Trajectory Planning for the Halobject

P.M. Nagelmaeker

DCT 2005.105

Traineeship report

dr. ir. G.W.M. Peters

Supervisor:  prof. dr. ir. M. Steinbuch

Technische Universiteit Eindhoven
Department Mechanical Engineering
Dynamics and Control Technology Group

Eindhoven, July, 2005
Contents

1 Introduction 4

2 Introduction to Motion Controllers 5
  2.1 Globally 5
  2.2 Trajectory planning 6
    2.2.1 Second order trajectory planning 6
    2.2.2 Feedforward design 7
    2.2.3 Third order trajectory planning 8

3 Trajectory Planning for the Walking Mechanism 10
  3.1 Introduction 10
  3.2 Trajectory planning 11
    3.2.1 Second order trajectory planning 14
    3.2.2 Third order trajectory planning 14
    3.2.3 Trajectory output turning motor 16
    3.2.4 Trajectory output cantilever motor 17
    3.2.5 Examples from simulation 19

4 Trajectory Planning for the Feelers Mechanism 20
  4.1 Introduction 20
  4.2 Trajectory planning 20
    4.2.1 Second order trajectory planning 23
    4.2.2 Trajectory output 23

5 Conclusion and recommendations 29

Bibliography 30

A States of the trajectory planning feelers 31
  A.1 State: Scan Mode and State: Stop Scan Mode A 31
  A.2 State: Stop Scan Mode B 33
  A.3 State: Start Pointing Mode B and State: Pointing Mode and State: Start Pointing Mode C 34
  A.4 State: Stop Pointing Mode 35
  A.5 State: Start Pointing Mode B 36
B C-source

B.1 Walking mechanism
   B.1.1 TP2.c ......................................................... 37
   B.1.2 TP3_C.c ....................................................... 39
   B.1.3 T02.c .......................................................... 41
   B.1.4 T03_C.c ........................................................ 44
B.2 Feeler mechanism .................................................. 50
   B.2.1 FTP2.c .......................................................... 50
   B.2.2 F102.c .......................................................... 51
Chapter 1

Introduction

The VARA/NPS broadcasting associations challenged a group of artists to revive one of the halls of the NET3 building in Hilversum. An artist came up with the idea to create a moving sculpture, for now called the Halobject. He turned to the University of Technology at Eindhoven (TU/e) with the request to develop the underlying technical infrastructure. The TU/e accepted this and the Halobject became one of the design projects for students to work on. Student groups studied on walking mechanisms, sensors, feelers and character for a year. Then a small group started the first realization phase of the ‘creature’. The complete framework, electrical system and operator panel were created. Furthermore the basic control software was developed. Currently the second phase is in progress. This phase focuses on creation of the needed software to let the Halobject walk autonomously in a space without bumping into stationary or moving objects. The final goal is a walking robot which interacts with its environment via sensors (light, sound, distance).

This report will discussed the trajectory planning developed for the Halobject. The trajectories are used as reference path for the electric motor to follow.
Chapter 2

Introduction to Motion Controllers

2.1 Globally

This chapter considers motion controllers and especially trajectory planning in a globally form. Its meant to give the reader a better understanding in the structure of a motion controller and to give insight in how trajectory planning works. Furthermore a simple design of a feedforward controller will be discussed. A combination of trajectory planning, feedback and feedforward control are often together constituted as motion controller.

The global structure of a motion controller is graphicly depicted in figure 2.1.

![General layout of motion controller](image)

Feedforward control is a well known technique for high performance motion control problems. It is, for instance, widely applied in robots.
An example of a motion task can be to perform a motion from a position A to a position B, starting at time $t$.
This task has to be performed by the motion controller which has several tasks:

- **Trajectory planning**
  The calculation of an allowable trajectory with respect to given constraints for each actuation device; for example a trajectory an electric motor has to follow, maybe better known as reference signal $r$;

- **Feedforward control**
  The calculation of a feedforward control signal on grounds of the calculated trajectory with the intention to obtain better tracking performance;
Feedback control

The processing of available measurements and calculation of a feedback control signal for each actuation device to compensate for unknown disturbances and unmodelled behavior.

In order to keep the design simple, these motion controller tasks are done for each actuator device separately. In this case, each actuating device is considered to be acting on a single mass moving along a single degree of freedom. The feedforward control problem is then to generate the force required to perform the acceleration of the mass according with the desired trajectory. However, the desired trajectory should be such that the required force is allowable from a mechanical point of view, and that no actuator saturation occurs.

The remaining part of this chapter will describe two different ways of trajectory calculation, namely the second and a third order trajectory planning. More detailed information about trajectory planning can be found in [1].

2.2 Trajectory planning

Consider the simple motion system as in figure 2.2 denoting m the mass, d the damping and k the stiffness of the system.

![Simple mass-damper-spring system](image)

The equation of motion is as follows:

\[ m\ddot{x} + d\dot{x} + kx = F \]  \hspace{1cm} (2.1)

2.2.1 Second order trajectory planning

With a given bound on acceleration \( \ddot{a} \), i.e. a bound on actuator force \( F \), we want a transition from current position A to desired position B over distance \( \dot{x} \). The shortest time within the motion can be performed is calculated as:

\[ \dot{x} = 2 \times \frac{1}{2} \ddot{a} t^2 \Rightarrow t_a = \sqrt{\frac{\dot{x}}{\ddot{a}}} \Rightarrow t_s = 2t_a \]  \hspace{1cm} (2.2)

with \( t_a \) denoting the constant acceleration time, \( t_s \) denoting the total trajectory executing time. At this point, the trajectory consist of a constant acceleration phase (\( \ddot{a} \)), followed by a constant deceleration phase (\( -\ddot{a} \)).
In case of a given bound on the velocity, denoted as $\bar{v}$, violation of $\bar{v}$ has to be checked by calculating the maximal reached velocity.

$$v = \bar{a}t_a$$  \hspace{1cm} (2.3)

If $\bar{v}$ is not violated:

$$t_x = 2t_a$$  \hspace{1cm} (2.4)

as in eq. 2.2.

If $\bar{v}$ is violated ($v \leq \bar{v}$):

$$t_a = \frac{\bar{v}}{\bar{a}} \Rightarrow x_a = 2 \times \frac{1}{2} \bar{a}t_a^2$$  \hspace{1cm} (2.5)

In order to reach the desired end position ($\bar{x} < x_a$) a constant velocity phase with duration $t_v$ has to be added:

$$t_v = \frac{\bar{x} - x_a}{\bar{v}}$$  \hspace{1cm} (2.6)

Now for the transition from A to B can be said:

$$t_x = 2t_a + t_v$$  \hspace{1cm} (2.7)

$$\bar{x} = \bar{a}t_a^2 + \bar{v}t_v$$  \hspace{1cm} (2.8)

When $v$ is not violated, $t_v$ in equation 2.8 turns zero.

Resuming, with given design constraint ($\bar{a}$ and $\bar{v}$), the second order trajectory algorithm calculates an acceleration profile which has to be integrated twice to get the desired trajectory profile of the position. See figure 2.3.

Based on the calculated profiles a feedforward controller can easily be designed. As stated before the feedforward force $F$ is calculated as in equation 2.7

$$F_{ff} = ma + dv + kx,$$ and represents the force which is needed to perform the desired trajectory. A typical feedforward design with PD-controller scheme can be seen in figure 2.4.

2.2.2 Feedforward design

Based on the calculated profiles a feedforward controller can easily be designed. As stated before the feedforward force $F$ is calculated as in equation 2.7, $F_{ff} = ma + dv + kx$, and represents the force which is needed to perform the desired trajectory. A typical feedforward design with PD-controller scheme can be seen in figure 2.4.
2.2.3 Third order trajectory planning

Compared to the second order trajectory profiles (step in acceleration profile), third order profiles (step in jerk \([\text{rad/s}^3]\)) have the advantage of being more smoothed. This implies a lower energy content at higher frequencies, which results in a lower high frequency content of the error signal and a decrease of the positioning error level. Furthermore this reduces the chance of demanding a motion which is physically impossible to perform by the given motion system (step in force not possible due to power amp limitation such as time rise).

Now the third order trajectory planning will be discussed. We will assume that a trajectory must be planned over a desired distance \(\bar{x}\) with respect to the given bounds on the jerk \((\bar{j})\), the acceleration \((\bar{a})\) and the velocity \((\bar{v})\). The symmetrical trajectory is completely determined by three time phases: the constant jerk phase \((t_j)\), the constant acceleration phase \((t_a)\) and the constant velocity phase \((t_v)\).

