1e4)/100;
        waitbar((current_time-t_start)/t_final,handle,sprintf('%1.2f ...
            %%',percentage));
        [contact_pt,hit_pt,error] = ...
            Get_hit_cont_pts(ROBOT,WORKSPACE,prev_contact_pt,r,guess);
        if(error); return; end %Checking for the error boolean value
        Contact_pts(index2+1,:) = contact_pt';
        %Getting the slopes z_x and z_y (del_z/del_x and del_z/del_y):
        [z_x,z_y] = Get_slopes(WORKSPACE,contact_pt,r,z_x_old,z_y_old);
        z_x_old = z_x; z_y_old = z_y; % update the "old" values.
        if(shift_x>r/10 || shift_y>r/10 || shift_z>r/10)
            fprintf('shift values:\t %i \t %i \t %i\n', shift_x, shift_y, shift_z);
            fprintf('Loop 2, current time: %f \n', current_time);
            fprintf('The robot might be \"jumping\" around...\n');
        end

        %Making sure there is a hit point before trying to update it
        if(~strcmp(hit_pt,'NaN'))
            bounces = bounces + 1;
            Hit_pts(bounces,:) = hit_pt';
            hit = true;
            fprintf('\t The boundary was hit at time: %g\n',current_time);
        end
        %We need to reset a few values that are used throughout the loops
        x0_robot = contacts0(1);
        y0_robot = contacts0(2);
        x0_surf = contacts0(3);
        y0_surf = contacts0(4);
        psi0 = contacts0(5);
        % Update the index:
        index2 = index2 + 1;
        % Update the time:
        current_time = current_time + dt;
        %Break out if max time is reached
        if(current_time >= t_final); break; end
    end %End of loop #2 (loop until boundary is hit)

    if getappdata(handle,'canceling'); delete(handle); return; end
    %update index1:
    index1 = index2;
    %Again, break out if max time is reached:
    if(current_time >= t_final); break; end
    %%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%% Part eight: figure out the angle of rotation:                   %%%
    prev_contact_pt = Contact_pts(index1-1,:);
    current_contact_pt = Contact_pts(index1,:);
    direction_vect = (current_contact_pt-prev_contact_pt)';
    x_hit = Hit_pts(end,1); y_hit = Hit_pts(end,2); z_hit = Hit_pts(end,3);

    %Getting the normal vector to the wall
    i = 0; done = false; tmp_found1 = false; need_break1 = false;
```

```
    for j=0:r/10
        for k=0:r/10
            if(k==0 && j==0)
                %Do nothing
            elseif(x_hit-i<1 || y_hit-j<1 || z_hit-k<1)
                need_break1 = true;
                break;
            elseif(WORKSPACE(x_hit,y_hit+j,z_hit+k))
                if(i*i+j*j+k*k >= r*r/100)
                    done = true;
                    break
                else
                    tmp_v1 = [i;j;k];
                    tmp_found1 = true;
                end
            elseif(WORKSPACE(x_hit,y_hit-j,z_hit+k))
                if(i*i+j*j+k*k >= r*r/100)
                    j = -j;
                    done = true;
                    break
                else
                    j = -j;
                    tmp_v1 = [i;j;k];
                    tmp_found1 = true;
                end
            elseif(WORKSPACE(x_hit,y_hit+j,z_hit-k))
                if(i*i+j*j+k*k >= r*r/100)
                    k = -k;
                    done = true;
                    break
                else
                    k = -k;
                    tmp_v1 = [i;j;k];
                    tmp_found1 = true;
                end
            elseif(WORKSPACE(x_hit,y_hit-j,z_hit-k))
                if(i*i+j*j+k*k >= r*r/100)
                    j = -j; k = -k;
                    done = true;
                    break
                else
                    j = -j; k = -k;
                    tmp_v1 = [i;j;k];
                    tmp_found1 = true;
                end
            end
        end
        if(need_break1); break; end;
        if(done); break; end;
    end
    if(done)
        v1 = [i;j;k];
    elseif(tmp_found1)
        v1 = tmp_v1;
    else
```

```
        v1 = 'NaN';
    end
    j = 0; done = false; tmp_found2 = false; need_break2 = false;
    for i=0:r/10
        for k=0:r/10
            if(k==0 && i==0)
                %Do nothing
            elseif(x_hit-i<1 || y_hit-j<1 || z_hit-k<1)
                need_break2 = true;
                break;
            elseif(WORKSPACE(x_hit+i,y_hit,z_hit+k))
                if(i*i+j*j+k*k >= r*r/100)
                    done = true;
                    break
                else
                    tmp_v2 = [i;j;k];
                    tmp_found2 = true;
                end
            elseif(WORKSPACE(x_hit-i,y_hit,z_hit+k))
                if(i*i+j*j+k*k >= r*r/100)
                    i = -i;
                    done = true;
                    break
                else
                    i = -i;
                    tmp_v2 = [i;j;k];
                    tmp_found2 = true;
                end
            elseif(WORKSPACE(x_hit+i,y_hit,z_hit-k))
                if(i*i+j*j+k*k >= r*r/100)
                    k = -k;
                    done = true;
                    break
                else
                    k = -k;
                    tmp_v2 = [i;j;k];
                    tmp_found2 = true;
                end
            elseif(WORKSPACE(x_hit-i,y_hit,z_hit-k))
                if(i*i+j*j+k*k >= r*r/100)
                    i = -i; k = -k;
                    done = true;
                    break
                else
                    i = -i; k = -k;
                    tmp_v2 = [i;j;k];
                    tmp_found2 = true;
                end
            end
        end
        if(need_break2); break; end;
        if(done); break; end;
    end
    if(done)
        v2 = [i;j;k];
```

