# Appendix B

# DAP_BC User Manual

(December 2018)

The Dynamic Analysis Program: Body Coordinates (`DAP_BC`) is a general-purpose program that is not developed around a specific problem—it can simulate the dynamics of a variety of problems based on the description and the data provided by the user. The program contains most of the features that have been discussed in Chapter 7 of the textbook, such as kinematic joints and force elements. A user can modify the program to include new features and capabilities. The program is capable of performing kinematic, inverse dynamic, and forward dynamic analyses by integrating the equations of motion (refer to Chapters 12 and 13). The program can also solve static and static equilibrium problems, and correct initial conditions on the coordinates and velocities prior to performing a forward dynamic analysis (refer to Chapter 14). An animation program that provides a simple animated stick drawing of the dynamic response of the analyzed model accompanies the analysis program.

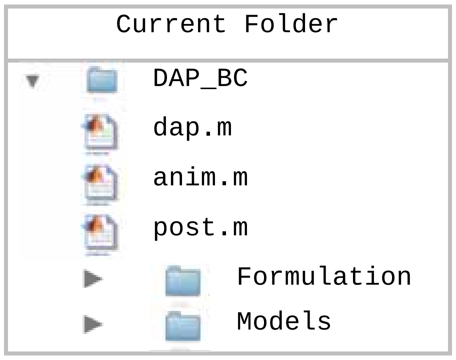
## B.1 USING DAP_BC

The program `DAP_BC` and the accompanying examples are all in one folder named `DAP_BC` followed by the date of its last revision. As shown in Figure B.1(a), the contents of this folder are three script M-files and two other folders. The folder named `Models` contains several examples of simple multibody systems, as shown in Figure B.1(b). One of these models, `SP` (Sliding Pendulum), is discussed in this manual in detail as a guide in learning how to use the program. Some of the other models are discussed in Chapter 8 of the textbook. If we construct any new models, they should be saved inside this folder.

The script M-file `dap` is the main program to execute in order to simulate the response of a system. The program prompts the user with a few questions prior to performing a simulation. After a simulation is completed, the user may execute the script M-file `anim` to visualize a crude animation of the simulated model. Following a simulation, we can also execute the script M-file `post` for post-processing of the simulated response. This script transforms the results into a more recognizable set of arrays for further analyses or plots.

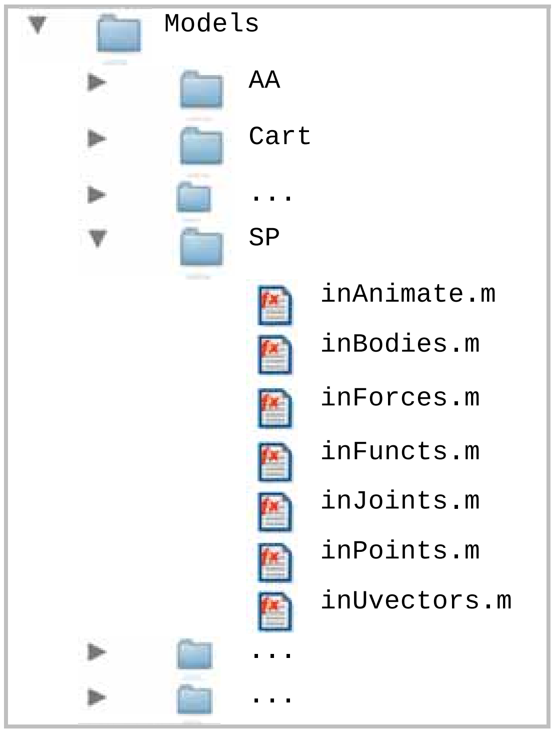
The folder `Formulation` contains other subfolders, script and function M-files. These are the body-coordinate formulations from Chapter 7 for constructing and evaluating constraints, Jacobian sub-matrices, forces, mass matrix, etc. To simulate the response of a multibody system, using the existing capabilities of the program, there is no need for a user to revise anything in this folder.

It is highly recommended to make the folder `DAP_BC` the "Current Folder" in MATLAB, as indicated in Figure B.1. This facilitates setting the "paths" to different folders and M-files. The program will then automatically set all the necessary paths.



```
Current Folder
▼ DAP_BC
    dap.m
    anim.m
    post.m
    ▶ Formulation
    ▶ Models
```

(a)



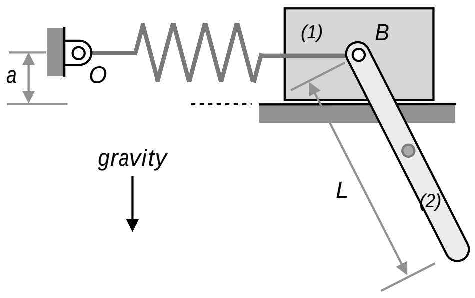
```
▼ Models
    ▶ AA
    ▶ Cart
    ▶ ...
    ▼ SP
        inAnimate.m
        inBodies.m
        inForces.m
        inFuncts.m
        inJoints.m
        inPoints.m
        inUvectors.m
    ▶ ...
    ▶ ...
```

(b)

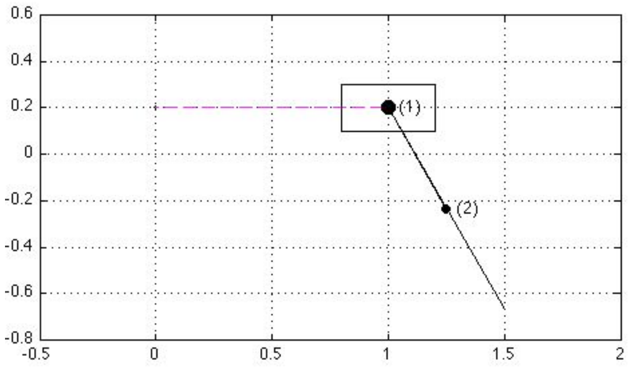
Figure B.1 Folders and script files of `DAP_BC`.

### B.1.1 Example

This sliding-pendulum example, shown in Figure B.2(a), was considered for FBD construction in Section 6.1.3. Body (*1*), the slider, is connected to the ground by a frictionless sliding joint, and to body (*2*), the pendulum, by a pin joint. A spring is attached between the ground and body (*1*), and the gravity acts on the system. All the necessary data are listed in Example 6.3. The body-coordinate model of this system is provided in the subfolder `SP`, which resides inside the folder `Models`. To simulate the response of this system, we perform the following steps.



(a)



(b)

Figure B.2 (a) Sliding-pendulum example and (b) its animated presentation.

In the Command Window we type `dap` and then respond to each question, as appears in the right column.

```
>> dap
Which folder contains the model? ...................    SP
```

We entered `SP` for Sliding Pendulum. Next, the program asks if we want the program to correct the initial conditions on the coordinates and velocities. If we are certain that the provided initial conditions satisfy the position and velocity constraints, we may answer no, otherwise we should respond yes.

```
Do you want to correct the initial conditions?
                                   [(y)es/(n)o] ....    y
```

Since we asked the program to correct the initial conditions, the corrected set of initial conditions will be reported:

```
Corrected coordinates
 x           y           phi
   1         0.2           0
1.25    -0.23301      0.5236
Corrected velocities
 x-dot       y-dot       phi-dot
  0    0    0
  0    0    0
```

Next, the program asks for the final time of integration (the starting time is "0") and the reporting time steps of the results (all in seconds).

```
Final time = ?   ...................................    4
Reporting time-step = ?   ..........................    0.02
```

At this point, the program starts integrating the equations of motion from "0" to "4" seconds. To keep the user informed of the progress, the program reports the integration time every 100 function-evaluations. (A function evaluation means that the equations of motion have been constructed and solved for the accelerations.) At the completion of integration, the program reports the number of function evaluations. This information is reported as a measure for the efficiency (or inefficiency) of the method of solution.

```
     0
    0.4605
    1.1473
    1.8110
    2.5231
    3.1091
    3.7231
Number of function evaluations = 655
>>
```

At the completion of the simulation, `dap` saves the simulated coordinates and velocities for each moving body from $t = 0$ to $t = 4.0$ at every $\Delta t = 0.02$ seconds. We can run either the script `anim.m` or `post.m` to view or process the response.

To view an animation of the results, in the Command Window we type:

```
>> anim
```

A graphical window will display a crude representation of the system in its initial configuration, as shown in Figure B.2(b). The body centers, the pin joints, and some of the defined points are marked on the plot. The spring is displayed in dashed-line. Pressing any keys on the keyboard initiates the animation of the response.