In the first place the constant jerk time is calculated, with no violation on \(\bar{a}\) and \(\bar{v}\) and assuming a symmetrical jerk profile:

\[
\bar{x} = 2jt_j^3 \implies t_j = \frac{3}{2}\sqrt[3]{\frac{\bar{x}}{2j}} \quad \text{and} \quad t_x = 4t_j
\]  

(2.9)

The maximum reached velocity will be:

\[
v = jt_j^2
\]

(2.10)

If \((v > \bar{v})\) the bound is violated and \(t_j\) must be calculated as follows:

\[
t_j = \sqrt[3]{\frac{\bar{v}}{j}}
\]

(2.11)

The maximum reached acceleration will be:

\[
a = j\bar{t}_j
\]

(2.12)

If \((a > \bar{a})\) the bound is violated and \(t_j\) must be recalculated:

\[
t_j = \frac{\bar{a}}{j}
\]

(2.13)

Now we have the maximal value for \(t_j\) with no bounds violated. The next step in the algorithm is to calculate \(t_a\). Assuming \(t_a > 0\) and \(\bar{v}\) is not violated we can say:

\[
\bar{x} = 2jt_j^3 + 3jt_j^2t_a + jt_jt_a^2 \implies
\]

(2.14)

\[
t_a = -\frac{3}{2}t_j + \frac{1}{2}\sqrt{t_j^2 + \frac{4\bar{x}}{jt_j}}
\]

(2.15)
The maximal reached velocity:

\[ v = \ddot{j}t_j^2 + \dddot{j}t_j t_a \quad (2.16) \]

If \( v > \ddot{v} \) the bound is violated and \( t_v \) must be recalculated as follows:

\[ t_a = \frac{\ddot{v} - \dddot{j}t_j^2}{\dddot{j}t_j} = \frac{\dddot{v}}{\dddot{a}} - t_j \quad (2.17) \]

Now we have calculated \( t_j \) and \( t_a \) to be the time-optimal solution under restriction of the given bounds. To reach the desired \( \ddot{x} \) we finally have to add a constant velocity phase. We can calculate \( t_v \):

\[ t_v = \frac{x - 2\ddot{j}t_j^3 - 3\dddot{j}t_j^2 t_a - \dddot{j}t_j^2 t_a^2}{\dddot{j}t_j} \quad (2.18) \]

and the total displacement \( \ddot{x} \) as function of the times \( t_j, t_a \) and \( t_v \):

\[ \ddot{x} = \dddot{j}(2t_j^3 + 3\dddot{j}t_j^2 t_a + \dddot{j}t_j^2 t_a^2 + t_j t_a t_v) \quad (2.19) \]

![Third order trajectory profiles](image)

Figure 2.5: Third order trajectory profiles

Integrating the calculated jerk profile three times will give the trajectory profile from point 0 to the desired end position. See figure 2.5.
Chapter 3

Trajectory Planning for the Walking Mechanism

3.1 Introduction

The walking mechanism of the Halobject [2] is a so called hexapod containing six legs which can be divided in three groups:

- The left legs. These two legs (left side, front and rear) always move synchronously and are driven with the left electric turning motor (70 Watt) and can only move forward and backwards.

- The right legs. Same as the left legs, only right.

- The cantilever leg (middle). The cantilever leg is driven by a 250 Watt electric motor. Turning this motor will result in lifting one side of the leg so the whole frame is lifted to one side. In lifted situation the Halobject is always resting on three points, the left or right legs (front and rear) and the opposite cantilever leg (middle).

![Diagram](image_url)

Figure 3.1: Frame construction of the walking mechanism
In figure 3.1 the walker mechanism is showed. The left-front and left-rear legs are interconnected by a driving-belt. A cogwheel which is mounted on the beam is driven by the electric motor. By turning this motor, the whole beam will turn around its middle axis and the left legs will make a forward or backward translation. The same principle is applied for the right legs.

![Figure 3.1: Walker Mechanism](image1)

In figure 3.2 the principle of the walking mechanism is showed.

- (a) The walking mechanism is in homing position. This position is determined by a homing procedure performed at each startup of the Halobject. All the six legs have contact with the ground. By tilting the left side of the Halobject with the cantilever and moving the left legs forward (which lost contact with the ground) the Halobject will start move forward.
- (b) The cantilever is tilted max and the left legs are still moving forward. The right legs have still contact with the floor.
- (c) The cantilever is again in its neutral position and all six legs have contact with the floor again. The Halobject has made a half step forwards. By repeating this for the right legs (not depicted here) the Halobject will move a next half step forward and return again in its homing position (a).

In the remaining part of the report those two steps (left and right, from homing position to homing position) will be called a walking cyclus. By decreasing the forward stroke of one side, turning is possible. When for example the forward stroke for the left turning motor is smaller than at the right side, the Halobject will turn left.

More detailed information about the walking mechanism can be found in [2].

### 3.2 Trajectory planning

Basically, the trajectory planning software for the walking mechanism has to translate the demands of the character into the right reference paths for the three electric motors so a desired walk behavior is established. The software in question is written in C. This c-file is integrated in MATLAB Simulink as an S-function. S-functions (system-functions) are a powerful mechanism for extending the capabilities of Simulink. This S-function has the following in- and outputs (see figure 4.3):

- **Inputs**
  1. Start [1/0]

    If this input turns true, the Halobject starts walking. The walking move has always to start from its homing position, obtained from the earlier stated homing procedure. If start turns zero again the movement has to stop in its homing position. So the Halobject completes
first the walking cyclus before it will stops. If start is true the software uses input 2...7 to calculate the three trajectories for each individual motor;

2. Step [rad]
   The variable step determines the stroke of the forward movement of the right and left turning legs. Due to geometric restriction this variable is limited to 0.7 radians;

3. Step height [rad]
   The variable step height prescribes the maximum tilting height of the cantilever and has to be expressed in radians. Due to geometric restriction this value is limited to 2.5 radians;

4. \( \alpha_{hal} [\text{deg}] \)
   \( \alpha_{hal} \) is the desired turning angle of the Halobject and represents the rotation of the Halobject after one walking cyclus and is expressed in degrees. In case of an unequal forward movement of both turning legs, the Halobject will turn. The value for the forward stroke of the outer-leg is equal to input 2, step. The value for the forward stroke of the inner-leg (which has to be smaller) is calculated using the geometry of the walking mechanism and the value for step. First the desired difference in stroke is calculated:

\[
\Delta S = w \tan(\alpha_{hal})
\]

with \( w \) the width of the walking mechanism. Now the desired step for the inner-leg can be calculated:

\[
\text{step}_{small} = 2 \arcsin(\sin(\text{step}/2) - \Delta S)
\]

The variable \( \alpha_{hal} \) is limited to -10 and +10 degrees to avoid contact of the right and left turning leg beams in case the difference between step and \( \text{step}_{small} \) gets to large. At least a positive value for \( \alpha_{hal} \) will result in a turn right.

5. Bound on acceleration [rad/s²]
   Bound on acceleration is used for second order trajectory planning for the turning motor left and right;

6. Bound on velocity [rad/s]
   Bound on velocity is used for second order trajectory planning for the turning motor left and right;

7. \( t_{rest} [\text{sec}] \)
   To make slow walking behavior possible, a rest period can be added which is inserted between the calculated trajectories. This will be discussed in detail in the following subsection.

8. Backward [t/o]
   If backward turns one the Halobject walks backward. If the Halobject is already walking, it will first finish the current step.

9. Emergency Stop [t/o]
   If a emergency stop is needed (for example an instruction from the obstacle avoidance software) the Halobject does not complete the walking cyclus but stops when the current step is finished.

10. Reset offset [t/o]
    In case of a emergency stop its possible that the Halobject has to stop outside the homing position (namely at the opposite of the homing position). In that case, a internal flag is set which will force the software to wait for Reset offset to turn true. If so, the Halobject steps back to reset this offset to return to the homing position.
• Outputs

1. Second order Trajectory turning motor left
   {position \([\text{rad}]\), velocity \([\text{rad/s}]\) and acceleration \([\text{rad/s}^2]\)}

2. Second order Trajectory turning motor right
   {position \([\text{rad}]\), velocity \([\text{rad/s}]\) and acceleration \([\text{rad/s}^2]\)}

3. Third order Trajectory cantilever motor
   {position \([\text{rad}]\), velocity \([\text{rad/s}]\), acceleration \([\text{rad/s}^2]\) and jerk \([\text{rad/s}^3]\)}
   The cantilever has to deal with a relative large mass displacement (i.e. tilting the Halobject). To avoid problems caused by applying a step in the acceleration profile a third order trajectory is calculated (with step in jerk).