```
    elseif(tmp_found2)
        v2 = tmp_v2;
    else
        v2 = 'NaN';
    end
    k = 0; done = false; tmp_found3 = false; need_break3 = false;
    for i=0:r/10
        for j=0:r/10
            if(j==0 && i==0)
                %Do nothing
            elseif(x_hit-i<1 || y_hit-j<1 || z_hit-k<1)
                need_break3 = true;
                break;
            elseif(WORKSPACE(x_hit+i,y_hit+j,z_hit))
                if(i*i+j*j+k*k >= r*r/100)
                    done = true;
                    break
                else
                    tmp_v3 = [i;j;k];
                    tmp_found3 = true;
                end
            elseif(WORKSPACE(x_hit-i,y_hit+j,z_hit))
                if(i*i+j*j+k*k >= r*r/100)
                    i = -i;
                    done = true;
                    break
                else
                    i = -i;
                    tmp_v3 = [i;j;k];
                    tmp_found3 = true;
                end
            elseif(WORKSPACE(x_hit+i,y_hit-j,z_hit))
                if(i*i+j*j+k*k >= r*r/100)
                    j = -j;
                    done = true;
                    break
                else
                    j = -j;
                    tmp_v3 = [i;j;k];
                    tmp_found3 = true;
                end
            elseif(WORKSPACE(x_hit-i,y_hit-j,z_hit))
                if(i*i+j*j+k*k >= r*r/100)
                    i = -i; j = -j;
                    done = true;
                    break
                else
                    i = -i; j = -j;
                    tmp_v3 = [i;j;k];
                    tmp_found3 = true;
                end
            end
        end
        if(need_break3); break; end;
        if(done); break; end;
```

```
    end
    if(done)
        v3 = [i;j;k];
    elseif(tmp_found3)
        v3 = tmp_v3;
    else
        v3 = 'NaN';
    end
    if(~strcmp(v1,'NaN')&&~strcmp(v2,'NaN')&&~strcmp(v3,'NaN'))
        norm_wall1 = cross(v1,v2);
        norm_wall2 = cross(v1,v3);
        norm_wall3 = cross(v2,v3);
        norm_walls = [norm_wall1,norm_wall2,norm_wall3];
        norms = [norm(norm_wall1),norm(norm_wall2),norm(norm_wall3)];
        index_norms = find(abs(norms - max(norms)) < 1e-5,1);
        norm_wall = norm_walls(:,index_norms);
    elseif(~strcmp(v1,'NaN')&&~strcmp(v2,'NaN'))
        norm_wall = cross(v1,v2);
    elseif(~strcmp(v1,'NaN')&&~strcmp(v3,'NaN'))
        norm_wall = cross(v1,v3);
    elseif(~strcmp(v2,'NaN')&&~strcmp(v3,'NaN'))
        norm_wall = cross(v2,v3);
    else
        fprintf('ERROR: The normal to the wall cannot be computed\n');
        return
    end
    if(norm(norm_wall)<1e-5);
        fprintf('ERROR: The normal to the wall is too close to (0,0,0)\n');
        return
    end
    %The normalizing the normal vector to the wall:
    norm_wall = norm_wall/norm(norm_wall);
    if(dot(direction_vect,norm_wall)<0); norm_wall = -norm_wall; end
    %The bouncing vector (the new direction):
    bounce_vector_0 = direction_vect-2*dot(direction_vect,norm_wall)*norm_wall;
    bounce_vector = bounce_vector_0;
    bounce_vector(3) = 0; %We don't want a z-direction
    bounce_vector = bounce_vector/norm(bounce_vector); %normalize
    %The reorientation vector (tangent to the surface):
    norm_wall = -norm_wall;
    reorientation_vector = ...
        bounce_vector_0-dot(bounce_vector_0,norm_wall)*norm_wall;
    reorientation_vector(3) = 0; %We don't want a z-direction
    if(reorientation_vector(1) == 0 && reorientation_vector(2) == 0)
        reorientation_vector(1) = norm_wall(2)*sign(wy);
        reorientation_vector(2) = -norm_wall(1)*sign(wy);
    end
    reorientation_vector = reorientation_vector/norm(reorientation_vector); ...
        %normalize
    if(isnan(reorientation_vector(1)) || isnan(reorientation_vector(2)))
        fprintf('ERROR: There was an issue computing the reorientation vector\n');
        return
    end
    wx = -bounce_vector(2);
    wy = bounce_vector(1);
```

```
    V = [0;0;0;wx;wy;0];