We can also execute the M-file `post` to recover some the results that were not reported by the integration process, such as the accelerations and Lagrange multipliers. The program `post` organizes the results in separate recognizable arrays (these arrays will be discussed later in this manual). We can, for example, plot the angle of body (*2*) versus time by typing in the Command Window the followings:

```
    >> plot(T,p(:,2),'k')
    >> grid on
```

The resulting plot is shown in Figure B.3.

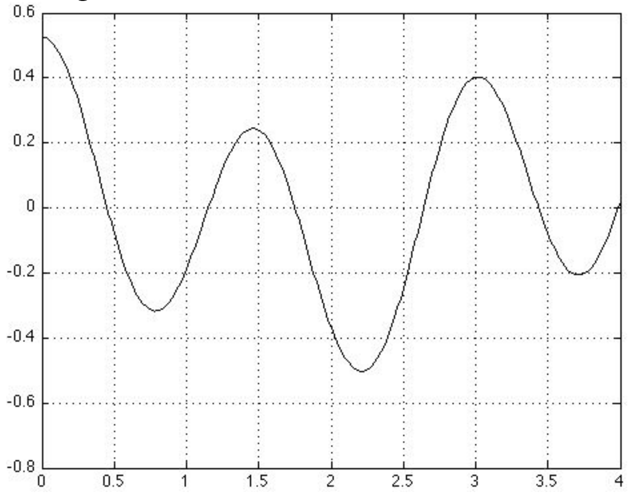


Figure B.3 Angle of the pendulum versus time.

Through this example, we demonstrated that it is simple to simulate the dynamic response of an existing model. We can repeat the same steps to simulate the response of other models that are already available in the folder `Models`. Next we describe how a model is constructed.

## B.2 CONSTRUCTING A MODEL

A multibody model is organized in seven function M-files as depicted in Figure B.1(b). If we look at the existing examples in the folder `Models`, we find that all of them contain exactly the same named function M-files. Some models may have an extra M-file named `user_force`, which will be discussed later in this section.

The function M-files provide data for the bodies, points, unit vectors, joints, forces, and any analytical functions that may be needed (for example, for a driver motor). Each set of data must be organized in a specific form, using MATLAB's *structure* array, or `struct`. The structure for each element (a body, a joint, etc.) contains certain fields, where some of the fields require data to be provided by the user. The required data and the structure of each element are described in the followings, using the sliding pendulum example.

**Example B.1**

To begin constructing a model for the sliding pendulum, we first separate the bodies, assign indices, attach to each a body-fixed frame, and define a global *x*-y frame. As shown in Figure B.4, the two moving bodies are marked as (*1*) and (*2*). We have identified three points as $O_0$, $B_1$ and $B_2$ that are needed for the joints and the spring attachment points. We have assigned indices to these points as [1], [2], [3] (the order of numbering is up to us). We also need two unit vectors for the translational joint: $\vec{u}_0$ on the ground and $\vec{u}_1$ on the slider. These unit vectors are numbered [[1]] and [[2]]. We are now ready to set up the M-files for this model. The constant dimensions are $a = 0.2$ m and $L = 1.0$ m.

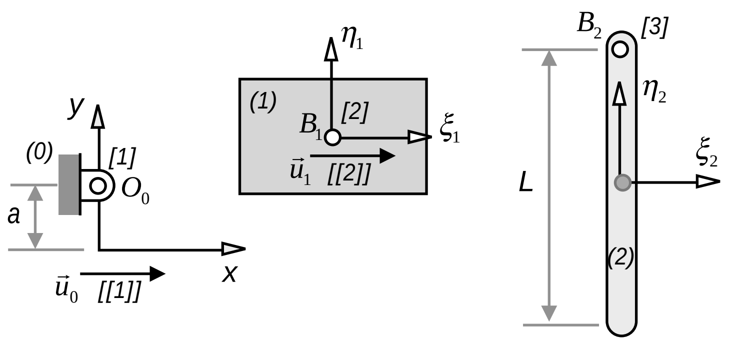


Figure B.4 Individual bodies and the assigned indices for the sliding pendulum example.

**M-file `inBodies`:** In this file we define the mass, moment of inertia, and initial conditions of the coordinates and velocities for all the bodies, except for the ground. To construct this file, we must follow the structure listed in Table B.1. There are six fields in this structure that need data from the user. The comment for each field should state clearly what the data is: for example, '`m`' is mass and '`J`' is moment of inertia. Each field also contains a default value; for example, if we do not state the mass for a body, the program will use the default value of 1.0 unit. The entries for the fields '`m`' and '`J`' will not be altered by the program. However, the values for other fields, for example '`r`', will be updated by the program during a simulation, as the bodies move. There are additional fields in this structure that are not listed in this table— the program will use those fields to save various data during a simulation.

Table B.1

```
function Body = Body_struct
Body = struct ( ...
    'm'     , 1     , ... % mass
    'J'     , 1     , ... % moment of inertia
    'r'     , [0;0] , ... % x and y coordinates
    'p'     , 0     , ... % angle phi
    'r_d'   , [0;0] , ... % time derivative of x and y
    'p_d'   , 0     , ... % time derivative of phi
    ...                   );
```

**Example B.1 continued** (`DAP_BC/Models/SP`)

The following data for the sliding pendulum are provided in the function M-file `inBodies`.

$m_1 = 5.0$

$J_1 = 4.0$

$x_1 = 1.0\,,\ y_1 = 0.2$

$\phi_1 = 0$

$m_2 = 2.0$

$J_2 = 0.2$

$x_2 = 1.25\,,\ y_2 = -0.233$

$\phi_2 = \pi / 6$

```
function inBodies
    include_global

B1 = Body_struct;
B1.m = 5.0;
B1.J = 4.0;
B1.r = [1.0; 0.2];

B2 = Body_struct;
B2.m = 2.0;
B2.J = 0.2;
B2.r = [1.25; -0.233];
B2.p = pi/6;

Bodies = [B1; B2];
```

In this M-file we have created two bodies, arbitrarily named `B1` and `B2`, which have been assigned the structure named `Body_struct`. We have not stated an angle for the first body or velocities for either body since their values are zero (defaults). The two created bodies are saved in the array `Bodies`.

The statement `include_global` must appear in this and all other input M-files. Through the global statements in the script "`include_global`", the array `Bodies` will become available to the rest of the program. From this point on, `Bodies(1)` will refer to the first body saved in this array, which is body (*1*), and `Bodies(2)` will refer to body (*2*). For example, we can access the coordinates of body (*2*) from any M-file that contains the `include_global` statement, by stating `Bodies(2).r`. In general, the order in which the defined bodies are placed in the array `Bodies` becomes body indices for a model.

**M-file `inPoints`:** In this file we provide body-fixed coordinates, $\breve{\mathbf{s}}_i^P$, for all the defined points, including the points on the ground. The data must be entered according to the structure listed in Table B.2.

Table B.2

```
function Point = Point_struct
Point = struct ( ...
    'Bindex'  , 0      , ... % body index
    'sPlocal' , [0;0]  , ... % body-fixed (local) coordinates
    ...                     );
```

**Example B.1** (cont.)

The following body-fixed coordinates are extracted from the figure.

$$\breve{\mathbf{s}}_0^O = \begin{Bmatrix} 0 \\ 0.2 \end{Bmatrix}$$

$$\breve{\mathbf{s}}_1^B = \begin{Bmatrix} 0 \\ 0 \end{Bmatrix}$$

$$\breve{\mathbf{s}}_2^B = \begin{Bmatrix} 0 \\ 0.5 \end{Bmatrix}$$

Based on the data, we construct the function M-file `inPoints`. The defined points are saved in the array `Points`, which is the output of this file. The order in which the defined points are placed in this array becomes point indices for this model.

```
function inPoints
    include_global

P1 = Point_struct;
P1.Bindex = 0;
P1.sPlocal = [0; 0.2];

P2 = Point_struct;
P2.Bindex = 1;
P2.sPlocal = [0; 0];

P3 = Point_struct;
P3.Bindex = 2;
P3.sPlocal = [0; 0.5];

Points = [P1; P2; P3];
```

**M-file `inUvectors`:** In this file we provide the body-fixed components, $\breve{\mathbf{u}}_i$, of the defined unit vectors, including the unit vectors on the ground. The data for the vectors must be entered according to the structure listed in Table B.3.