In figure 3.3 the global structure of the software can be seen with the IO’s as described above. The software is divided in two different parts, the trajectory planning and the trajectory output.

![Figure 3.3: Global structure of the trajectory planning software](image)

The trajectory planning calculates a acceleration profile for both the turning motors and outputs a vector with the corresponding switching times (9) for the acceleration profiles \([a(9)]\) and the acceleration values \(a_{\text{left}}\) and \(a_{\text{right}}\) which only differ in case of cornering. The trajectory planning for the cantilever motor calculates a jerk profile and outputs the vector \(j(31)\) with the 31 switching times to generate the jerk profile. \(j(1)\) contains the corresponding jerk value. The trajectory output blocks for the two turning motors and the cantilever motor aims to output the acceleration and jerk profile as function of time. By integrating these profiles the desired outputs are achieved, namely AVP (Acceleration, Velocity, Position) for both turning motor and JAVP (Jerk, AVP) for the cantilever motor. To
avoid drift, these integrators are resetted when the velocity or the position value equals zero. An example of calculated trajectories as used for the walking mechanism is showed in figure 3.4.

In this example the reference path for the left turning motor is larger than the right reference path so a right turn will be achieved. The value for $\alpha_{hal}$ was 10 degrees what means that the Halobject executed a 10 degrees rotation within one walking cycle (6.55 sec). Furthermore can be seen that the cantilever leg tilts the Halobject during one walking cycle first on the left side, subsequently on the right side.

The trajectory planning and output which will generate such reference paths will be discussed in detail in the following four subsections.

### 3.2.1 Second order trajectory planning

The c-file TP2.c represents the block "Second order trajectory planning Turning motor". This file calculates a second order trajectory for the turning legs with respect to the desired step value, rest time $t_{rest}$ and cornering angle $\alpha_{hal}$. Moreover the bounds on the acceleration and velocity may not be exceeded. The calculated switching times needed for the acceleration profile are saved in the vector $at[9]$. The scalar $a[i]$ contains the corresponding value for the acceleration.

If no cornering is desired, the value $a[i]$ will be used for both $a_{left}[i]$ and $a_{right}[i]$. If cornering is desired, i.e. $\alpha_{hal} \neq 0$, the value from $a[i]$ is used for either $a_{left}[i]$ or $a_{right}[i]$ depending on which side is the outside. The outside leg which has to make the largest stroke, will use the value from $a[i]$. The stroke for the innerleg, which has to be smaller, is calculated with use of the desired cornering angle. As explained in the previous section the corresponding value for the inside forward stroke, $step\_small$ is calculated. Again with respect to the earlier mentioned input values (with exception for the value for step) the switching times for the acceleration profile are calculated and saved in the correct vector ($a_{left}[i]$ or $a_{right}[i]$). To reduce the computers calculation time as much as possible, these calculation are only performed once when new input value is achieved.

### 3.2.2 Third order trajectory planning

The c-file TP3.C represents the block "Third order trajectory planning Turning motor" from figure 3.3 and calculates a third order trajectory for the cantilever motor. In contrast with the above-
mentioned trajectory planning, as being the time optimal solution with the given constraints, the calculation of the trajectory for the cantilever differs from the standard method as described in chapter 2.2.3. In principle, the movement of the cantilever can be derived from the calculated trajectory of the turning legs. The cantilever has to tilt the Halobject twice during one walking cycle (figure 3.4). Only the time for one walk cycle is needed and this end-time is already calculated in the previous subsection. So this c-files uses the switching times information in vector $\text{at}[j]$ as calculated in TP2.c. In fact only the last element in this vector is needed which holds the duration of one walking cycle. Now by dividing this duration time in a symmetrical jerk profile a third order trajectory can be calculated such that it exactly matches the turning motor trajectory concerning the time-scale. In figure 3.5 an example of a generated jerk profile is depicted. By integrating this profile, the remaining paths are achieved.

![Diagram of Jerk Profile](image)

Figure 3.5: Trajectory planning for the cantilever

The jerk profile is generated as follows:
When we consider only the first part of the trajectory, namely the transition from 0 to 2 radians (step height, $t = 0...1.64$ sec). This part consists of:

- one constant velocity phase $t_v$;
- one constant acceleration and one constant deceleration phase $t_a$;
- two positive and two negative jerk phases $t_j$.

Now by choosing:

\[
  t_j = \frac{t_{end}}{72} \quad (3.3)
\]

\[
  t_a = 4t_j \quad (3.4)
\]

\[
  t_v = 6t_j \quad (3.5)
\]

the jerk profile as in figure 3.5 is generated such that the above mentioned velocity, acceleration and deceleration phase are equal in duration.
The structure of the jerk profile is now determined. Only the value for the jerk has to be calculated so the desired step height is achieved. Consider again the earlier mentioned first part of the trajectory which consist of the three equal phases $t_v$ and two times $t_a$ (so $t_v = t_a$). First a second order trajectory is calculated. The reached position of the first part is:

$$p = \frac{1}{2}a_2t_a^2 + a_2t_a^2 + \frac{1}{2}a_2t_a^2 = 2a_2t_a^2 \Rightarrow a_2 = \frac{p}{2t_a^2}$$  \(3.6\)

with $p$ equal to the desired step height and $a_2$ the desired acceleration value to reach this step height with a second order profile. Now we make the extension to a third order profile. The reached velocity in the second order case is:

$$v_2 = a_2 \cdot t_a$$  \(3.7\)

and in the third order case:

$$v_3 = \frac{1}{2}a_3 \cdot t_j + 4a_3 \cdot t_j = 5a_3 \cdot t_j$$  \(3.8\)

To reach the same velocity (and so the same end-position) as in the second order case, the needed acceleration value in third order case has to be larger:

$$v_2 = v_3 \Rightarrow a_2 \cdot t_a = 5a_3 \cdot t_j \Rightarrow 6a_2 \cdot t_j = 5a_3 \cdot t_j \Rightarrow a_3 = \frac{6}{5}a_2$$  \(3.9\)

Finally, to reach $a_3$ the jerk has to be:

$$j = \frac{a_3}{t_j}$$  \(3.10\)

The switching times information is saved in vector $jt[31]$ and the value for the jerk in scalar $j$. This information is send to the block "Trajectory output cantilever motor" where the actual reference paths are generated as a function of time (as in figure [3.3]) and is discusses in section 3.2.5. Again, these calculations are only performed once when new input is received and new trajectories have to be calculated.

If input backward turns one, the Halobject has to walk backwards. To do so the trajectory for the cantilever has to be multiplied with $-1$. Now the order of lifting each side is changed which will result in a backward walking movement.

Finally, one remark about equation\[3.3\] With a running frequency of the software of 1000 Hz, the resolution for plotting the jerk profile is 1 msec. When the calculated value for $t_j$ is between two samples rounding errors occur which will lead to a mismatch of the cantilever and the turning trajectories concerning the time-scale.

To avoid this problem, the value $t_{end}$ is rounded such that the division $\frac{1000 \cdot t_{end}}{t_j}$ outputs a integer. This holds a maximum rounding error of $72/2 = 36$ msec. Moreover this rounding occurs not in this c-file but in the c-file TP2.c where $t_{end}$ is calculated. This means that the turning and cantilever trajectory will match exactly and only a small difference occurs between the desired and calculated trajectory concerning the stroke. The maximal error in the position signal is $3.4e^{-3}$ radians which is acceptable for the walking mechanism.

### 3.2.3 Trajectory output turning motor

The c-file TP2.c represents the blocks "Trajectory Output Turning motor Right" and "Trajectory Output Turning motor Left". This c-files uses the earlier calculated acceleration profile information (at[39], a_right or a_left) to output the acceleration reference signal for the right and left turning motor. This signal is integrated once and twice to achieve respectively the velocity and position signal. It uses a counter triggered each sample time (1 msec) which is reseted at the end of each acceleration profile, i.e. walking cyclus. By using counters, the software has time information needed to output the trajectory signals, but independently from the absolute running time. As depicted in figure [3.3] the trajectory output software has to deal with three more inputs, that means 'start [1/0]', 'emergency stop [1/0]' and 'reset offset [1/0]'.
• **start [i/o]**
  If `start [i/o]` turns true the Halobject has to start walking. With use of `a(t)[9]` and `a_right` or `a_left` the acceleration profile is generated as function of time with use of the counter. By integration the position and velocity signal are achieved and the AVP-signal is constructed which represents the output of the T02.c. If during walking new acceleration profile information is received, i.e. a new trajectory, the walking cycle is first completed and the new trajectory is applied in the succeeding walking cycle. If `start [i/o]` turns zero again, the walk movement has to stop. First the walking cycle is finished so the Halobject will always stop in the homing position.