    %Check we are not going over time:
    if(current_time >= t_final); break; end

    if getappdata(handle,'canceling'); delete(handle); return; end
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%% Part nine: move away from the boundary                          %%%
    move = 1/(5*dt);%r/5;
    for index3 = index1:index1+move %Loop #3: Loop a little to move away
                                    %from boundary
        %Again we are rolling:
        R_psi = R(psi0);
        R_cocf_psi = R_cocf(psi0);
        [bigZ, bigC] = Get_bigZ_bigC(z_x, z_y, ...
            x0_robot, y0_robot, p_hat, M_robot, K_robot, ...
            T_robot, R_psi, R_cocf_psi, V);
        current_time = current_time + dt;
        t_sol(index3+1) = current_time;
        c_sol(index3+1,:) = ((bigZ^(-1))*dt*bigC + contacts0')';
        c_sol(index3+1,1) = contacts0(1);
        c_sol(index3+1,2) = contacts0(2);
        contacts0(3) = c_sol(index3+1,3);
        contacts0(4) = c_sol(index3+1,4);
        c_sol(index3+1,5) = contacts0(5);
        %Shift the robot to it's new position:
        prev_contact_pt = Contact_pts(index3,:);
        shift_x = round(c_sol(index3+1,3)) - round(prev_contact_pt(1));
        shift_y = round(c_sol(index3+1,4)) - round(prev_contact_pt(2));

        %Locally, everything is a plane. So we can use the equation of a
        %plane: z = z_x*x + z_y*y + c => dz = z_x*dx + z_y*dy
        shift_z = round(z_x*shift_x + z_y*shift_y);
        if(shift_x>r/10 || shift_y>r/10 || shift_z>r/10)
            fprintf('shift values:\t %i \t %i \t %i\n', shift_x, shift_y, shift_z);
            fprintf('Loop 3, current time: %f \n', current_time);
            fprintf('The robot might be \"jumping\" around...\n');
        end
        ROBOT = circshift(ROBOT,[shift_x, shift_y, shift_z]);
        %Check we are not going over time
        if(current_time >= t_final); break; end
        %Check whether a boundary was hit.
        %In theory there should be enough room so that it doesn't.
        %If a boundary is hit an error message will be displayed.
        guess = ...
            round([c_sol(index3+1,3),c_sol(index3+1,4),prev_contact_pt(3)+shift_z]);
        if getappdata(handle,'canceling'); delete(handle); return; end
        percentage = round(current_time/t_final*1e4)/100;
        waitbar((current_time-t_start)/t_final,handle,sprintf('%1.2f ...
            %%',percentage));
        [contact_pt,hit_pt,error] = ...
            Get_hit_cont_pts(ROBOT,WORKSPACE,prev_contact_pt,r,guess);
        if(error); return; end %Checking for the error boolean value
        Contact_pts(index3+1,:) = contact_pt';

```

```
        %Getting the slopes z_x and z_y (del_z/del_x and del_z/del_y):
        [z_x,z_y] = Get_slopes(WORKSPACE,contact_pt,r,z_x_old,z_y_old);
        z_x_old = z_x; z_y_old = z_y; % update the "old" values.
        %In the event where there is a hit point then an error message is
        %displayed and the program jumps to plotting
        if(~strcmp(hit_pt,'NaN'))
            fprintf('WARNING: The boundary was hit before reorientation\n');
            break_val = 0;
            %break;
        end
        %We need to reset a few values that are used throughout the loops
        x0_robot = contacts0(1);
        y0_robot = contacts0(2);
        x0_surf = contacts0(3);
        y0_surf = contacts0(4);
        psi0 = contacts0(5);
    end %End of loop #3 (moving away from the boundary)

    if(break_val); break; end
    %Updating index1:
    index1 = index3+1;
    %Check we are not going over time:
    if(current_time >= t_final); break; end

    if getappdata(handle,'canceling'); delete(handle); return; end
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%% Part ten: Reorientating the robot                             %%%
    % The robot now has to reorientate to be in the direction of the
    % tangent line to the wall
    wx = -reorientation_vector(2);
    wy = reorientation_vector(1);
    V = [0; 0; 0; wx; wy; 0];

    % The code can now move back up to allow for the robot to roll in the
    % new direction
end %End of loop #1 (loop until final allowed time)

if getappdata(handle,'canceling'); delete(handle); return; end
percentage = round(current_time/t_final*1e4)/100;
waitbar((current_time-t_start)/t_final,handle,sprintf('%1.2f %%',percentage));
%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% Final part: Plot                                                     %%%
%The first plot is the contact coordinates
figure(1); clf;
%Make the psi angle between -pi and pi as well as y_robot
%contacts_total(:,2) = mod(contacts_total(:,2),2*pi) - pi;
%contacts_total(:,5) = mod(contacts_total(:,5),2*pi) - pi;
plot(t_sol, c_sol(:,1), 'b', ... %x_robot
     t_sol, c_sol(:,2), 'r', ... %y_robot
     t_sol, c_sol(:,3), 'g', ... %x_surf
     t_sol, c_sol(:,4), 'c', ... %y_surf
     t_sol, c_sol(:,5), 'm', ... %psi
     t_sol, 0);                            %The t-axis
minix = min(t_sol);                  %Minimum time value
```