Table B.3

```
function Unit = Unit_struct
Unit = struct ( ...
    'Bindex' , 0     , ... % body index
    'ulocal' , [1;0] , ... % body-fixed (local) components
    ...                    );
```

**Example B.1** (cont.)

The following body-fixed components are defined for the two unit vectors in the model:

$$\mathbf{u}'_0 = \begin{Bmatrix} 1.0 \\ 0 \end{Bmatrix},\ \mathbf{u}'_1 = \begin{Bmatrix} 1.0 \\ 0 \end{Bmatrix}$$

Based on the data, we construct the function M-file `inUvectors`.

The output of this file is the array `Uvectors`.

```
function inUvectors
    include_global

U1 = Unit_struct;
U1.Bindex = 0;
U1.ulocal = [1.0; 0];

U2 = Unit_struct;
U2.Bindex = 1;
U2.ulocal = [1.0; 0];

Uvectors = [U1; U2];
```

**M-file `inForces`:** In this file we define the forces that act on the bodies, such as gravity, spring-dampers, or other force elements. Different force elements require different type of data; therefore the structure for forces, as is listed in Table B.4, contains more fields than those for bodies or points.

Table B.4

```
function Force = Force_struct
Force = struct ( ...
    'type'   , 'ptp',   ... % element type: ptp, weight, f, T, etc.
    'iPindex', 0    ,   ... % index of the point on body (i)
    'jPindex', 0    ,   ... % index of the point on body (j)
    'iBindex', 0    ,   ... % index of body (i)
    'jBindex', 0    ,   ... % index of body (j)
    'k'      , 0    ,   ... % spring stiffness
    'L0'     , 0    ,   ... % undeformed length
    'theta0' , 0    ,   ... % undeformed angle
    'dc'     , 0    ,   ... % damping coefficient
    'f_a'    , 0    ,   ... % constant actuator force
    'T_a'    , 0    ,   ... % constant actuator torque
    'gravity', 9.81 ,   ... % gravitational constant
    'wgt'    , [0;-1],  ... % gravitational direction
    'flocal' , [0;0],   ... % constant force in local frame
    'f'      , [0;0],   ... % constant force in x-y frame
    'T'      , 0        ... % constant torque
    'iFunct' , 0        ... % analytical function index
                       );
```

The first field for a force element is its `type`. The available force elements types are:

Gravitational force (`.type = 'weight'`)
Point-to-point spring-damper-actuator (`.type = 'ptp'`)
Rotational spring-damper-actuator (`.type = 'rot-sda'`)
Constant force described in body-fixed frame (`.type = 'flocal'`)
Constant force described in *x-y* frame (`.type = 'f'`)
Constant torque (`.type = 'T'`)
User supplied forces (`.type = 'user'`)

Each type requires different set of data as described in the followings:

| `.type` | Required entries | Comments |
|---|---|---|
| `‘ptp’` | `iPindex` `jPindex` `k` `L0` `dc` `f_a` | There is no need to provide body indices. Since the point indices have already been provided, the program can determine the corresponding body indices. The program constructs a vector $\vec{d}$ between the two defined points. |
| `‘rot-sda’` | `iBindex` `jBindex` `k` `theta0` `dc` `T_a` | |
| `‘weight’` | `gravity` `wgt` | The gravitational force to be applied on all bodies. |
| `‘flocal’` | `iBindex` `flocal` | This is a constant force defined in the body-fixed frame that will be applied continuously on the body. |
| `‘f’` | `iBindex` `f` | This is a constant force defined in the *x*-*y* frame that will be applied continuously on the body. |
| `‘T’` | `iBindex` `T` | This is a constant torque that will be applied continuously on the body. |
| `‘user’` | | The M-file `user_force.m` is required. Such a file will be discussed in some of the examples. Also see the note following the example. |

**Note:** The default settings for the gravitational constant and the direction of the gravitational force are 9.81 m/s$^2$ and in the negative *y*-direction. With this default setting for the gravitational constant, the program assumes that the SI units are used in all the input M-files. If we wish to switch to a different system of units, it should be done here.

**Example B.1** (cont.)

The spring characteristic data is:

$$k = 20\,,\ {}^0L = 0.6$$

The spring is connected between points [1] and [2].

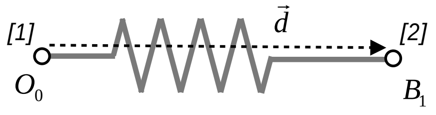


Figure B.5

The program constructs a vector $\vec{d}$ between the two defined points. We also need to apply gravitational forces on the bodies in the default direction.

Based on the data, we construct the function M-file `inForces`. The output of this file is the array `Forces`.

```
function inForces
    include_global

F1 = Force_struct;
F1.type = 'ptp'; % default
F1.iPindex = 2;
F1.jPindex = 1;
F1.k = 20;
F1.L0 = 0.6;

F2 = Force_struct;
F2.type = 'weight';

Forces = [F1; F2];
```

**Note on Contact:** In the folder `Forces` (inside the folder `Formulations`) there are three function M-files for modeling contact between a body and the ground. The files are `Contact`, `Contact_LN`, and `Contact_FM`. The M-files `Contact_LN` and `Contact_FM` describe the continuous contact force models of Eqs. 11.42 and 11.43 respectively. The M-file `Contact` uses these force models to determine to determine the contact force between a point on a body and the ground in the *y*-direction. It then applies the force and its corresponding moment to the force-array of that body. The M-file `Contact` can be used in a `user_force` file to model contact. The function `Contact` requires the following input arguments:

```
      Contact(Ci, Pi, Bi, k, e, Mi)
```

where,

| | |
|---|---|
| `Ci:` | a given index for the contact |
| `Pi:` | point index |
| `Bi:` | body index |
| `Mi:` | model index = 1 Eq. 11.42; = 2 Eq. 11.43 |
| `k, e:` | model parameters |

As an example, assume in a model point *A* on body (3) and point *B* on body (2) may contact the ground (not necessarily simultaneously). Assume the indices for points *A* and *B* are [5] and [7] respectively. Assume for a given set of parameters we want to use Eq. 11.42 for both contact points. For this purpose we need the following statements in the `user_force` file:

```
    k = 10^11; e = 0.95;
    Contact(1, 5, 3, k, e, 1)
    Contact(2, 7, 2, k, e, 1)
```

An example is provided in the model **`Rod`**.

**M-file `inJoints`:** In this file we define the constraints that are applied on the bodies, either as kinematic joints or as drivers. Since different joints or drivers require different type of data, the structure for these constraints contains different fields, as listed in Table B.5.

Table B.5

```
function Joint = Joint_struct
Joint = struct ( ...
    'type'     , 'rev' , ... % joint type: rev, tran, rev-rev, etc.
    'iBindex'  , 0     , ... % body index i
    'jBindex'  , 0     , ... % body index j
    'iPindex'  , 0     , ... % point Pi index
    'jPindex'  , 0     , ... % point Pj index
    'iUindex'  , 0     , ... % unit vector u_i index
    'jUindex'  , 0     , ... % unit vector u_j index
    'iFunct'   , 0     , ... % function index
    'L'        , 0     , ... % constant length
    'R'        , 1     , ... % constant radius
    'x0'       , 0     , ... % initial condition for a disc
    'p0'       , 0     , ... % initial condition for a disc or rigid
    ...
);
```

The first field for a joint element is its `type`. The available joint elements are:

Revolute or pin (`.type = ‘rev’`)
Translational or sliding (`.type = ‘tran’`)

Revolute-revolute (`.type = 'rev-rev'`)
Revolute-translational (`.type = 'rev-tran'`)
Rigid or bracket (`.type = 'rigid'`)
Rolling disc (`.type = 'disc'`)
Relative rotational driver (`.type = 'rel-rot'`)
Relative translational driver (`.type = 'rel-tran'`)

Each type of joint requires different set of data as described in the followings:

| `.type` | Required entries | Comments |
|---|---|---|
| `'rev'` | `iPindex` `jPindex` | There is no need to enter body indices, since the point indices can provide that information. |
| `'tran'` | `iPindex` `jPindex` `iUindex` `jUindex` | |
| `'rev-rev'` | `iPindex` `jPindex` `L` | |
| `'rev-tran'` | `iPindex` `jPindex` `iUindex` `L` | |
| `'rigid'` | `iBindex` `jBindex` | Refer to Eq. **7.35**. |
| `'disc'` | `iBindex` `R` `x0` `p0` | A circular disc rolling on the horizontal ground. Refer to Eq. **7.38**. |
| `'rel-rot'` | `iBindex` `jBindex` `iFunct` | A rotational motor or actuator (driver) that requires an analytical function. |
| `'rel-tran'` | `iPindex` `jPindex` `iFunct` | A translational motor or actuator (driver) that requires an analytical function. |

**Example B.1 (**cont.)

In this model we have a translational and a revolute joint. For the translational joint we state the type to be '`tran`', and we define one point and one unit vector on each of the bodies.