• **emergency stop [i/o]**
  As mentioned above, the Halobject will only stop walking after finishing the walking cycle. In case of an desired emergency stop ("soft" emergency stop) this might be a problem. When for example an object is detected in the blind zone of the ultrasonic sensors, i.e. a object detected within ± 60 cm of the Halobject, it is not advisable to finish the complete walking cycle. In a worst-case-scenario the forward movement of the Halobject after receiving an emergency stop can reach to 14 cm. Therefore the Halobject will in case of an estop not at all costs finish the walking cycle. The trajectory will be stopped as soon as the Halobject touches the ground with all his six legs, i.e at the half or at the end of the trajectory. In the second case the same situation occurs as a regular stop walking. However, in the first case additional software is needed there the Halobject stopped outside the homing position. The movement now has stopped in the opposite of the homing position (at \( t = 3.15 \text{ sec} \) in figure \[3.4\]). To deal with this offset from the homing position (for the turning legs) an additional input is created aiming to reset this offset, `reset offset`.
  Lastly, by introducing the emergency stop the worst-case-scenario forward movement is reduced to \( 34/2 = 17 \text{ cm} \).

• **reset offset [i/o]**
  In case of an offset to the homing position, the internal flag `offset` is set and by this the software is forced to wait for `reset offset` to turn true. If so, the last used trajectory is used to move the turning legs back to the homing position to reset the offset. This movement has to be combined with a corresponding movement of the cantilever to tilt the Halobject so it a back-step is achieved. This will be further discussed in the next section \[3.2.3\].
  So, without resetting the offset, no walking is allowed. The moment to reset this offset is a decision to be made for the central operating software ("character"). When `reset offset` always equals one, an eventually offset after estop, is immediately resetted.

### 3.2.4 Trajectory output cantilever motor

The block "Trajectory output cantilever motor" from figure \[3.3\] is represented by the c-file T03.C.c. The jerk profile information `j(t)[3]` and `j` calculated in TP3.C.c is used to output the jerk signal as function of time using a counter. By integration, the JAVP signal is constructed which is the output of this file. Just like "Trajectory output turning motor" the same input signals have to be processed:

• **start [i/o]**
  If `start [i/o]` turns true the Halobject has to start walking. As mentioned above the JAVP signals are generated as function of time. Just like "Trajectory Output Turning motor" the walking cycle is first finished after `start` turns false.

• **emergency stop [i/o]**
  As discussed earlier, after receiving an estop the trajectory stops only at the half or at the end of the walking cycle.
• *reset offset [i/o]*
  In case the turning legs have an offset to the homing position after an estop, the software is waiting for *reset offset* to turn true. If so the cantilever has to tilt the Halobject. In fact the last tilt move has to be repeated to let the Halobject step back and let the turning mechanism return in the homing position. In case the Halobject was already walking back when the estop was received, automatically a step forward will be made.
3.2.5 Examples from simulation

In figure 3.6 (a) the AVP signals from the turning motor are showed during cornering. In figure 3.6 (b) the JAVP signal is showed from the cantilever motor. In figure 3.7 (a) the three position paths are showed. At $t = 6 \text{ sec}$ the value *backward* turns one. The current trajectory is finished and at $t = 8.5 \text{ sec}$ the Halobject will start walking backwards.

In figure 3.7 (b) again the three position paths are showed. Now at $t = 10 \text{ sec}$ an emergency stop is received. The current step is finished first and at $t = 12.85 \text{ sec}$ the Halobject has stopped. The offset of the turning legs from the homing position is resetted at $t = 15 \text{ sec}$ when *reset offset* turns true and the system is ready to walk again.

![Figure 3.6: (a) Turning motor (b) Cantilever motor](image)

![Figure 3.7: (a) Transition from walking forwards to backwards (b) emergency stop succeeded by reset offset](image)
Chapter 4

Trajectory Planning for the Feelers Mechanism

This chapter will discuss the motion controller as designed for the feelers mechanism on the Halobject. First of all a short overview will be given of the system which has to be controlled. In section 4.2 the trajectory planning for the feeler mechanism will be described. Bla Bla.

4.1 Introduction

The Halobject is equipped with two feelers which have to react on sounds from its surrounding. Therefore the Halobject is equipped with three microphone’s which are mounted in the mantle of the Halobject and are equally divided over the contour (with 120 degrees between them). Sound localization by means of a data-processing algorithm is possible. There interaction of the Halobject with its surrounding is desired, the central controlling software (i.e. the character) can act on this input in several ways. For example the Halobject can decide to be curious and moves towards the sound source. Concerning the feeler mechanism the Halobject can decide to scan the surrounding with its feelers or actually point to a sound source.

Basically the feelers has to fulfill the following functions:

- The feelers are in scan mode and have to make a rotating movement to make the intension that the Halobject is scanning his surroundings;
- The feelers are in pointing mode and are pointing to a certain position in its surrounding;
- The feelers are in sleep mode. No action occurs.

During a previous internship, a two DOF mechanical system has been designed which can fulfill these demands. This system makes use of two electric motors that are perpendicularly positioned so a rotational movement can be achieved. The maximal allowable stroke from the midpoint is 18 degrees. In figure 4.1 and 4.2 the Halobject with the feelers and the feeler mechanism design is showed.

4.2 Trajectory planning

Elementarily, the trajectory planning software has to translate the demands of the character into the right reference paths for the two single axis motors so a desired movement is established (or no movement in case of the sleep mode). The software in question is written in C. This c-file is integrated
in MATLAB Simulink as an S-function. S-functions (system-functions) are a powerful mechanism for extending the capabilities of Simulink. This S-function has the following in- and outputs (see figure 4.3):

- **Inputs**
  1. **Scan mode [1/0]**
     If scan mode is true, the scan movement will start. The scan movement will always start from position zero \((x, y) = (0, 0)\). If scan mode is false, the scan movements stops and feelers return to zero \((x, y) = (0, 0)\). The scan mode uses second order trajectory planning with the following three parameters (input 2, 3 and 4);
2. **Deflection** \([\text{degrees}]\)
   This parameter is used in scan mode and represents the deflection in degrees of the feelers and is limited to 18 degrees due to the earlier stated mechanical restrictions. This signal is converted to radians and compensated for transmission of the worm - wormwheel set \((1 : 25)\).

3. **Bound on acceleration** \([\text{rad}/s^2]\)
   Bound on acceleration is used for second order trajectory planning.

4. **Bound on velocity** \([\text{rad}/s]\)
   Bound on velocity is used for second order trajectory planning.

5. **Pointing mode** \([1/0]\)
   If pointing mode is true, the pointing move will start. The three possible states from where transition to pointing mode can take place:
   
   (a) **Sleeping mode.** Feelers are in zero position \((x, y) = (0, 0)\).
   
   (b) **Scan mode.** \(\sqrt{x^2 + y^2} = \text{deflection}\)
   
   (c) **Pointing mode.** Feelers are pointing to desired pointing angle. \((x, y) = (x_{\text{des}}, y_{\text{des}})\) and are waiting for new input.

   If pointing mode turns false during pointing mode, the pointing will first finish the pointing move (if it was not ready) and will return to position zero \((x, y) = (0, 0)\). Pointing mode will always overrule scan mode, i.e. when pointing mode is true the software doesn’t look for the value of scan mode. This means when pointing mode returns to zero, and scan mode is still true, the feelers will move from pointing position, via zero, to the rotating scan move. All the used trajectories are second order.

6. **Pointing angle** \([\text{deg}]\)
   Pointing angle is required in pointing mode and is translated in a corresponding value for the x- and y-axis.