```
maxix = max(t_sol);                 %Maximum time value
miniy = min(min(c_sol));        %Minimum contacts value
maxiy = max(max(c_sol));        %Maximum contacts value
%Note: the two min/max are necessary because contacts_total is a matrix
%and not just a vector
range = [minix*1.1, maxix*1.1, miniy*1.1, maxiy*1.1];
%Set the range of the y-axis:
axis(range)
legend('x_{robot}', 'y_{robot}', 'x_{surface}', 'y_{surface}', ...
    'angle \psi', 'Location', 'best');
title('Contact coordinates of rolling robot');
xlabel('time');

%Figure 1, but split in 2 to see better.
figure(2); clf; hold on;
subplot(2,1,1)
plot(t_sol, c_sol(:,1), 'b', ...
     t_sol, c_sol(:,2), 'r', ...
     t_sol, c_sol(:,5), 'm', ...
     t_sol, 0);
legend('x_{robot}', 'y_{robot}', 'angle \psi', 'Location', 'best');
title('Robot Contact coordinates of rolling robot');
xlabel('time');
subplot(2,1,2)
plot(t_sol, c_sol(:,3), 'g', ...
     t_sol, c_sol(:,4), 'c', ...
     t_sol, 0);
legend('x_{surface}', 'y_{surface}', 'Location', 'best');
title('Surface Contact coordinates of rolling robot');
xlabel('time');

%The third plot is the robot with the final "workspace" coordinates (not
%centered at 0)
%Since the robot is always a sphere, this plot is kind of useless, but just
%in case something goes wrong it is nice to have it.
figure(3); clf; hold on;
[x_robot,y_robot,z_robot]=ind2sub(size(ROBOT), find(ROBOT));
plot3(x_robot,y_robot,z_robot,'b.');
%Over plot the contact point:
contact_robot = robot(c_sol(:,1)', c_sol(:,2)');
contact_robot = contact_robot + (n/2+1)*ones(size(contact_robot));
%plot3(contact_robot(1), contact_robot(2), contact_robot(3), ...
    %'r.','markersize', 6);
xlabel('x-axis'); ylabel('y-axis'); zlabel('z-axis')
title('Mapping x_{robot} and y_{robot} back to the sphere rolling on the plane')
view(160,10); axis equal; box on; colormap cool;
hold off;

%The fourth plot is the surface with the two boundaries as well as the
%contacts over plotted
figure(4); clf; hold on;
[x_workspace,y_workspace,z_workspace]=ind2sub(size(WORKSPACE), find(WORKSPACE));
plot3(x_workspace,y_workspace,z_workspace,'b.','markersize',0.1);
xlabel('x-axis'); ylabel('y-axis'); zlabel('z-axis')
%Over plot the contact points:
```

```
x_contact_surf = Contact_pts(:,1);
y_contact_surf = Contact_pts(:,2);
z_contact_surf = Contact_pts(:,3);
plot3(x_contact_surf, y_contact_surf, z_contact_surf, ...
    'r.','markersize', 6);
title('Mapping x_{surface} and y_{surface} back to the surface')
view(-15,65); axis equal; box on; colormap cool;
hold off;

%The fifth plot is a nicer version of the fourth plot:
figure(5); clf; hold on;
clear('ROBOT'); clear('WORKSPACE');
if(selection == 1) %Surface and walls were selected separately
    %Plotting the surface:
    Surf=false(n,n);
    Z_surf = zeros(n,n);
    for i=1:n
        filename=[char(Bitmaps_dir) '/' char(surf_str) '/' int2str(i) '.png'];
        Surf(:,:)=imread(filename);
        Z_surf(Surf') = i;
    end
    [X_surf,Y_surf] = meshgrid((1:n),(1:n));
    mesh(X_surf,Y_surf,Z_surf);
    %Plotting the first wall:
    Wall1=false(n,n);
    Z_wall1 = zeros(n,n);
    for i=1:n
        filename=[char(Bitmaps_dir) '/' char(wall1_str) '/' int2str(i) '.png'];
        Wall1(:,:)=imread(filename);
        Z_wall1(Wall1') = i;
    end
    [X_wall1,Y_wall1] = meshgrid((1:n),(1:n));
    waterfall(X_wall1,Y_wall1,Z_wall1);
    %Plotting the second wall:
    Wall2=false(n,n);
    Z_wall2 = zeros(n,n);
    for i=1:n
        filename=[char(Bitmaps_dir) '/' char(wall2_str) '/' int2str(i) '.png'];
        Wall2(:,:)=imread(filename);
        Z_wall2(Wall2') = i;
    end
    [X_wall2,Y_wall2] = meshgrid((1:n),(1:n));
    waterfall(X_wall2,Y_wall2,Z_wall2);

else %The full workspace was entered at once
    Workspace = false(n,n);
    Z_workspace = zeros(n,n);
    for i=1:n
        filename=[char(Bitmaps_dir) '/' char(workspace_str) '/' int2str(i) '.png'];
        Workspace(:,:)=imread(filename);
        Z_workspace(Workspace') = i;
    end
    [X_workspace,Y_workspace] = meshgrid((1:n),(1:n));
    waterfall(X_workspace,Y_workspace,Z_workspace);
end
```