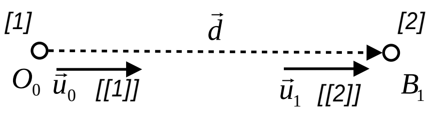


Figure B.6

For the revolute joint we need to define the indices of the two coinciding points as [2] and [3].

The output of this file is the array `Joints`.

```
function inJoints
    include_global

J1 = Joint_struct;
J1.type = 'tran';
J1.iPindex = 2;
J1.jPindex = 1;
J1.iUindex = 2;
J1.jUindex = 1;

J2 = Joint_struct;
J2.type = 'rev';
J2.iPindex = 2;
J2.jPindex = 3;

Joints = [J1; J2];
```

For the translational joint, we must make sure that the point `iPindex` and the vector `iUindex` have previously been defined on the same body, and the same for the point and vector on the other body. The program will construct a vector $\vec{d}$ between the two points with the arrow pointing to point `iPindex`.

**M-file `inFuncts`:** A user may define a function such as $f = f(t)$ to describe the kinematics of a driver motor as function of time, for example, or the force of an actuator. The parameter $t$ could be the time or a coordinate such as $x$ of a body. The structure for these elements contains different fields, as listed in Table B.6.

Table B.6

```
function Funct = Funct_struct
    include_global
Funct = struct ( ...
    'type'      , 'a'    , ... % function type a, b, or c
    't_start'   , 0      , ... % required for functions b, c
    'f_start'   , 0      , ... % required for functions b, c
    't_end'     , 1      , ... % required for functions b, c
    'f_end'     , 1      , ... % required for function b
    'dfdt_end'  , 1      , ... % required for function c
    'ncoeff'    , 3      , ... % number of coefficients
    'coeff'     , []       ... % required for function a
                           );
```

The first field for a function element is its `type`. In this version of the program, three types of functions are available. Based on the type of the function, the corresponding parameters and coefficients are saved in the other fields.

**Type "a":** This function can be used, for example, to define a constant angular velocity motor, such as $\phi_i = \pi + 2\pi t$. The general expression for this function is:

$$f = c_1 + c_2 t + c_3 t^2 \tag{8.1}$$

For the driver function $\phi_i = \pi + 2\pi t$, the coefficients are $c_1 = \pi$, $c_2 = 2\pi$, and $c_3 = 0$. The user must supply values for the three coefficients; e.g.,

```
        F1.type ='a';
        F1.coeff = [pi 2*pi 0];
```

This is the only function that the user must provide the coefficients directly. For the other two function types, the program determines the coefficients based on several parameters that are provided by a user.

**Type "b":** This function can be used to vary the coordinate of a point or the length of an actuator from an initial value to a final value in a specified period of time. The general expression for this function is:

$$f = c_1 t^3 + c_2 t^4 + c_3 t^5 \tag{8.2}$$

This function and its derivatives are depicted in Figure B.7. The function has zero first and second derivatives at the start and end points. For this function we do not provide values for the three coefficients—we must supply four parameters: the *start* and the *end* values for $t$ and $f$. The program computes the three coefficients in the M-file `functData` and saves them in the array `coeff`. The following is an example for the required data:

```
        F1.type ='b';
        F1.t_start = 0;
        F1.f_start = 0.3;
        F1.t_end = 2;
        F1.f_end = 1.5;
```

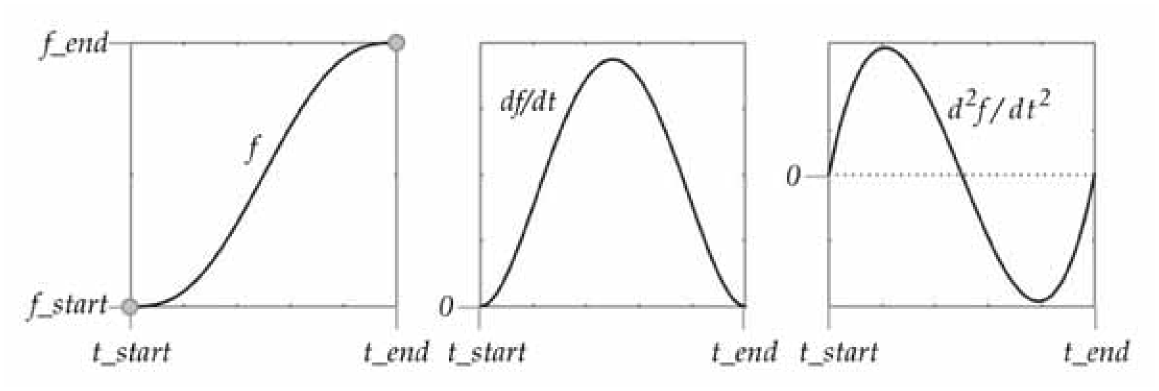


Figure B.7: Type “b” function and its first and second derivatives

**Type “c”:** The first derivative of this function rises from zero to a final value during a specified time period and keeps that value for the remaining duration of simulation. This function can be useful in simulations where a driver should raise the velocity (first derivative) from zero to a specified speed. The general expression for this function is:

$$f = c_1 + c_2 t^4 + c_3 t^5 + c_4 t^6 \qquad (8.3)$$

This function and its derivatives are depicted in Figure B.8, where the function has zero first, second, and third derivatives at the start, and zero second and third derivatives at the end. The first derivative has a non-zero value at the end point. The user must supply four parameters: the *start* and *end* values for *t*, the start value for *f*, and the end value for the first derivative. The program adjusts the `ncoeff` from the default value of 3 to 4, and computes the four coefficients of the function in the M-file `functData` and saves them in the array `coeff`. The following is an example for the required data:

```
F1.typ2 =’c’;
F1.t_start = 0;
F1.f_start = 0;
F1.t_end = 2;
F1.dfdt_end = 1.5;
```

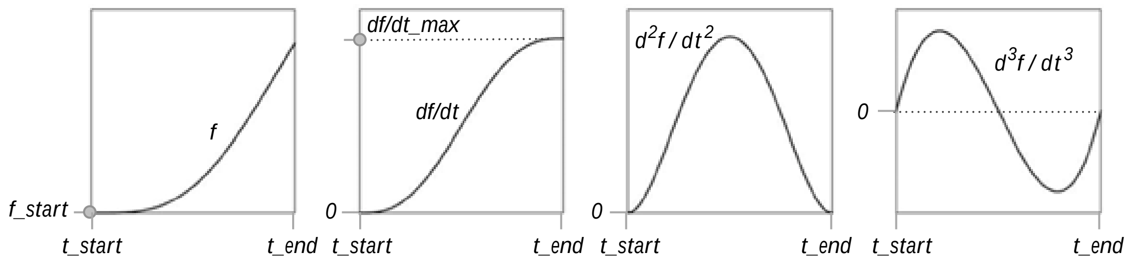


Figure B.8: Type “c” function and its first, second, and third derivatives

**Example B.1 (**cont.)

There are no analytical functions in the sliding pendulum model. However, we still need to construct the M-file `inFuncts` and provide a *blank* array `Functs` as the output of this M-file.

```
function inFuncts
    include_global

Functs = [];
```

**Note:** This is a short note on how the program uses a function during a simulation. There is a function named `functs` that the program uses as

```
[fun, fun_d, fun_dd] = functs(n, t)
```

The first input to this function is the function number (index) `n`, which could be 1, 2, ... based on the order of the functions that have been defined in `inFunct`. The second input entry `t` is the parameter time or a coordinate. `functs` returns the value of the function and its first and second derivatives as `fun`, `fun_d`, and `fun_dd` respectively.

This information becomes useful if the user wants to apply one of the available functions in a user supplied force function, or to add other capabilities to the program, or to develop a driver constraint in a model in the program `DAP_JC` (refer to Chapter 9).