- **Outputs**
  1. Second order Trajectory x-axis (position \([\text{rad}]\), velocity \([\text{rad}/s]\) and acceleration \([\text{rad}/s^2]\))
  2. Second order Trajectory y-axis (position \([\text{rad}]\), velocity \([\text{rad}/s]\) and acceleration \([\text{rad}/s^2]\))

   The above mentioned zero position of the feeler mechanism \((x, y) = (0, 0)\) is determined by a homing procedure during the startup of the Halobject. So \((x, y) = (0, 0)\) represents the exact center of the feeler position.

   In **figure 4.3**, the global structure of the software can be seen with the IO’s as described above. The software is divided in two different parts, the trajectory planning and the trajectory output.

   ![Figure 4.3: IO of the trajectory planning software](image)

Both parts will be discussed in resp. subsections **4.2.1** and **4.2.2**
4.2.1 Second order trajectory planning

The c-file FTP2_c calculates a second order trajectory planning which will result in a scanning movement of the feeler. By generating a sinus reference for both x- and y-axis, but with 90 degrees delay between them, the feeler will describe a circle with a radius equal to the amplitude of the sinus. The disadvantage of applying a sine as reference signal is the startup of the movement. Taking the derivative of the sine-signal (position) will give a cosine signal for the velocity which will introduce a step at $t_0$. To solve this problem the second order trajectory planning as described in chapter two is used. First a trajectory is calculated from zero to the desired amplitude with the given constraints for $\ddot{a}$ and $\dot{v}$. The switching times for the acceleration profile are saved in vector $at_{init}$ with width four. The corresponding acceleration is equal to the value in scalar $aa$.

These variables are the same for the x- and y-axis.

$$at_{init} = [t_0 \ t_1 \ t_2 \ t_3] \ [sec]$$ (4.1)

$$aa = \ddot{a} \ [rad/sec^2]$$ (4.2)

Subsequently, a reference signal for the scan mode is calculated. This signal has to start where the previous trajectory stopped, namely at the desired amplitude. From there a second order approximation of a sine has to start. The calculated switching times needed for the acceleration profile are saved in vector $at$. The corresponding acceleration is the same as for the initialization, namely $aa$.

$$at = [t_0 \ t_1 \ t_2 \ t_3 \ t_4 \ t_5] \ [sec]$$ (4.3)

With these trajectories a smooth transition from zero into the scan mode is guaranteed. To keep the computer’s calculation time as small as possible, these calculation are only performed once when one of the three inputs is changed.

4.2.2 Trajectory output

The c-file FTP2_c uses the calculated acceleration profile information ($at_{init}$, $at$, $aa$) to output the acceleration reference signal. This signal is integrated once and twice to achieve respectively the velocity and position signal for the x- and y-axis. It uses two counters (for the x- and y-axis) triggered each sample time (1 msec) which are resetted at the end of each acceleration profile. By using counters, the software has time information needed to output the trajectory signals, but independently from the absolute running time.

As depicted in figure 4.3, the trajectory output software has to deal with three more inputs like 'scan mode [1/0]', 'pointing mode [1/0]' and 'pointing angle [deg]'. Therefore several states have been defined. In addition to the trivial states like scan mode and pointing mode, some extra states have been introduced to ensure that the desired behavior is emulated. The second order trajectory planning will guarantee a smooth movement of the feeler.

In figure 4.4 the statechart diagram is depicted which shows the sequence of states the software goes through. Transitions (depicted as arrows between states) indicate that, in response to an event, an object will go from one state to another and perform an action. The corresponding text-string contains the event that triggers a transition. When in a given state, an object waits for an event to go to a different state. Below, the different states will be discussed. More detailed information can be found in appendix A.

Initial state.

To determine the neutral position of the feelers, a homing procedure is performed during the startup of the Halobject. When this homing procedure is done (and the feelers are in homing position $(x, y) = (0, 0)$), transition to next state takes place (‘State: Waiting For Start’).
State: Waiting For Start.

The feelers are in homing position and are waiting for start, i.e. waiting for the booleans scan mode or pointing mode to turn true.

State: Scan Mode.

Transition from ‘State: Waiting For Start’ when start scan == 1. This state contains five sub-states:

- **State: Initialization**
  This state realizes the transition from the x- and y-motor from zero to the desired deflection value. When entering this state, first initialization of the x-axis will start using the acceleration switching times from vector at_init. When the trajectory is finished, and the x-axis reached the desired deflection value, the flag initxdone is set and transition to ‘State: scan x-axis’ takes place. The time to start the initialization of the y-axis is important to achieve a correct scanning movement. During the scan mode the criterium \( \sqrt{x^2 + y^2} = \text{deflection} \) must be satisfied. This means that the feeler is moving over a circle with a certain velocity \( \omega \). For the initialization state it means that the y-axis has to start at a certain time, such that it reaches its maximum deflection when the x-axis passes the zero (the x-axis is already in ‘State: scan x-axis’). From this point the earlier stated circle-criterium is satisfied, the flag initxdone is set and transition from the y-axis to ‘State: scan y-axis’ occurs.

- **State: Scan x-axis**
  This state realizes the scan movement of the x-axis after the flag initxdone is set. As explained in section 4.2.1 it uses the switching times from vector at to output the acceleration profile as function of counter x. By integration the velocity and position signal are obtained.

- **State: Scan y-axis**
  Same as State: Scan x-axis, but now for the y-axis (with initxdone and counter y).

- **State: Stop scan mode A x-axis**
  During scan mode, the status of boolean start scan is only checked when the velocity (of one axis) equals zero. If the start scan turns zero, the scan mode has to stop and the feelers must return to zero. This state will take care for the transition from scan mode to the homing position. A trajectory is calculated to let the x-axis return to zero.

- **State: Stop scan mode A y-axis**
  Same as for the x-axis. When both x and y-axis are returned to zero, transition to State: Waiting For Start takes place. In figures 4.5 and 4.6 the startup of the scan mode, the scan mode itself and stop scan mode is showed.

- **State: Stop scan mode B**
  During scan mode, the status of boolean pointing mode is only checked when the velocity (from x- or y-axis) equals zero. If pointing mode turns one, transition from scan to pointing mode has to occur (so the feelers are pointing to a certain angle). This state realizes the first part of it and aims to let the feeler stop on the scanning circle. From there transition to State: Start Pointing Mode B takes place which realizes the second movement to the desired pointing position. So the first part of the pointing mode occurs in this state. In figure 4.7 the transition from scan mode to pointing mode is depicted. At \( t = 8 \text{ sec} \) pointing mode turns true. At \( t = 9.4 \text{ sec} \), when the velocity of the first axis (y-axis) becomes zero, the value for pointing mode is checked. At this point transition to State: Stop scan mode B occurs. In this case the transition to this state first takes place for the y-axis because this axis reaches first the zero-velocity point. The axis which reaches State: Stop scan mode B first, will move to zero. The other will stay in scan mode until it also reaches the zero-velocity point. When the first axis, here y, reaches position zero, transition to the next state, State: Start Pointing Mode B occurs. When the second axis (x-axis) reaches its maximum amplitude (and so zero velocity) the State: Stop scan mode B is finished, and also transition to the next state takes place.
The reason for first returning to zero for the axis which enters first this state, is to ensure that no trajectory is calculated which will result in a movement outside the desired scanning radius (so $\sqrt{x^2 + y^2} = \text{deflection}$ is not exceeded).

State: Start Pointing Mode B.
This state succeeds the previous state and realizes the movement from a zero-velocity point (axis at zero or at maximum amplitude) to the desired endpoint with respect to the desired pointing angle. This desired pointing angle is converted to corresponding values for the x- and y-axis. In case of the example (figure 4.7) first the x-axis reaches this state and a trajectory is calculated to the desired end point. Same procedure is repeated when the y-axis enters this state. This state is finished when both axis have finished their trajectory and the feeler has reached his desired pointing position. Finally, this state transist to State: Pointing Mode.
This state can also be reached from State: Waiting For Start when pointing mode turns one and a movement from the homing position $(x, y) = (0, 0)$ to the desired pointing position has to be established.

State: Pointing Mode.
Feeler is in pointing position and is waiting for a new pointing angle or waiting for pointing mode to turn false (and pointing mode has to finish).

State: Start Pointing Mode C.
In case a new pointing angle is received, State: Start Pointing Mode C calculates new trajectories for x- and y-axis so movement to the new pointing position is realized. When both trajectories have finished transition back to State: Pointing Mode occurs. Additional software was needed to ensure that $\sqrt{x^2 + y^2} = \text{deflection}$ is not exceeded, i.e. the calculated trajectories will never result in a movement of the feeler outside the scanning radius. Basically it determines the quadrant of the scanning-circle for the current position and with respect to the new pointing position it calculates the trajectories and determines which axis to start first. The second axis will start a certain adjustable period later. Figures 4.7 and 4.8 show the above described states.