```

%Over plot the contact points:
plot3(x_contact_surf, y_contact_surf, z_contact_surf, ...
    'r.','markersize', 6);
title('Mapping x_{surface} and y_{surface} back to the surface')
view(-15,65); axis equal; box on; colormap cool;
xlabel('x-axis'); ylabel('y-axis'); zlabel('z-axis')
hold off;

fprintf('The robot bounced a total of %i times \n', bounces);
delete(handle);
clear variables;
toc
```

2 Get_bigZ_bigC.m

```
function [bigZ, bigC] = Get_bigZ_bigC(z_x, z_y, ...
    x0_robot, y0_robot, p_hat, M_robot, K_robot, T_robot, ...
    R_psi, R_cocf_psi, V)
%Returns the big Z matrix as well as the big C (right hand side) vector

% - x0_robot and y0_robot are the current values for the x and y parameters
% of the robot
% - p_hat is used especially when computing Ad_g matrices and it is easier
% to have it as a parameter passed to the function rather than compute it
% here
% M_robot is the metric tensor of the robot
% K_robot is the curvature tensor of the robot
% T_robot is the torsion form of the robot
% R_psi is another matrix necessary for the computations
% R_cocf_psi is there for the same reason as R_psi
% V is the velocity vector

%% The surface:
% The surface is approximated to a plane:
% @(x,y) [x+cst; y+cst; z_x*x + z_y*y + cst]

%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% Initializing a few matrices and vectors before getting              %%%
%%%             to rewriting the system of equations                    %%%
I_surf = [1+z_x^2, z_x*z_y; z_x*z_y, 1+z_y^2]; %The first fundamental form
M_surf = sqrtm(I_surf);                        %The metric tensor
R_fcf = [-sin(x0_robot)*cos(y0_robot), -sin(y0_robot), -cos(x0_robot)*cos(y0_robot)
         -sin(x0_robot)*sin(y0_robot),  cos(y0_robot), -cos(x0_robot)*sin(y0_robot)
         cos(x0_robot)                      0               -sin(x0_robot)];
Adg_fcf = [R_fcf, p_hat*R_fcf; zeros(3), R_fcf];


%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% Part three:   Rewriting the system of equations                     %%%
```

```
A = (M_robot^(-1))*(K_robot)^(-1);
B = (M_surf^(-1))*R_psi*(K_robot)^(-1);
C_f = T_robot*M_robot;
RR = [R_cocf_psi, zeros(3); zeros(3), R_cocf_psi];
J = [M_surf; zeros(4,2)];
L = eye(6)-((Adg_fcf)^(-2));
Q1 = [0 0 0 0 -1 0
    0 0 0 1 0 0];
Q2 = [1 0 0 0 0 0
    0 1 0 0 0 0];
Q3 = [0 0 0 0 0 1];
Q4 = [0; 0; 0; 0; 0; 1];
S1 = Q1*L*RR*J;
S2 = Q2*L*RR*J;
S3 = Q3*L*RR*J;
T1 = Q1*L*Q4;
T2 = Q2*L*Q4;
T3 = Q3*L*Q4;
V1 = Q1*((Adg_fcf)^(-1))*V;
V2 = Q2*((Adg_fcf)^(-1))*V;
V3 = Q3*((Adg_fcf)^(-1))*V;


%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% Part four:  Getting the big Z matrix and the big C vector          %%%
bigZ = [eye(2), -A*S1, -A*T1
        zeros(2), eye(2)-B*(S1+K_robot*S2), -B*(T1+K_robot*T2)
        -C_f, -S3, 1-T3];
bigC = [A*V1; B*(V1+K_robot*V2); V3];


end
```

3 Get_hit_contact_pts.m

```
function [contact_pt,hit_pt,error] = Get_hit_cont_pts(ROBOT,WORKSPACE,...
    prev_contact_pt,r,guess)
done = false;
trial = 0; %Trials to shift the robot around
while(~done)
    done = true; %Shouldn't need to loop around
    %Returns the current contact point (with the surface) along with the
    %current hit point (if such hit point exists) as well as with a boolean
    %type error variable

    error = 0; %If all goes well, there is no error.
    num_contacts = 1; %The 'theoretical' number of contact points (1 or 2)

    %Because of the irregularities in the bitmaps (one cannot make a
    %perfect sphere out of cubes), there might be a few contact points
    %where in theory (with a perfect sphere) there would only be one; hence
```

