```
void compute_external_forces()
{
  int i, j, k, row, col;
  //  double ri[2], rj[2];
  double dr[2], mag, theta, dij, fx, fy, tz;
  IObj obji, objj;

  void copy_object(), update_objects(), SIMfwd_kinematics(), SIMarm_Jacobian();

// copy the pos/vel info from device (mobile_base, arms, toy) structures
  update_objects();      //    into inertial "IObj objects[NBODY]" array

  // initialize net_extForce sums
  objects[BASE].net_extForce[X] = objects[BASE].net_extForce[Y] =
    objects[BASE].net_extForce[THETA] = 0.0;

  objects[ARM1].net_extForce[X] = objects[ARM1].net_extForce[Y] =
    objects[ARM1].net_extForce[THETA] = 0.0;

  objects[ARM2].net_extForce[X] = objects[ARM2].net_extForce[Y] =
    objects[ARM2].net_extForce[THETA] = 0.0;

  objects[TOY].net_extForce[X] = objects[TOY].net_extForce[Y] =
    objects[TOY].net_extForce[THETA] = 0.0;

  for (i=0; i<(NBODY-1); ++i) { // compute force on body i by body j
    for (j=(i+1); j<NBODY; ++j) {

      copy_object(i, &obji); copy_object(j, &objj);

      //      if ((i==3) && (objects[i].id == TRIANGLE)) { // triangle object
      // dr[X] = dr[Y] = 0.0;
      // // sum compressive forces/moments over three vertices of
      //        // rigid triangle
      //
      // for (k = -1; k <= 1; k++) {
      //   // position of the first vertex
      //   theta = (double)k*(2.0*M_PI/3.0);
      //   //   xt = RT*cos(theta);
      //   //   yt = RT*sin(theta);
      //   r[3][X] = objects[i].position[X]
      //     + (objects[i].spoke_length * cos(theta))
      //          * cos(objects[i].position[THETA])
      //     - (objects[i].spoke_length * sin(theta))
      //          * sin(objects[i].position[THETA]);
      //   r[3][Y] = objects[i].position[Y]
      //     + (objects[i].spoke_length * cos(theta))
      //          * sin(objects[i].position[THETA])
      //     + (objects[i].spoke_length * sin(theta))
      //          * cos(objects[i].position[THETA]);
```

```
//   // the relative position vector
//   dr[X] = r[3][X] - r[j][X];
//   dr[Y] = r[3][Y] - r[j][Y];
//   mag = sqrt(SQR(dr[X]) + SQR(dr[Y]));
//   dij = MAX(0.0,
//     (objects[i].circle_radius+objects[j].circle_radius-mag));
//   fx = K_COLLIDE*dij*(dr[X]/mag);
//   fy = K_COLLIDE*dij*(dr[Y]/mag);
//   tz = (objects[i].spoke_length *
//                   cos(objects[i].position[THETA])) * fy -
//      (objects[i].spoke_length *
//                   sin(objects[i].position[THETA])) * fx;
//
//   objects[i].net_extForce[X] += fx;
//   objects[i].net_extForce[Y] += fy;
//   objects[i].net_extForce[THETA] += tz;
//   // (i=3, j=4) is only combination, j=4 is the occupancy grid,
//   objects[j].net_extForce[X] -= fx;
//   objects[j].net_extForce[Y] -= fy;
//   objects[j].net_extForce[THETA] -= tz;
// }
//      }
//
//      else if ((j==3)&&(objects[j].id == TRIANGLE)) {
// dr[X] = dr[Y] = 0.0;
//        // sum compressive forces/moments over three vertices of
//        // rigid triangle
// for (k = -1; k <= 1; k++) {
//   // position of the first vertex
//   theta = (double)k*(2.0*M_PI/3.0);
//   //   xt = RT*cos((double)k*one_twenty);
//   //   yt = RT*sin((double)k*one_twenty);
//   r[3][X] = objects[j].position[X]
//     + (objects[j].spoke_length * cos(theta))
//           * cos(objects[j].position[THETA])
//     - (objects[j].spoke_length * sin(theta))
//           * sin(objects[j].position[THETA]);
//   r[3][Y] = objects[j].position[Y]
//     + (objects[i].spoke_length * cos(theta))
//           * sin(objects[j].position[THETA])
//     + (objects[i].spoke_length * sin(theta))
//           * cos(objects[j].position[THETA]);
//   // the relative position vector
//   dr[X] = r[i][X] - r[3][X];
//   dr[Y] = r[i][Y] - r[3][Y];
//   mag = sqrt(SQR(dr[X]) + SQR(dr[Y]));
//   dij=MAX(0.0,(objects[i].circle_radius
//                          + objects[j].circle_radius-mag));
//   fx = K_COLLIDE*dij*(dr[X]/mag);
//   fy = K_COLLIDE*dij*(dr[Y]/mag);