State: Stop Pointing Mode.
Transition to this state can only occur from State: Pointing Mode when pointing mode turns false. Pointing mode has to end and this state realizes the movement from the current pointing position to homing position $(x, y) = (0, 0)$. When this position is reached transition to State: Waiting For Start occurs.
Figure 4.4: States chart
Figure 4.5: xyp profiles during scan mode with start at $t = 0$ and stop at $t = 10$ sec.

Figure 4.6: position of follower during scan mode with start from midpoint and return to midpoint after stop scan.
Pointing mode = 1 & pointing angle = 35 deg
new pointing angle 195 deg
Pointing mode = 1 & pointing angle = 35 deg
new pointing angle 195 deg

Figure 4.7: avp profiles during scan mode with transition pointing mode (pointing mode = 1 @ t = 8 sec).

Figure 4.8: position of feeder during scan mode with start from midpoint and return to midpoint after stop scan.
Chapter 5

Conclusion and recommendations

The realized software was first tested in Matlab Simulink. Simulations turned out that the software was capable of generating the correct trajectory with respect to the inputs. Subsequently the trajectory planning software was integrated with a controller and implemented on the PC104 system of the Halobject. Implementation-tests turned out that the software functioned correctly. Summarizing, the developed trajectory planning for the Halobject guarantees a safe use of the Halobject concerning the operation of the Halobject. Furthermore smooth movement of the walk and feeler mechanisms is guaranteed.

In order to make the scanning movement of the feeler mechanism look more naturally, a random factor could be added which affects the rotational speed and/or the amplitude of the scanning move.
Bibliography


Appendix A

States of the trajectory planning feelers

A.1 State: Scan Mode and State: Stop Scan Mode A
State: Scan Mode & State: Stop Scan Mode A

Scan mode:
2nd order trajectory

Stop Scan Mode A
Transition from Scan mode to zero
A.2 State: Stop Scan Mode B

State: Stop Scan Mode B

Transition from Scan Mode to 4 possible stop positions dependent on current scan position.

From State: Scan Mode

CASE 1:  
- startpointing = 1
- startpointing = 0
CASE 2:
- startpointing = 1
- startpointing = 0

Determine start order

CASE 1
CASE 2

CASE 1
(x,y) = (max, 0)
CASE 2
(x,y) = (min, 0)

[ startpointing = 1 ]
[ startpointing = 1 ]

State: Start Pointing x-axis

[ x = 0,0 ]
[ y = 0,0 ]

int pointings x done
int pointings y done

To State: Start Pointing Mode B

CASE 1
(x,y) = (0, *)
CASE 2
(x,y) = (*, 0)

[ startpointing = 1 ]
[ startpointing = 1 ]
A.3 State: Start Pointing Mode B and State: Pointing Mode and State: Start Pointing Mode C
A.4 State: Stop Pointing Mode

State: Stop Pointing Mode

Transition from pointing position to zero

From State: Pointing Mode

\[
\text{startprinting} \leq 1
\]

State transition to zero x-axis

\[
(x \leq 0.0)
\]

Pointing x DONE

To State: Waiting for Start

\[
(x, y) = (0.0, 0.0)
\]

\[
\text{startprinting} \leq 0
\]

\[
\text{pointingsystem} \leq 1
\]

Pointing y DONE

\[
(x, y) = (0.0, 0.0)
\]

\[
\text{startprinting} \leq 0
\]
A.5 State: Start Pointing Mode B

State: Start Pointing Mode B

Situation in case of transition from State: Waiting for Start.

Transition from zero to pointing position

From State: Waiting for Start

[pointingmode == 1]

Set init values

[startpointing2 == 1]

See State: Start Pointing Mode B, Situation 2

Determine startpointing2 & startpointing2

[startpointing2 == 1]

State: Start Pointing y-axis Part 2

[x == y == 0]

Function Determine startpointing2 & startpointing2 skipped because of Function "Set init values"

Both States: Start Pointing x and y will start at same time

Pointing x Done

(startpointing2 == 0)

Pointing y Done

[startpointingdone == 1]

[pointingdone == 1]

To State: Pointing Mode

(x,y) = (xdel, ydel)
Appendix B

C-source

B.1 Walking mechanism

B.1.1 Second order trajectory planning Turning motor

```c
/* Trajectory Planning Calculation
 * Walking Mechanism
 * Version 2.0
 * P.R. Bagelaeker april '05
 *
*/

#define S_FUNCTION_NAME TP2
#define S_FUNCTION_LEVEL 2
#define pi 3.1415927
#include "simstruc.h"
#include <math.h>

static void milInitializeSizes(Simstruc *S)
{
    ssSetSumSFcnsParams(S, 0);
    if (ssGetSumSFcnsParams(S) != ssGetSFcnsParamCount(S))
    {
        return;
    }

    if (!ssGetNumInputPorts(S, 5)) return;

    ssSetInputPortWidth(S, 0, 1);  /*x*/
    ssSetInputPortWidth(S, 1, 1);  /*Ybound*/
    ssSetInputPortWidth(S, 2, 1);  /*alpha_hk*/
    ssSetInputPortWidth(S, 3, 1);  /*rest*/
    ssSetInputPortDirFeedThrough(S, 0, 1);
    ssSetInputPortDirFeedThrough(S, 1, 1);
    ssSetInputPortDirFeedThrough(S, 2, 1);
    ssSetInputPortDirFeedThrough(S, 3, 1);
    ssSetInputPortDirFeedThrough(S, 4, 1);

    if (!ssGetNumOutputPorts(S, 3)) return;
    ssSetOutputPortWidth(S, 0, 9);
    ssSetOutputPortWidth(S, 1, 9);
    ssSetOutputPortWidth(S, 2, 9);
    ssSetNumWork(S, 5);  /*Number of Real Workfactors*/
    ssSetNumSampleTimes(S, 1);
    ssSetOptions(S, SSOLPTION_EXCEPTION_FREE_ONE |
                 SSOLPTION_EXPR_TWICE_EVALUATION);
}

static void milInitializeSampleTimes(Simstruc *S)
```

37
{ 
ssSetSampleTime(S, 0., INHERITED_SAMPLE_TIME);
ssSetOffsetTime(S, 0.0);
}

#define MLI_START
static void mlStart(BisStruct *s)
{
real T
  worka = ssGetWork(S);
  work[0]=0;  /\ xleft /
  work[1]=0;  /\ y /
  work[2]=0;  /\ a /
  work[3]=0;  /\ xright /
  work[4]=0;  /\ t rest /
}

static void mlOutputs(BisStruct *s, int T tid)
{
int T
  i,delta_t,straight;
  real T
    x,xz,x1,xleft,xright,yb,ab,a,alfa,t_1r,t_2r,tj1,t_1,t_2;
    st[0],as[0],as[1];
  InputemPtrType isi = ssGetInputForthemSignalPtrs(S, 0);  /\ xin */
  InputemPtrType isi2 = ssGetInputForthemSignalPtrs(S, 1);  /\ xbound */
  InputemPtrType isi3 = ssGetInputForthemSignalPtrs(S, 2);  /\ alfa hal */
  InputemPtrType isi4 = ssGetInputForthemSignalPtrs(S, 3);  /\ trest */
  real T
    yi = ssGetOutputsForthemSignal(S, 0);
    y2 = ssGetOutputsForthemSignal(S, 1);
    y3 = ssGetOutputsForthemSignal(S, 2);
    workb = ssGetWork(S);

  time=1000;
  pi=;
    x1=fabs(isi[0]);  /\ abs values of input */
    y1=fabs(isi2[1]);
    a=fabs(isi3[2]);
    alfa=fabs(isi4[3])/180;
    xrest=fabs(y1)/180;
    trest=fabs(alfa)*time;
    x=x2=asin(sin(0.5*x1)-0.5*tan(fabs(alfa)));  /\ smallest forward stroke when cornering */
    if (alfa >= 0) {
      xleft=x1;
      xright=xz;
    } else if (alfa < 0) {
      xright=x1;
      xrest=xz;
    }
    if (x1 = x2){  /\ --- Determine cornering yes/no */
      stright=1;
    } else (straight=0);
  }  /* Calculate trajectory with largest tend (= largest x) */
  t_1r=sqrt(x1/ab);  /\ Const Acceleration Time */
  if (y1>ab+t_1r)  /\ y bound violated */
    { 
      t_1r=ab/t_1r;
      t_2r = (t_1r-t_1r)*t/ab;  /\ Const Velocity Time */
      ab = x1/(t_1r+t_1r+t_2r);  /\ New bound on acceleration */
      t_1=floor(t_1r+1000.0.5);  /\ rounding */
      t_2=floor(t_2r+1000.0.5);  /\ rounding */
  }  /* Calculate Acc. Profile */
  t_1=0.;  /\ end time of cycle*/
  tj=floor(t_1+0.5);  /\ desired end time with respect to rounding error of zero*/
delta_t=fabs(t6-t6des);  