```
    %the distinction between num_contacts (the 'theoretical' number of
    %contact points) and the true number of contact points found.
    [contacts_x,contacts_y,contacts_z] = ind2sub(size(ROBOT),find(ROBOT ...
        & WORKSPACE));
    if(length(contacts_x)~=length(contacts_y) || ...
            length(contacts_y)~=length(contacts_z) ...
            || length(contacts_z)~=length(contacts_x))
        fprinf('WARNING: vectors length differ in contact matrix\n');
    end
    %Vector length, and tolerance level of spacing beween points that
    %should in theory be touching each other
    len = length(contacts_x); delta = r/10;
    %Checking that there is at least one contact point (the robot is not
    %flying around).
    %If we get unlucky and the sphere does not land right on the surface,
    %we can try and move it very slightly.
    if(len < 1)
        if(trial > 0)
            fprintf('ERROR: contact has been lost\n');
            contact_pt = zeros(3,1); hit_pt = zeros(3,1); error = 1;
            return
        end
        delta_move = round(r/10);
        finished = false;
        for i=0:delta_move
            for j=0:delta_move
                for k=0:delta_move
                    tmp_ROBOT = circshift(ROBOT,[i, j, k]);
                    tmp_index = find(tmp_ROBOT & WORKSPACE,1);
                    if(~isempty(tmp_index))
                        finished = true;
                        break;
                    end
                    tmp_ROBOT = circshift(ROBOT,[-i, j, k]);
                    tmp_index = find(tmp_ROBOT & WORKSPACE,1);
                    if(~isempty(tmp_index))
                        finished = true;
                        i = -i;
                        break;
                    end
                    tmp_ROBOT = circshift(ROBOT,[i, -j, k]);
                    tmp_index = find(tmp_ROBOT & WORKSPACE,1);
                    if(~isempty(tmp_index))
                        finished = true;
                        j = -j;
                        break;
                    end
                    tmp_ROBOT = circshift(ROBOT,[i, j, -k]);
                    tmp_index = find(tmp_ROBOT & WORKSPACE,1);
                    if(~isempty(tmp_index))
                        finished = true;
                        k = -k;
                        break;
                    end
                    tmp_ROBOT = circshift(ROBOT,[i, -j, -k]);
```

```
                    tmp_index = find(tmp_ROBOT & WORKSPACE,1);
                    if(~isempty(tmp_index))
                        finished = true;
                        j = -j; k = -k;
                        break;
                    end
                    tmp_ROBOT = circshift(ROBOT,[-i, j, -k]);
                    tmp_index = find(tmp_ROBOT & WORKSPACE,1);
                    if(~isempty(tmp_index))
                        finished = true;
                        i = -i; k = -k;
                        break;
                    end
                    tmp_ROBOT = circshift(ROBOT,[-i, -j, k]);
                    tmp_index = find(tmp_ROBOT & WORKSPACE,1);
                    if(~isempty(tmp_index))
                        finished = true;
                        i = -i; j = -j;
                        break;
                    end
                    tmp_ROBOT = circshift(ROBOT,[-i, -j, -k]);
                    tmp_index = find(tmp_ROBOT & WORKSPACE,1);
                    if(~isempty(tmp_index))
                        finished = true;
                        i = -i; j = -j; k = -k;
                        break;
                    end
                end
                if(finished); break; end;
            end
            if(finished); break; end;
        end
        if(~finished);
            fprintf('ERROR: No contact has been found within range\n');
            contact_pt = zeros(3,1); hit_pt = zeros(3,1); error = 1;
            return
        end
        shift_x = i;
        shift_y = j;
        shift_z = k;
        ROBOT = circshift(ROBOT,[shift_x, shift_y, shift_z]);
        done = false;
        trial = trial + 1;
    else
        jump_index = 0; %To track where the discontinuity is located
        for i=1:len-1
            if(abs(contacts_x(i+1)-contacts_x(i)) > delta ...
                    && abs(contacts_y(i+1)-contacts_y(i)) > delta ...
                    && abs(contacts_z(i+1)-contacts_z(i)) > delta)
                jump_index = i;
                num_contacts = num_contacts + 1; %2 contacts/hit
            end
        end
        %Checking that there are no more than 2 theoretical contact points:
        if(num_contacts > 2)
```

```
            fprintf('ERROR: there are more than 2 theoretical contact points\n');
            fprintf('\t\t num_contacts = %i\n',num_contacts);
            contact_pt = zeros(3,1); hit_pt = zeros(3,1); error = 1;
            return
        end
        %There are two cases: either there is 1 theoretical contact point
        %or there are two. Those must be taken care of separately.
        if(num_contacts == 1) %Only 1 contact point. Note: jump_index == 0
            if(~strcmp(guess,'NaN') ...
                    && WORKSPACE(guess(1),guess(2),guess(3)) ...
                    && ROBOT(guess(1),guess(2),guess(3)))
                contact_pt = guess;
            else
                %Where the theoretical contact point is located in the
                %contact vectors:
                contact_index = ceil(len/2);
                contact_pt = [contacts_x(contact_index); ...
                   contacts_y(contact_index); contacts_z(contact_index)];
            end
            hit_pt = 'NaN';
        else %2 contact points (more complicated because I need to
            %differentiate between the contact vectors and the hit vectors

            %Making sure a second contact point is allowed:
            if(strcmp(prev_contact_pt,'NaN'))
                fprintf('ERROR: there cannot be a second contact point at this ...
                    point \n');
                contact_pt = zeros(3,1); hit_pt = zeros(3,1); error = 1;
                return
            end