**M-file `inAnimate`:** One of the best ways to determine whether a constructed model performs as expected is by visualization. Looking at the orientation of bodies relative to each other at the initial time can help identify some obvious errors in a model. Observing an animation of the system in motion can reveal more about the model. The M-file `anim` takes the output of `dap` and provides an animation of the simulated response. Additional points or outlines for improving the presentation of an animation, but are not required for the analysis, can be provided in the M-file `inAnimate`. In this file we may provide data for shape and color of bodies, additional points for better visualization, and plot parameters. There is no new structure associated with this file—we use the existing structures of bodies and points.

The program `anim` displays the body mass centers and all the defined points as small circles. The program draws lines between a mass center and every point that is defined on that body. The assigned color to a body will be used for the lines as well (the default color is *black*). We can specify a different color, based on MATLAB's color abbreviations.

We may choose a shape, out of the three available shapes shown in Figure B.9, and assign it to a body. For any of these shapes, the program assumes that the geometric center of the shape is the origin of the $\xi-\eta$ frame (mass center). The data for a shape is saved in additional fields in the body structure as stated in Table B.7, and additional explanation in Table B.8.

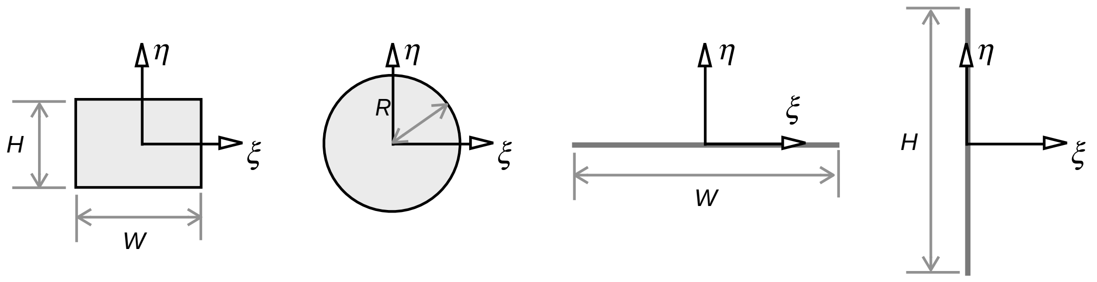


Figure B.9 Available standard shapes.

Table B.7

```
function Body = Body_struct
Body = struct ( ...
    ...
    'shape' , ' '   , ... % 'circle', 'rect', ‘line’
    'R'     , 1     , ... % radius of the circle
    'W'     , 0     , ... % width of the rectangle
    'H'     , 0     , ... % height of the rectangle
    'color' , 'k'   , ... % default color for the body
    ...                    );
```

Table B.8

| Shape | Required statements | Comments |
|---|---|---|
| Rectangle | `Bodies(i).shape = ‘rect’;` `Bodies(i).W = 0.4;` `Bodies(i).H = 0.2;` | |
| Circle | `Bodies(i).shape = ‘circle’;` `Bodies(i).R = 1.2;` | |
| Line | `Bodies(i).shape = ‘line’;` `Bodies(i).W = 1.4;` | Line along the $\xi$ -axis |
| Line | `Bodies(i).shape = ‘line’;` `Bodies(i).H = 0.8;` | Line along the $\eta$ -axis |

In addition to the defined attachment points of joints and application points of force elements, other points could be defined on a body for enhancing the presentation of its *nonstandard* shape. As an example, assume the triangular shaped body shown in Figure B.10(a) is connected to other bodies at pin joints *A* and *B*. With only two points, the body will be displayed as shown in Figure B.10(b). We may define a third point, such as *C*, to display the shape shown in Figure B.10(c), which may be a better description for a triangle.

Any point that is defined in the M-file `inAnimate` must be saved in the array `Points_anim`. These points are in addition to the points that have already been defined in M-file `inPoints` and saved in the array `Points`. The array `Points_anim` will be appended to the array `Points` for the animation.

For a rotating circular body, such as a rolling disc, we may want to define one arbitrary point on its circumference. The constructed line, as shown in Figure B.10(d), will provide a reference to realize the rotation of the circular body.

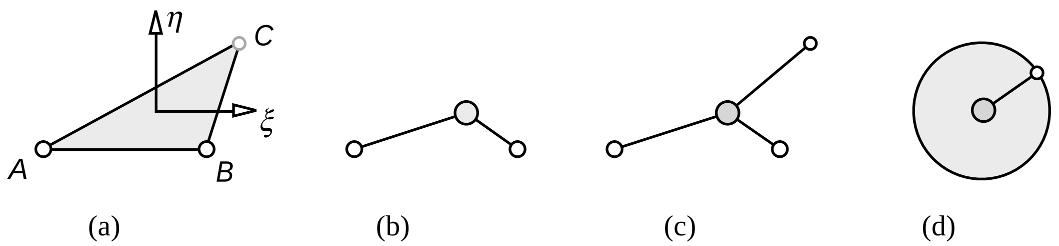


Figure B.10 Defining additional points for better visualization of animated objects.

In the M-file `inAnimate` we must set the scaling for the axes of the animation plot window. The scaling should be based on the dimensions used in a model and the expected range of its motion.

**Example B.1 (**cont.)

In this example there is no need to define any new points for animation. Therefore, we set `Points_anim` to be a *blank* array.

We have set the shape of body (*1*) to be a rectangle having a width of 0.4 and a height of 0.2. The

```
function inAnimate
    include_global

Points_anim = [];

Bodies(1).shape = 'rect';
Bodies(1).W = 0.4;
```

shape of body (2) is a line along the body's $\eta$-axis, with a length (height) of 1.0 unit.

The scaling parameters for the animation plot window are also defined in this file.

```
Bodies(1).H = 0.2;

Bodies(2).shape = 'line';
Bodies(2).H = 1.0;

xmin = -0.5; xmax =  2.0;
ymin = -1.0; ymax =  0.5;
```

## B.3 POSTPROCESSING

The program `dap` simulates the dynamic response of a system by integrating the equations of motion from the initial time $t = 0$ to the final time stated by the user. The program uses the MATLAB's integrator `ode45`, which reports the results at every $\Delta t$ seconds (time period) as specified by the user[1]. At every reporting time step, the integrator saves the time in an array named `T`, and saves the coordinates and velocities for all the bodies in an array named `uT`. None of the other results, such as accelerations and Lagrange multipliers, are saved. The coordinates and velocities are saved column-wise. For example, if we integrate the equations of motion of the sliding-pendulum system that contains two bodies for 4.0 seconds, and ask for $\Delta t = 0.02$ reporting intervals, the arrays will be formed as listed in Table B.9. Note that for $n_b$ bodies, there will be $6n_b$ columns in the array `uT`.

Table B.9 Arrays `T` and `uT` contain the output from the integrator `ode45`.

| $t$ |
|---|
| 0.00 |
| 0.02 |
| ... |
| 3.98 |
| 4.00 |

| $x_1$ , | $y_1$ | $\phi_1$ | $x_2$ | $y_2$ | . . . . | $\dot{x}_2$ | $\dot{y}_2$ | $\dot{\phi}_2$ |
|---|---|---|---|---|---|---|---|---|
| 1.00 | 0.20 | 0.00 | 1.25 | -0.23 | . . . . | 0.00 | 0.00 | 0.00 |
| 0.99 | 0.20 | 0.00 | 1.24 | -0.23 | . . . . | -0.06 | -0.03 | -0.13 |
| | | . . . . | | | . . . . | | . . . . | |
| 1.06 | 0.20 | 0.00 | 1.06 | -0.30 | . . . . | 0.21 | 00.01 | 1.32 |
| 1.05 | 0.20 | 0.00 | 1.06 | -0.29 | . . . . | 0.20 | 0.01 | 1.37 |

The program `post` takes the computed values for the coordinates and velocities from `uT` at each reporting time step. It recovers the missing results by solving the equations of motion again at every time step, and then saves all the results in several arrays as listed in

**Note:** In this version of the program the potential energy is only computed for the gravitational force and the point-to-point springs with linear characteristics. Contributions to the potential energy from other conservative forces could be included in the M-file `post` by the user.

Table B.10. This program also computes the kinetic energy, the potential energy, and the total energy at every time steps and saves the results. The user can refer to these arrays to perform further analyses with the simulated results.