/*start routine to round t6 to t6des to avoid roundings problems*/
if (delta_t<0) {
    if (t6 < t6des) { 
        t6=t6+delta_t/4;  /* enlarge const. acc. time */
    } else if (t6 > t6des) { 
        t6=t6-4*delta_t/4;  /* shorten const. acc. time */
    }
}

t6+=t1*2*st_2;

if (delta_t<0) {
    if (t6 < t6des) { 
        t6+=2*delta_t/2;  /* enlarge const. acc. time */
    } else if (t6 > t6des) { 
        t6-=2*delta_t/2;  /* shorten const. acc. time */
    }
}
t1+=st_1;
t2+=st_2;
t3+=st_1;
t4+=st_1;
t5+=st_1;
t6+=st_1;
t7+=st_1;
t8+=st_1;


/*calculate trajectory for smallest x (xleft or xright)*/

/*only during cornering*/
if (straight == 0) {
    t = t1/1000; 
    t2=t2/1000;
    x=x/(t_t*1+t_2);
} else if (straight != 1) {
    for (i=0; i<9; i++) { /* output*/
        y[i]=at[i];
        if (xleft > xright) { 
            y[i]=a[i]; 
            y[i]=a[i];
        } else if (xleft < xright) { 
            y[i]=a[i]; 
            y[i]=a[i];
        }
    }
}

/* end if (new data) loop*/

/* == Storing old input ----------------------------- */
work[0]=x[1];
work[1]=x[2];
work[2]=x[3];
work[3]=x[4];
work[4]=x[5];
work[5]=x[6];
work[6]=x[7];
work[7]=x[8];

}

static void m1Terminate(Simstruct *s)
{
}

#endif

#include "Hsimlink.c" /* MEX-file interface mechanism */
#define "g_scheme.h" /* Code generation registration function */
#endif

B.1.2 Third order trajectory planning Cantilever motor

/* Trajectory Planning Calculation for Cantilever Motor
* Version 2.0
* P.H. Nijelmaker april '05
*/

39
```c
#define S_FUNCTION_NAME TP3_C
#define S_FUNCTION_LEVEL 2
#define stime 1000  /* sample frequency*/

#include "sinstruc.h"
#include <math.h>

static void mil1InitializeSimes(SimStruct *s)
{
    ssSetNumSFcnParams(S, 0);
    if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S))
    {
        return;
    }
    if (!ssSetNumInputPorts(S, 3)) return;
    ssSetInputPortWidth(S, 0, 1);  /*stepheight*/
    ssSetInputPortWidth(S, 1, 9);  /*t*/
    ssSetInputPortWidth(S, 2, 1);  /*backwards?*/
    ssSetInputPortDirectFeedThrough(S, 0, 1);
    ssSetInputPortDirectFeedThrough(S, 1, 1);
    ssSetInputPortDirectFeedThrough(S, 2, 1);
    if (!ssSetNumOutputPorts(S, 2)) return;
    ssSetOutputPortWidth(S, 0, 31);
    ssSetOutputPortWidth(S, 1, 31);
    ssSetNumWork(S, 11);       /*number of real workfactor*/
    ssSetNumSampleSizes(S, 1);
    ssSetOptions(S, SS_OPTION_EXCEPTION_FREE_CODE | SS_OPT_IN_USE_TLC_WITH_ACCELERATOR);
}

static void mil1InitializeSampleTimes(SimStruct *s)
{
    ssSetSampleTime(S, 0, INHERITED_SAMPLE_TIME);
    ssSetOffsetTime(S, 0, 0.0);
}

#define MIL1_START
#define MIL1_END

static void mil1Start(SimStruct *s)
{
    int_T i;
    real_T *work = ssGetWork(S);
    for (i=0; i<11; i++) { work[i]=0; }
}

static void mil1Outputs(SimStruct *s, int_T tid)
{
    int_T i,back;
    real_T x_s,a_3,t_end,ta,t,j,tress;
    real_T x[t][31],y[31];
    Input1Ptr0Type in = ssGetInputPort0Sim1InputPtr(S,0);
    Input1Ptr0Type in1 = ssGetInputPort3Sim1InputPtr(S,1);
    Input1Ptr0Type in2 = ssGetInputPort3Sim1InputPtr(S,2);
    real_T *y1 = ssGetOutputPort0Sim1Output(S,0);
    real_T *y2 = ssGetOutputPort0Sim1Output(S,1);
    real_T *work = ssGetWork(S);
    /* == Getting Absolute value of Input ----------------------------- */
    if (*in1[0] != work[0] || *in2[0] != work[3] || *in3[0] != work[6])
    {
        x=fabs(*in1[0]); /* abs values of input*/
        t_end = in2[3]/stime; /* end time of first half of cycles -- at [3]*/
        ta=t_end/6; /* values for 2nd order output */
```
a=-(2*t+a*a);

s=3*a*a/6;             /* new max value for a with bound on jerk*/
tj=ta/6;               /* jerk [md/s^3 with tj=ta/6 and ta=(endo/12)/
tj=ta/6;               /* jerk [md/s^3 with tj=ta/6 and ta=(endo/12)/
trext=(i+3[4]*i+2[3])/stine;


j[8]=j[7];


j[16]=j[15];

j[21]=j[20];

j[22]=j[21]+t[22];

j[29]=j[28];

j[30]=j[29]+t[30];

if (*i+3[0] == 0) {  /* walking forward*/
bk=1;            
else if (*i+3[0] == 1) {  /* walking backward*/
bk=0;
}

for (i=0; i<=i++;)
{
  x[1]=floor(x[1]+10000/0.5);
  x[2]=max(jx[1];
}  
/* end if (new data?) loop*/

void m1Terminate(SimStruct ss)
{
}

#define MATLAB_MEX_FILE  /* Is this file being compiled as a MEX-file? */
#include "simlink.c"   /* MEX-file interface mechanism */
else
#include "cgs_nsw.m"  /* Code generation registration function */
}  

B.1.3 Trajectory output turning motor

/* Trajectory Output Calculation for turning motor  
* Version 1.4  
* P.H. Meijermeijer May '05  
*/
#define S_FUNCTION_NAME   m1u2
#define S_FUNCTION_LEVEL 2
#define stine 10000
#include "sinscruc.h"
#include <math.h>

static void m1lInitalize(SimStruct ss)
{
ssSetNumSFcnParams(S, 0);
if (ssGetNumFcsParams(S) != ssGetFcsParamCount(S))
{
    return;
}

if (!ssGetNumInputPorts(S, 6)) return;
ssSetInputPortWidth(S, 0, 9);
ssSetInputPortWidth(S, 1, 9);
ssSetInputPortWidth(S, 2, 1);
ssSetInputPortWidth(S, 3, 1);
ssSetInputPortWidth(S, 4, 1);
ssSetInputPortDirectFeedthrough(S, 0, 1);
ssSetInputPortDirectFeedthrough(S, 1, 1);
ssSetInputPortDirectFeedthrough(S, 2, 1);
ssSetInputPortDirectFeedthrough(S, 3, 1);
ssSetInputPortDirectFeedthrough(S, 4, 1);

if (!ssGetNumOutputPorts(S, 3)) return;
ssSetOutputPortWidth(S, 0, 1);  /* acc */
ssSetOutputPortWidth(S, 1, 1);  /* vel */
ssSetOutputPortWidth(S, 2, 1);  /* pos */

ssSetNumWork(0, 6);
ssSetNumWork(0, 29);
    /* IS = 3 = 0...8 = tot wid + in[2][9] = old[i][v][x[i][x][i][x][i][x] */

ssSetSampleTimes(S, 1);
ssSetOptions(S, SS_OPTION_EXCEPTION_FREE_CODE |
            SS_OPTION_USE_TLC_WITH_ACCELERATOR);
}

static void mllInitializeSampleTimes(SimStruct *s)
{
    ssSetSampleTime(S, 0, SS_RESAMPLED_SAMPLE_TIME);
    ssSetOffsetTime(S, 0, 0.0);
}