            %The first part of the contacts vectors
            contacts_x1 = contacts_x(1:jump_index);
            contacts_y1 = contacts_y(1:jump_index);
            contacts_z1 = contacts_z(1:jump_index);
            len1 = length(contacts_x1);
            contact_index1 = ceil(len1/2);
            contact_pt1 = [contacts_x1(contact_index1); ...
                contacts_y1(contact_index1); contacts_z1(contact_index1)];
            %The second part of the contacts vectors
            contacts_x2 = contacts_x(jump_index+1:end);
            contacts_y2 = contacts_y(jump_index+1:end);
            contacts_z2 = contacts_z(jump_index+1:end);
            len2 = length(contacts_x2);
            contact_index2 = ceil(len2/2);
            contact_pt2 = [contacts_x2(contact_index2); ...
                contacts_y2(contact_index2); contacts_z2(contact_index2)];
            %Find which contact_pt is the real one and which is the hit_pt
            d1 = sum((prev_contact_pt-contact_pt1').^2);
            d2 = sum((prev_contact_pt-contact_pt2').^2);
            if(d1<d2) %contact_pt1 is closer to the previous contact point)
                contact_pt = contact_pt1;
                hit_pt = contact_pt2;
            else
                contact_pt = contact_pt2;
```

```
                hit_pt = contact_pt1;
            end
        end
    end
end
end
```

4 Get_slopes.m

```
function [z_x, z_y]=Get_slopes(WORKSPACE, contact_pt, r, z_x_old, z_y_old)
%Returns the partial derivatives (slopes) of the tangent plane at the
%contact_pt
x = contact_pt(1); y = contact_pt(2); z = contact_pt(3);
i = 0; j = 0; k1 = 1; k2 = 1; done1 = false; done2 = false;
while(~done1)
    if(WORKSPACE(x+i,y,z+k1))
        done1 = true;
    elseif(WORKSPACE(x-i,y,z+k1))
        i = -i; done1 = true;
    elseif(WORKSPACE(x+i,y,z-k1))
        k1 = -k1; done1 = true;
    elseif(WORKSPACE(x-i,y,z-k1))
        i = -i; k1 = -k1; done1 = true;
    else
        k1 = k1+1;
        if(k1 > r/5); k1 = 0; i = i+1; end
    end
    if(i > r/5)
        fprintf('WARNING: Distance between points is large\n');
    end
end
z_x = k1/i;

while(~done2)
    if(WORKSPACE(x,y+j,z+k2))
        done2 = true;
    elseif(WORKSPACE(x,y-j,z+k2))
        j = -j; done2 = true;
    elseif(WORKSPACE(x,y+j,z-k2))
        k2 = -k2; done2 = true;
    elseif(WORKSPACE(x,y-j,z-k2))
        j = -j; k2 = -k2; done2 = true;
    else
        k2 = k2+1;
        if(k2 > r/5); k2 = 0; j = j+1; end
    end
    if(k2 > r/5)
        fprintf('WARNING: Distance between points is large\n');
    end
end
z_y = k2/j;
if(i == 0)
```

```
    fprintf('WARNING: z_x = Inf, z_x was not updated\n');
    z_x = z_x_old;
end
if(j == 0)
    fprintf('WARNING: z_y = Inf, z_y was not updated\n');
    z_y = z_y_old;
end

end
```

5 Bitmaps.m

```
clear ('variables'); close all;

%The "size" of the workspace
%When dealing with a flat plane, this corresponds to the length of a square
%in which the sphere is located at the center
size_workspace = 10;
%The number of pixels used on one side of the square box defining the full
%workspace of the robot
n = 500; %1000; %1100; %1250;
%For the main program, n/2 needs to be an integer
%Thus, n should be even:
if(mod(n,2)); fprintf('ERROR: n is not even'); return; end
%For better readability of the main program the value n is saved to a file
f = fopen('n.txt','w'); fprintf(f,'%i',n); fclose(f);

%The robot (sphere):
tic
% delete('Sphere_robot/*');
rho = n/size_workspace; %Radius of the sphere in pixels
% %Again, for better readability of the main program the value of rho is
% %saved to a file:
% f = fopen('rho.txt','w'); fprintf(f,'%i',rho); fclose(f);
% centerX = n/2;
% centerY = centerX;
% for i=1:(2*rho)
%     z = -rho + i - 1;
%     %fprintf('true radius = %f \t', sqrt(rho*rho - z*z));
%     radius = round(sqrt(rho*rho - z*z));
%     %fprintf('radius = %i \n', radius);
%     d = (5 - radius * 4)/4;
%     x = 0;
%     y = radius;
%     Z = false(n);
%     while(x<=y)
%         Z(centerX + x, centerY + y) = 1;
%         Z(centerX + x, centerY - y) = 1;
%         Z(centerX - x, centerY + y) = 1;
%         Z(centerX - x, centerY - y) = 1;
%         Z(centerX + y, centerY + x) = 1;
%         Z(centerX + y, centerY - x) = 1;
```