**Note:** In this version of the program the potential energy is only computed for the gravitational force and the point-to-point springs with linear characteristics. Contributions to the potential energy from other conservative forces could be included in the M-file `post` by the user.

Table B.10

| Array | Size | Description |
|---|---|---|
| `r` | `nt × nB × 2` | Translational coordinates |

[1] The reporting time step is not the same as the integration time step. Integration time step used by `ode45` is variable and most likely much smaller than the reporting time step.

| | | |
|---|---|---|
| `rd` | `nt × nB × 2` | Translational velocities |
| `rdd` | `nt × nB × 2` | Translational accelerations |
| `p` | `nt × nB` | Rotational coordinates |
| `pd` | `nt × nB` | Rotational velocities |
| `pdd` | `nt × nB` | Rotational acceleration |
| `rP` | `nt × nP × 2` | Coordinates of points |
| `rPd` | `nt × nP × 2` | Velocity of points |
| `Jac` | `nt × nConst × nB3` | Jacobian matrix |
| `Lam` | `nt × nConst` | Lagrange multipliers |
| `eng` | `nt × 3` | Energy (kinetic, potential, total) |

| | |
|---|---|
| `nt` | Number of time steps (rows) including $t = 0$ |
| `nB` | Number of bodies |
| `nP` | Number of points |
| `nConst` | Number of constraints |
| `nB3` | `3` times `nB` |

**Examples**
To plot the acceleration of body (3) in the $y$-direction; that is $\ddot{y}_3$, versus time, we use:
```
plot(T,rdd(:,3,2))
```
To plot the $x$-component of the velocity of the 5th point in the model versus time, we use:
```
plot(T,rPd(:,5,1))
```
To plot the kinetic, potential, and total energies, we can use:
```
plot(T,eng(:,1)'r')
hold on
plot(T,eng(:,2),'g')
plot(T,eng(:,3))
```

## B.4 APPLICATION EXAMPLES

These example models are discussed in Chapter 8 of the textbook. All the models reside in the folder `Models`.

### B.4.1 Double A-Arm Suspension (DAP_BC/`Models`/AA)

Refer to Section 8.1.1 of the textbook for figures and further discussion.

The input M-files are developed according to the body and point indices that are discussed in Section 8.1.1. In the M-file `inForces` two sets of initial conditions for the coordinates are provided: one correct set and one incorrect set. In the M-file `inForces` we state that the second force elements is described in a user supplied force function (`user_force`) where we provide the logic to determine whether the tire is in contact with the ground, and if so, to determine the tire force.

```
function inBodies
    include_global

B1 = Body_struct;
B1.m = 2;
B1.J = 0.5;
```

```
function inPoints
    include_global

Q1 = Point_struct;
Q1.Bindex = 1;
Q1.sPlocal = [-0.24; 0];
```

```
B1.r = [0.4398; 0.2512]; % correct
B1.p = -0.0367; % correct
% B1.r = [0.51; 0.28]; % incorrect
% B1.p = 340*pi/180; % incorrect

B2 = Body_struct;
B2.r = [0.6817; 0.3498]; % correct
B2.p = 0.0783; % correct
% B2.r = [0.75; 0.35]; % incorrect
% B2.p = 0; % incorrect
B2.m = 30;
B2.J = 2.5;

B3 = Body_struct;
B3.r = [0.4463; 0.4308]; % correct
B3.p = 6.5222; % correct
% B3.r = [0.49; 0.41]; % incorrect
% B3.p = 350*pi/180; % incorrect
B3.m = 1;
B3.J = 0.5;

Bodies = [B1; B2; B3];
```

```

A1 = Point_struct;
A1.Bindex = 1;
A1.sPlocal = [0.18;0];

A2 = Point_struct;
A2.Bindex = 2;
A2.sPlocal = [-0.07;-0.10];

B2 = Point_struct;
B2.Bindex = 2;
B2.sPlocal = [-0.10;0.12];

B3 = Point_struct;
B3.Bindex = 3;
B3.sPlocal = [0.13;0];

O3 = Point_struct;
O3.Bindex = 3;
O3.sPlocal = [-0.13;0];

O0 = Point_struct;
O0.Bindex = 0;
O0.sPlocal = [0.32;0.40];

Q0 = Point_struct;
Q0.Bindex = 0;
Q0.sPlocal = [0.20;0.26];

E1 = Point_struct;
E1.Bindex = 1;
E1.sPlocal = [0;0];

F0 = Point_struct;
F0.Bindex = 0;
F0.sPlocal = [0.38;0.43];

Points = [Q1; A1; A2; B2; ...
     B3; O3; O0; Q0; E1; F0];
```

```
function inJoints
    include_global

J1 = Joint_struct;
J1.type = 'rev'; %Q
J1.iPindex = 1;
J1.jPindex = 8;

J2 = Joint_struct;
J2.type = 'rev'; % A
J2.iPindex = 2;
J2.jPindex = 3;

J3 = Joint_struct;
J3.type = 'rev'; % B
J3.iPindex = 4;
J3.jPindex = 5;
```

```
function inForces
    include_global

S1 = Force_struct;
S1.type = 'ptp';
S1.iPindex = 9;
S1.jPindex = 10;
S1.k = 90000;
S1.L0 = 0.23;
S1.dc = 1100;

S2 = Force_struct;
S2.type = 'user';
S2.k = 50000;
S2.L0 = 0.35;
S2.dc = 1000;

S3 = Force_struct;
```

```
J4 = Joint_struct;
J4.type = 'rev'; % O
J4.iPindex = 6;
J4.jPindex = 7;

Joints = [J1; J2; J3; J4];
```

```
S3.type = 'weight';

Forces = [S1; S2; S3];
```

```
function user_force
    include_global

% Unilateral spring-damper representing the radial tire force
    del = Bodies(2).r(2) - Forces(2).L0;
if del < 0
    fy = Forces(2).k*del + ...
         Forces(2).dc*Bodies(2).r_d(2);
    fsd = [0; -fy];
    Bodies(2).f = Bodies(2).f + fsd;
end

% ...
% del = Bodies(2).r(2) – 0.35;
% ...
% fy = 40000*del + 1000*Bodies(2).r_d(2);
% ...
```

Following a simulation, to plot the response for one of the coordinates, for example the *y*-coordinate of body (*2*) versus time, we can use the following command:

```
plot(T, uT(:,5))
```

Or, we can execute the M-file `post` and then use this command:

```
plot(T, r(:,2,2))
```

### B.4.2 MacPherson Suspension (`DAP_BC/Models/MP_A, MP_B, MP_C`)

Refer to Section 8.1.2 of the textbook for figures and further discussion.

**`DAP_BC/Models/MP_A`**: This model consists of 3 moving bodies, 3 pin joints, and 1 translational joint. The input M-files are developed according to the body and point indices that are discussed in Section 8.1.2.

```
function inBodies
    include_global

B1 = Body_struct;
B1.r = [0.5840; 0.3586];
B1.p = 6.0819;
B1.m = 20;
B1.J = 2.5;

B2 = Body_struct;
B2.m = 2;
B2.J = 0.5;
B2.r = [0.3450; 0.2900];
B2.p = 0;

B3 = Body_struct;
```

```
function inPoints
    include_global

A1 = Point_struct;
A1.Bindex = 1;
A1.sPlocal = [ 0.00; -0.07];

B1 = Point_struct;
B1.Bindex = 1;
B1.sPlocal = [-0.17;  0.25];

C1 = Point_struct;
C1.Bindex = 1;
C1.sPlocal = [ 0.11; -0.02];

O0 = Point_struct;
```

```
B3.r = [0.4528; 0.6862];
B3.p = 5.0019;
B3.m = 0.5;
B3.J = 0.2;

Bodies = [B1; B2; B3];
```

```
O0.Bindex = 0;
O0.sPlocal = [ 0.41;  0.83];

Q0 = Point_struct;
Q0.Bindex = 0;
Q0.sPlocal = [ 0.12; 0.29];

Q2 = Point_struct;
Q2.Bindex = 2;
Q2.sPlocal = [-0.225; 0.00];

A2 = Point_struct;
A2.Bindex = 2;
A2.sPlocal = [ 0.225; 0.00];

O3 = Point_struct;
O3.Bindex = 3;
O3.sPlocal = [-0.15;  0.00];

Points = [A1; B1; C1; O0; ...
     Q0; Q2; A2; O3];
```

```
function inUvectors
    include_global

V1 = Unit_struct;
V1.Bindex = 1;
V1.ulocal  = [0.47; -0.88];

V2 = Unit_struct;
V2.Bindex = 3;
V2.ulocal  = [1; 0];

Uvectors = [V1; V2];
```

```
function inJoints
    include_global

J1 = Joint_struct;
J1.type = 'rev'; %A
J1.iPindex = 1;
J1.jPindex = 7;

J2 = Joint_struct;
J2.type = 'rev'; % Q
J2.iPindex = 5;
J2.jPindex = 6;

J3 = Joint_struct;
J3.type = 'rev'; % O
J3.iPindex = 4;
J3.jPindex = 8;

J4 = Joint_struct;
J4.type = 'tran'; % O-B
J4.iUindex = 1;
J4.jUindex = 2;
J4.iPindex = 2;
J4.jPindex = 8;

Joints = [J1; J2; J3; J4];
```

```
function inForces
    include_global

S1 = Force_struct;
S1.type = 'ptp';
S1.iPindex = 2; % B1
S1.jPindex = 4; % O0
S1.k = 20000;
S1.L0 = 0.34;
```

```
function user_force
    include_global

del = Bodies(1).r(2) - Forces(2).L0;

if del < 0
    fy = Forces(2).k*del +
Forces(2).dc*Bodies(1).r_d(2);
    fsd = [0; -fy];
```

```
S1.dc = 1100;

S2 = Force_struct;
S2.type = 'user'; % tire
S2.k = 100000;
S2.L0 = 0.30;
S2.dc = 1000;

S3 = Force_struct;
S3.type = 'weight';

Forces = [S1; S2; S3];
```

```
    Bodies(1).f = Bodies(1).f + fsd;
    Bodies(1).n = Bodies(1).n + ...
           s_rot(Points(3).sP)'*fsd;
end
```