//   tz = (objects[j].spoke_length *
```

```
      //                   cos(objects[j].position[THETA])) * fy -
      //               (objects[j].spoke_length *
      //                   sin(objects[j].position[THETA])) * fx;
      //
      //   objects[i].net_extForce[X] += fx;
      //   objects[i].net_extForce[Y] += fy;
      //   objects[i].net_extForce[THETA] = 0.0;
      //
      //   objects[j].net_extForce[X] -= fx;
      //   objects[j].net_extForce[Y] -= fy;
      //   objects[j].net_extForce[THETA] -= tz;
      // }
      //     }

      // body #4 is the occupancy grid - sum compression
      if (j==4){
printf("checking body i=%d bouncing on OBSTACLE j=%d\n", i,j);
       for (row=0; row<NBINS; ++row) {
         for (col=0; col<NBINS; ++col) {
           if (Roger.world_map.occupancy_map[row][col] == OBSTACLE) {
             dr[X] = objects[i].position[X] - (MIN_X + (col+0.5)*XDELTA);
             dr[Y] = objects[i].position[Y] - (MAX_Y - (row+0.5)*YDELTA);
             mag = sqrt(SQR(dr[X]) + SQR(dr[Y]));
             dij = MAX(0.0, (objects[i].circle_radius + R_OBSTACLE - mag));
             fx = K_COLLIDE*dij*(dr[X]/mag);
             fy = K_COLLIDE*dij*(dr[Y]/mag);
             tz = 0.0;

             objects[i].net_extForce[X] += fx;
             objects[i].net_extForce[Y] += fy;
             objects[i].net_extForce[THETA] += tz;

      //       objects[j].net_extForce[X] -= fx;
      //       objects[j].net_extForce[Y] -= fy;
      //       objects[j].net_extForce[THETA] += 0.0;
           }
         }
       }
printf("\tforce on body i=%d  f = [%6.4lf %6.4lf %6.4lf]\n",
       i, fx, fy, tz);
      }
      else { // j not equal to 4: BASE || ARM#1 || ARM#2 || circle object


      // for (k = -1; k <= 1; k++) {
      //   // position of the first vertex
      //   theta = (double)k*(2.0*M_PI/3.0);
      //   //   xt = RT*cos(theta);
      //   //   yt = RT*sin(theta);
      //   r[3][X] = objects[i].position[X]
      //     + (objects[i].spoke_length * cos(theta))
```

```
      //          * cos(objects[i].position[THETA])
      //     - (objects[i].spoke_length * sin(theta))
      //          * sin(objects[i].position[THETA]);
      //   r[3][Y] = objects[i].position[Y]
      //     + (objects[i].spoke_length * cos(theta))
      //          * sin(objects[i].position[THETA])
      //     + (objects[i].spoke_length * sin(theta))
      //          * cos(objects[i].position[THETA]);
      //   // the relative position vector
dr[X] = obji.position[X] - objj.position[X];
dr[Y] = obji.position[Y] - objj.position[Y];
mag = sqrt(SQR(dr[X]) + SQR(dr[Y]));
dij = MAX(0.0, (obji.circle_radius + objj.circle_radius - mag));
fx = K_COLLIDE*dij*(dr[X]/mag);
fy = K_COLLIDE*dij*(dr[Y]/mag);
tz = (obji.spoke_length * cos(obji.position[THETA])) * fy -
  (obji.spoke_length * sin(obji.position[THETA])) * fx;

objects[i].net_extForce[X] += fx;
objects[i].net_extForce[Y] += fy;
objects[i].net_extForce[THETA] += tz;

objects[j].net_extForce[X] -= fx;
objects[j].net_extForce[Y] -= fy;
objects[j].net_extForce[THETA] -= tz;
      }
    }
  }

  // BASE
  mobile_base.extForce[X] = objects[BASE].net_extForce[X];
  mobile_base.extForce[Y] = objects[BASE].net_extForce[Y];

  // ARM #1
  //  reality check: why do you need the negative of fb?
  arms[LEFT][NARM_FRAMES - 1].extForce[X] = -objects[ARM1].net_extForce[X];
  arms[LEFT][NARM_FRAMES - 1].extForce[Y] = -objects[ARM1].net_extForce[Y];

  // ARM #2
  //  reality check: why do you need the negative of fb?
  arms[RIGHT][NARM_FRAMES - 1].extForce[X] = -objects[ARM2].net_extForce[X];
  arms[RIGHT][NARM_FRAMES - 1].extForce[Y] = -objects[ARM2].net_extForce[Y];

  // TOY OBJECT
  toy.net_extForce[X] = objects[TOY].net_extForce[X];
  toy.net_extForce[Y] = objects[TOY].net_extForce[Y];
  toy.net_extForce[THETA] = 0.0;

  //  printf("exiting compute_external_forces()\n"); fflush(stdout);
}
```