```
%         Z(centerX - y, centerY + x) = 1;
%         Z(centerX - y, centerY - x) = 1;
%         if(d < 0)
%             d = d + 2 * x + 1;
%         else
%             d = d + 2 * (x - y) + 1;
%             y = y-1;
%         end
%         x = x + 1;
%     end
%     imwrite(Z,['Sphere_robot/' int2str(i) '.png'],'png');
% end
% Z = false(n);
% for i=(2*rho+1):n
%     imwrite(Z,['Sphere_robot/' int2str(i) '.png'],'png');
% end
% clear Z;
% toc
tic
%The surfaces and walls:
%Store the surfaces and walls in parametric equations form
f_functions = {@(s,t)s %Flat surf
               @(s,t)s %Tilted surf x
               @(s,t)s %Tilted surf y
               @(s,t)s %Tilted surf x,y
               @(s,t)s %Sine wave x, surf
               @(s,t)s %Sine wave y, surf
               @(s,t)s %Sine wave x,y, surf
               @(s,t)s %Gaussian dome, surf
               @(s,t)s %Sine wall 1
               @(s,t)s %Sine wall 2
               @(s,t)s %Sine tilted wall 1
               @(s,t)s %Sine tilted wall 2
               @(s,t)s %Tilted wall 1
               @(s,t)s}; %Tilted wall 2
g_functions = {@(s,t)t %Flat surf
               @(s,t)t %Tilted surf x
               @(s,t)t %Tilted surf y
               @(s,t)t %Tilted surf x,y
               @(s,t)t %Sine wave x, surf
               @(s,t)t %Sine wave y, surf
               @(s,t)t %Sine wave x,y, surf
               @(s,t)t %Gaussian dome, surf
               @(s,t)rho*sin(s/rho/4)+7.1*rho %Sine wall 1
               @(s,t)rho*sin(s/rho/4)+2.9*rho %Sine wall 2
               @(s,t)s+2.5*rho+rho*sin(s/rho/2) %Sine tilted wall 1
               @(s,t)s-2.5*rho-rho*sin(s/rho/2) %Sine tilted wall 2
               @(s,t)s+3*rho %Tilted wall 1
               @(s,t)s-3*rho}; %Tilted wall 2
h_functions = {@(s,t)n/2+1+0*s+0*t%Flat surf
               @(s,t)s %Tilted surf x
               @(s,t)t %Tilted surf y
               @(s,t)s/2+t/2 %Tilted surf x,y
               @(s,t)rho*sin(s/rho/4)+rho+1 %Sine wave x, surf
               @(s,t)rho*sin(t/rho/4)+rho+1 %Sine wave y, surf
```

```
                @(s,t)rho*sin((s+t)/rho/8)+rho+1 %Sine wave x,y, surf
                @(s,t)rho*sqrt(rho)*(exp(-((s-n/2).^2+(t-n/2).^2)/rho^3.8))-ceil((rho*(sqrt(rho)-7
                    %Gaussian dome, surf
                @(s,t)t %Sine wall 1
                @(s,t)t %Sine wall 2
                @(s,t)t %Sine tilted wall 1
                @(s,t)t %Sine tilted wall 2
                @(s,t)t/3.5 %Tilted wall 1
                @(s,t)t/3.5}; %Tilted wall 2
%The names of all the different surfaces and walls
names = {'Flat_Surf'; 'Tilted_x_Surf'; 'Tilted_y_Surf'; 'Tilted_xy_Surf'; ...
    'Sine_x_Surf'; 'Sine_y_Surf'; 'Sine_xy_Surf'; 'Gaussian_dome_Surf'; ...
    'Sine_Top_Wall'; 'Sine_Bot_Wall'; 'Sine_Tilted_Top_Wall'; ...
    'Sine_Tilted_Bot_Wall'; 'Tilted_Top_Wall'; 'Tilted_Bot_Wall'};
num_surf = length(names);
%for k = 1:num_surf
for k=13:14
    tic
    name = names{k};
    %mkdir(name);
    delete([name '/*']);
    %Grab the correct functions
    f = f_functions{k}; g = g_functions{k}; h = h_functions{k};
    I = false(n,n,n);      % Stores the surface in 3D matrix form
    s = (1:n); t = (1:n);  % Parametric vectors
    [S,T] = meshgrid(s,t); % Corresponding parametric matrices
    X = f(S,T); Y = g(S,T); Z = h(S,T); % Surface in 3 matrices form
    X(X<1) = 1; Y(Y<1) = 1; Z(Z<1) = 1; % Values must be > 1
    X(X>n) = n; Y(Y>n) = n; Z(Z>n) = n; % Values must be < n+1
    %Reshape the matrices into vector form
    x = round(reshape(X,[1,numel(X)]));
    y = round(reshape(Y,[1,numel(Y)]));
    z = round(reshape(Z,[1,numel(Z)]));
    v = unique([x',y',z'],'rows'); %Remove duplicates
    x = v(:,1); y = v(:,2); z = v(:,3);
    linIndex = sub2ind(size(I), x, y, z); %Transforming into linear index
    I(linIndex) = true; % Store the surface into the 3D matrix
    for i = 1:n
        imwrite(I(:,:,i),[name '/' int2str(i) '.png'], 'png');
    end
    %Save the picture
    handle = figure;set(handle, 'Visible', 'off');
    mesh(round(X),round(Y),round(Z));
    view(150,130); axis equal; box on; colormap cool;
    xlabel('x-axis'); ylabel('y-axis'); zlabel('z-axis');
    saveas(handle,[int2str(k) '.fig']);
    %In order to display the figure, one needs to do the following:
    %openfig('figure.fig', 'Visible');
    toc
end
toc
clear ('variables'); close all;
```