Following a simulation, we can plot the coordinate and velocity of the mass center of body (*2*) versus time using the following commands:

```
      plot(T, uT(:,5))
```

Or,

```
      plot(T, uT(:,14))
```

We note that since the model contains 3 bodies, $y_2$ is in the 5th column of the `uT` matrix and $\dot{y}_2$ is in the 14th column.

We can also execute `post` and then plot the *y*-coordinate of point *C* versus time using the command:

```
      plot(T,rP(:,3,2))
```

Or use the following command for the velocity of C in the *y*-direction.

```
      plot(T,rPd(:,3,2))
```

We note that *C* is point number [3].

**`DAP_BC/Models/MP_B`**: This model consists of 2 moving bodies, 2 pin joints, and 1 revolute-translational joint. The input M-files are developed according to the body and point indices that are discussed in Section 8.1.2.

```
function inBodies
    include_global

B1 = Body_struct;
B1.r = [0.5840; 0.3586];
B1.p = 6.0819;
B1.m = 20;
B1.J = 2.5;

B2 = Body_struct;
B2.m = 2;
B2.J = 0.5;
B2.r = [0.3450; 0.2900];
B2.p = 0;

Bodies = [B1; B2];
```

```
function inPoints
    include_global

A1 = Point_struct;
A1.Bindex = 1;
A1.sPlocal = [ 0.00; -0.07];

B1 = Point_struct;
B1.Bindex = 1;
B1.sPlocal = [-0.17;  0.25];

C1 = Point_struct;
C1.Bindex = 1;
C1.sPlocal = [ 0.11; -0.02];

O0 = Point_struct;
O0.Bindex = 0;
O0.sPlocal = [ 0.41;  0.83];

Q0 = Point_struct;
Q0.Bindex = 0;
Q0.sPlocal = [ 0.12; 0.29];

Q2 = Point_struct;
Q2.Bindex = 2;
```

```
Q2.sPlocal = [-0.225; 0.00];

A2 = Point_struct;
A2.Bindex = 2;
A2.sPlocal = [ 0.225; 0.00];

Points = [A1; B1; C1; O0; ...
     Q0; Q2; A2];
```

```
function inUvectors
    include_global

V1 = Unit_struct;
V1.Bindex = 1;
V1.ulocal  = [0.47; -0.88];

Uvectors = [V1];
```

```
function inJoints
    include_global

J1 = Joint_struct;
J1.type = 'rev'; %A
J1.iPindex = 1;
J1.jPindex = 7;

J2 = Joint_struct;
J2.type = 'rev-tran'; % O-B
J2.iPindex = 2;
J2.jPindex = 4;
J2.iUindex = 1;

J3 = Joint_struct;
J3.type = 'rev'; % Q
J3.iPindex = 5;
J3.jPindex = 6;

Joints = [J1; J2; J3];
```

```
function inForces
    include_global

S1 = Force_struct;
S1.type = 'ptp';
S1.iPindex = 2; % B1
S1.jPindex = 4; % O0
S1.k = 20000;
S1.L0 = 0.34;
S1.dc = 1100;
S1.f_a = 0;

S2 = Force_struct;
S2.type = 'user'; % tire
S2.k = 100000;
S2.L0 = 0.30;
S2.dc = 1000;

S3 = Force_struct;
S3.type = 'weight';

Forces = [S1; S2; S3];
```

```
function user_force
    include_global

del = Bodies(1).r(2) - Forces(2).L0;

if del < 0
    fy = Forces(2).k*del +
Forces(2).dc*Bodies(1).r_d(2);
    fsd = [0; -fy];
    Bodies(1).f = Bodies(1).f + fsd;
    Bodies(1).n = Bodies(1).n + ...
           s_rot(Points(3).sP)'*fsd;
end
```

**`DAP_BC/Models/MP_C`**: This model consists of 1 moving body, 1 revolute-revolute and 1 revolute-translational joint. The input M-files are developed according to the body and point indices that are discussed in Section 8.1.2.

```
function inBodies
    include_global

B1 = Body_struct;
B1.r = [0.5840; 0.3586];
B1.p = 6.0819;
B1.m = 20;
B1.J = 2.5;

Bodies = [B1];
```

```
function inPoints
    include_global

A1 = Point_struct;
A1.Bindex = 1;
A1.sPlocal = [ 0.00; -0.07];

B1 = Point_struct;
B1.Bindex = 1;
B1.sPlocal = [-0.17;  0.25];

C1 = Point_struct;
C1.Bindex = 1;
C1.sPlocal = [ 0.11; -0.02];

O0 = Point_struct;
O0.Bindex = 0;
O0.sPlocal = [ 0.41;  0.83];

Q0 = Point_struct;
Q0.Bindex = 0;
Q0.sPlocal = [ 0.12; 0.29];

Points = [A1; B1; C1; O0; Q0];
```

```
function inUvectors
    include_global

V1 = Unit_struct;
V1.Bindex = 1;
V1.ulocal  = [0.47; -0.88];

Uvectors = [V1];
```

```
function inJoints
    include_global

J1 = Joint_struct;
J1.type = 'rev-rev'; % Q-A
J1.iPindex = 1;
J1.jPindex = 5;
J1.L = 0.45;

J2 = Joint_struct;
J2.type = 'rev-tran'; % O-B
J2.iPindex = 2;
J2.jPindex = 4;
J2.iUindex = 1;

Joints = [J1; J2];
```

```
function inForces
    include_global

S1 = Force_struct;
S1.type = 'ptp'; % default
S1.iPindex = 2; % B1
S1.jPindex = 4; % O0
S1.k = 20000;
S1.L0 = 0.34;
S1.dc = 1100;
S1.f_a = 0;
```

```
function user_force
    include_global

del = Bodies(1).r(2) - Forces(2).L0;

if del < 0
    fy = Forces(2).k*del +
Forces(2).dc*Bodies(1).r_d(2);
    fsd = [0; -fy];
    Bodies(1).f = Bodies(1).f + fsd;
    Bodies(1).n = Bodies(1).n + ...
```

```

S2 = Force_struct;
S2.type = 'user'; % tire
S2.k = 100000;
S2.L0 = 0.30;
S2.dc = 1000;

S3 = Force_struct;
S3.type = 'weight';
Forces = [S1; S2; S3];
```

```
                  s_rot(Points(3).sP)'*fsd;
end
```

## B.4.3 Cart (DAP_BC/Models/Cart_A, Cart_B, Cart_C)

Refer to Section 8.1.3 of the textbook for figures and further discussion.

**`DAP_BC/Models/Cart_A`**: This model consists of 3 moving bodies, 2 pin joints, 2 disc joints with no-slip condition, and a driver constraint for the motor. The input M-files are developed according to the body and point indices that are discussed in Section 8.1.3. It is assumed that the rear wheel, body (*2*), rolls with a constant angular velocity of $2\pi$ rad/sec in the clockwise direction and zero angular acceleration.

A *relative-rotation* constraint is defined as the driver. This constraint acts on body (2) and it refers to “function 1” for its analytical description and parameters. In the M-file `inFunct` we state the function type as “a” and, therefore, its parameters are: zero initial angle, $-2\pi$ (clockwise) angular velocity, and zero angular acceleration.

```
function inBodies
    include_global

B1 = Body_struct;
B1.r = [0.5840; 0.3586];
B1.p = 6.0819;
B1.m = 20;
B1.J = 2.5;

B2 = Body_struct;
B2.m = 2;
B2.J = 0.5;
B2.r = [0.3450; 0.2900];
B2.p = 0;

B3 = Body_struct;
B3.r = [0.4528; 0.6862];
B3.p = 5.0019;
B3.m = 0.5;
B3.J = 0.2;

Bodies = [B1; B2; B3];
```

```
function inPoints
    include_global

P1 = Point_struct;
P1.Bindex = 1;
P1.sPlocal = [-0.3; -0.1];

P2 = Point_struct;
P2.Bindex = 1;
P2.sPlocal = [ 0.3; -0.1];

P3 = Point_struct;
P3.Bindex = 2;
P3.sPlocal = [ 0; 0];

P4 = Point_struct;
P4.Bindex = 3;
P4.sPlocal = [ 0; 0];

Points = [P1; P2; P3; P4];
```

```
function inJoints
    include_global

J1 = Joint_struct;
J1.type = 'rev';
J1.iPindex = 1;
J1.jPindex = 3;
```

```
function inForces
    include_global

F1 = Force_struct;
F1.type = 'weight';  % include the
weight

Forces = [F1];
```

```
J2 = Joint_struct;
J2.type = 'rev';
J2.iPindex = 2;
J2.jPindex = 4;

J3 = Joint_struct;
J3.type = 'disc';
J3.iBindex = 2;
J3.R = 0.1;
J3.x0 = 0.2;

J4 = Joint_struct;
J4.type = 'disc';
J4.iBindex = 3;
J4.R = 0.1;
J4.x0 = 0.8;
J4.p0 = 0; % default

J5 = Joint_struct;
J5.type = 'rel-rot'; % motor driver
J5.iBindex = 2;
J5.jBindex = 1;
J5.iFunct = 1;

Joints = [J1; J2; J3; J4; J5];
```

```
function inFuncts
    include_global

F1 = Funct_struct;
F1.type = 'a';
F1.coeff = [0 -2*pi 0];

Functs = [F1];
```

**`DAP_BC/Models/Cart_B`**: This model is the same as the model of Cart_A except for the driver constraint that refers to function "c" instead of "a".

```
function inFuncts
    include_global

F1 = Funct_struct;
F1.type = 'c';
F1.t_end = 2.0;
F1.dfdt_end = -2*pi;

Functs = [F1];
```

After executing `post` we can plot the Lagrange multiplier associated with the driver constraint, the no-slip constraint on the rear wheel, and the no-slip constraint on the front wheel using the following commands respectively:

```
>> plot(T,Lam(:,9))
>> figure
>> plot(T,Lam(:,6))
>> hold on
>> plot(T,Lam(:,8),'r')
```

**`DAP_BC/Models/Cart_C`**: In this model we consider an applied torque on the rear wheel, where it is assumed that the torque is generated by an electric motor with a known torque-speed characteristic. We remove the driver constraint (`rel-rot`) from the `inJoints` M-file of the original model. In the M-file `inForces` we add a second force element as `user` type. We then provide the following `user_force` M-file. In this file, based on the angular velocity of the rear wheel and the torque-speed characteristics of the motor we determine the torque of the motor and apply it on the wheel. Note that we apply this torque with a negative sign since we want to rotate the wheel CW and move the cart from left to right.

```
function user_force
    include_global
% Motor
    omega_max = 4*pi; T_max = 20;
    omega = abs(Bodies(2).p_d);
    T_motor = T_max*(1 - omega/omega_max);
    if T_motor > T_max
        T_motor = T_max;
    end
    Bodies(2).n = Bodies(2).n - T_motor;
    Bodies(1).n = Bodies(1).n + T_motor;
```

**`Models/Cart_D`**: In this model we add a resistive force to the model `Cart_C`. This resistive force is added to the `user_force` M-file and include it in the array of force in the *x*-direction for body (*1*).

```
function user_force
    ...
% Aerodynamic resistive force
    damp_aero = 10;
    x_d = Bodies(1).r_d(1);
    f_aero = damp_aero*x_d^2;
    Bodies(1).f(1) = Bodies(1).f(1) - f_aero;
```

### B.4.4 Conveyor Belt and Friction (DAP_BC/Models/CB)

Refer to Section 8.1.4 of the textbook for figures and further discussion.

```
function inBodies
    include_global

B1 = Body_struct;
B1.m = 1.0;
B1.J = 1.0;
B1.r = [1.0; 0.2];
B1.r_d = [ 0.0; 0.0];

Bodies = [B1];
```

```
function inPoints
    include_global

P1 = Point_struct;
P1.Bindex = 0;
P1.sPlocal = [ 0; 0.2];

P2 = Point_struct;
P2.Bindex = 1;
P2.sPlocal = [ 0; 0];

Points = [P1; P2];
```

```
function inUvectors
    include_global

U1 = Unit_struct;
```

```
function inJoints
    include_global

J1 = Joint_struct;
```

```
U1.Bindex = 0;
U1.ulocal  = [ 1.0; 0];

U2 = Unit_struct;
U2.Bindex = 1;
U2.ulocal  = [ 1.0; 0];

Uvectors = [U1; U2];
```

```
J1.type = 'tran';
J1.iPindex = 2;
J1.jPindex = 1;
J1.iUindex = 2;
J1.jUindex = 1;

Joints = [J1];
```

```
function inForces
    include_global

F1 = Force_struct;
F1.type = 'ptp';
F1.iPindex = 2;
F1.jPindex = 1;
F1.k = 10;
F1.L0 = 0.8;
F1.dc = 0;

F2 = Force_struct;
F2.type = 'weight';

F3 = Force_struct;
F3.type = 'user';

Forces = [F1; F2; F3];
```

```
function user_force
   include_global

% Anderson et al. friction model
   mu_d = 0.15; mu_s = 0.2; k_v = 0.0;
   v_s = 0.001; p = 2; k_t = 10000;
   fy = 9.81; % normal force
   v_conv = 0.1;
   v = v_conv - Bodies(1).r_d(1);
   ff = Friction_A(mu_s, mu_d, v_s, p,k_t,v);
   fx = fy*(ff + k_v*v);
   fs = [fx; 0];
   Bodies(1).f = Bodies(1).f + fs;
```

### B.4.5 Rod Impacting the Ground (DAP_BC/Models/Rod)

Refer to Section 8.1.5 of the textbook for figures and further discussion.

```
function inBodies
    include_global

B1 = Body_struct;
B1.m = 1.0;
B1.J = 0.01;
B1.r = [0; 1];
B1.p = pi/4;
B1.r_d = [0; -6];

Bodies = [B1];
```

```
function inPoints
    include_global

P1 = Point_struct;
P1.Bindex = 1;
P1.sPlocal = [ 0; -1];

P2 = Point_struct;
P2.Bindex = 1;
P2.sPlocal = [ 0; 1];

Points = [P1; P2];
```

```
function inForces
    include_global

S1 = Force_struct;
S1.type = 'weight';

S2 = Force_struct;
S2.type = 'user';

Forces = [S1; S2];
```

```
function user_force
    include_global
    global pen1_d0 pen2_d0

% Parameters for the contact model
    k = 10^11; e = 0.95;

% Contact(Contact index, Point index, Body index, k, e, Model index)
%    Model-index = 1: Eq. 11.42
%    Model-index = 2: Eq. 11.43

% Point [1] on Body (1)
    Contact(1, 1, 1, k, e, 1)
% Point [2] on Body (1)
    Contact(2, 2, 1, k, e, 1)
```

**Note**: To observe the peaks of the acceleration response, we must select the reporting time steps to be very small, such as 0.0001 seconds.