#define MLL_START

static void mllStart(SimStruct *s)
{
    int_T i;
    int_T workI = ssGetWork(s);
    real_T workA = ssGetWork(s);

    for (i=0; i<6; i++)
    {
        workI[i]=0;
        workA[i]=0;
    }
}

static void mllOutputs(SimStruct *s, int_T tid)
{
    int_T i,timem,tmand,tmdiv2,count,ready,start,estop,offset;
    real_T a,sold;

    InputParam_type in1 = ssGetInputParam(SigmaPtr(s,0));  /* In [0..9] */
    InputParam_type in2 = ssGetInputParam(SigmaPtr(s,1));  /* In [0..9] */
    InputParam_type in3 = ssGetInputParam(SigmaPtr(s,2));  /* In [0..9] */
    InputParam_type in4 = ssGetInputParam(SigmaPtr(s,3));  /* In [0..9] */
    InputParam_type in5 = ssGetInputParam(SigmaPtr(s,4));  /* In [0..9] */

    real_T y1 = ssGetOutputParam(SigmaPtr(s,0));  /* Acc */
    real_T y2 = ssGetOutputParam(SigmaPtr(s,1));  /* Vel */
    real_T y3 = ssGetOutputParam(SigmaPtr(s,2));  /* Pos */

    int_T workI = ssGetWork(S);
    /* work factors integer */
    real_T workA = ssGetWork(S);
    /* work factors real */

    count=workI[0];
    ready=workI[1];
    start=workI[2];
    estop=workI[3];
    offset=workI[4];  /* 1 if emergency stop activated at first half of cycle */
    sold=workA[18];
stime=1000;
tend=work[8+21];    /* end time cycles in counts */
tenddiv=work[3+21]; /* half of cycles in counts */
if (ready == 1){
    ready=0;
}
if (count == 0) {
    ready=1;
}
if (ready == 1 && offset == 0) { /* Getting new input once a cycle*/
    for (i=0; i<9; i++) /* Getting New Input Values */
    {
        work[i+2] = int[i]; /* int[i]*/
        work[i+4] = int[i]; /* int [i]*/
    }
    work1[2] = int[0]; /* getting value for start*/
    start=work[2];
}
if (ready == 1 || count == tenddiv) { /* Getting value for estop*/
    work[3] = int[0];
    estop=work[3];
    if (count == tenddiv && estop == 1) {
        offset=1;
        work[4]=1;
    }
}
if (start == 1 && estop == 0 && offset == 0) /* Start walk*/
{
    /* Calculating Acc as functions of counts --------------------*/
    if (count < work[1+21])
    {
        a=work[9]; /* ab*/
    }
    else if (count > work[1+21] && count < work[2+21])
    {
        a=work[10];
    }
    else if (count > work[2+21] && count < work[3+21])
    {
        a=work[11];
    }
    else if (count > work[3+21] && count < work[4+21])
    {
        a=work[12];
    }
    else if (count > work[4+21] && count < work[5+21])
    {
        a=work[13];
    }
    else if (count > work[5+21] && count < work[6+21])
    {
        a=work[14];
    }
    else if (count > work[6+21] && count < work[7+21])
    {
        a=work[15];
    }
    else if (count > work[7+21] && count < work[8+21])
    {
        a=work[16];
    }
    else if (count > work[8+21] && count <= work[9+21])
    {
        a=0;
    }
}
/* ---------- integrators ------------------------------------------*/
work[19]=work[19]+old/stime;
/* ---------- reset Integrators ------------------------------------*/
if (count == 0 || count == tenddiv) /* Reset integrator velocity*/
    {work[16]=-0;}
if (count == 0) /* Reset integrator position*/
    {work[20]=-0;}
/* ---------- generate outputs ------------------------------------*/
y1=0;
y2=work[19];
y3=work[20];
work[18]=a;    /* storing old value of a*/
/* ---------- ophogen Counts --------------------------------------*/
if (count < tend) {
    {work(0)++;}
    {work(0)++;}
} /* end if start == 1*/
else if (start == 0 || estop == 1 && count == 0) { /* ESTOP received*/
    *y1=0;
    *y2=0;}
```c
+y3=0;

if (offset == 1) {
    if (*in[0] == 1) { /* request for reset */
        vors[1] = 1; /* reset */
    }
    if (vors[1] == 1) { /* if offset == 1 */
        if (count >= work[i+1] && count < work[i+2])
            a-work[i+3]; /* a = ah */
        else if (count >= work[i+2] && count < work[i+3])
            a-work[i+4]; /* a = 0 */
        else if (count >= work[i+3] && count <= work[i+4])
            a-work[i+5]; /* a = ah */
    } /* ---- Integrators --------------------------------- */
    work[19]=work[19]+s*time;

    /* ---- Reset Integrators ---------------------------------*/
    if (count == 0 || count == tend+2) /* Reset Integrator velocity */
        {work[19]=0;}
    if (count == 0) /* Reset Integrator position */
        {work[20]=0;}

    /* ---- Generate outputs -----------------------------------*/
    y1=a;
    y2=work[19];
    y3=work[20];
    work[18]=a; /* Storing old value of a */

    /* ---- Update Counts --------------------------------------*/
    if (count<=tend)
        for (i=0; i<5; i++)
            if (count=i-1)
                {work[i]=0; /* count = */
                 work[i+1]=0; /* offset = */
                 work[i+2]=0; /* reset = */
                 }
    } /* end of reset period */
}
}

static void m1Termate(SimStruct *d)
{
}

#endif MATLAB_MEX_FILE
/* Is this file being compiled as a MEX-file? */
#include "simulink.c" /* MEX-file interface mechanism */
#define include "q_sfun.h" /* Code generation registration function */
#endif

B.1.4 Trajectory output cantilever motor

/* Trajectory Output Calculation
   * Version 1.7
   * P.N. Nagelaeker mei '06
   */

#define S_FUNCTION_NAME TDL_C
#define S_FUNCTION_LEVEL 2
#define time 1000

#include "simstruct.h"
#include <math.h>

static void m1tOutputssSetSizes(SimStruct *d)
{
    ssSetNumSFcnParams(d, 0);
    if (ssGetNumSFcnParams(d) != ssGetSFcnParamsCount(d))
    {
        return;
    }
}

44
if (isSetNumInputPorts(S, 0) return;

ssNetInputPortWidth(S, 0, 31);
     // #0..30  j
ssNetInputPortWidth(S, 1, 31);
     // #31..60  j
ssNetInputPortWidth(S, 2, 1);
     // #62  start
ssNetInputPortWidth(S, 3, 1);
     // #63  stop
ssNetInputPortWidth(S, 4, 1);
     // #64  reset offset?

ssNetInputPortDirectFeedThrough(S, 0, 1);
ssNetInputPortDirectFeedThrough(S, 1, 1);
ssNetInputPortDirectFeedThrough(S, 2, 1);
ssNetInputPortDirectFeedThrough(S, 3, 1);
ssNetInputPortDirectFeedThrough(S, 4, 1);

if (isSetNumOutputPorts(S, 4)) return;
ssNetOutputPortWidth(S, 0, 1);
ssNetOutputPortWidth(S, 1, 1);
ssNetOutputPortWidth(S, 2, 1);
ssNetOutputPortWidth(S, 3, 1);

ssNetNumWork(S, 6);
ssNetNumWork(S, 67);

ssNetSampleTimes(S, 1);

ssNetOptionS(S, 0, OPTION_EXCEPTION_FREE_CODE | OPTION_USE_TLC_WITH_INTEGRATOR);
}

static void mblInitializeSampleTimes(SimStruct *S)
{
    ssNetSampleTime(S, 0, INHERITED_SAMPLE_TIME);
    ssNetOffsetTime(S, 0, 0.0);
}

#define MII_START
static void mIIStart(SimStruct *S)
{
    int T
    int T
    int T
    int T

    work[0]=0;
    /* count*/
    work[1]=0;
    /* ready bit*/
    work[2]=0;
    /* start bit*/
    work[3]=0;
    /* emergency stop bit*/
    work[4]=0;
    /* offset*/
    work[5]=0;
    /* reset*/

    for (i=0; i<67; i++) {
        work[i]=0;
    }
}

static void mIIOutputs(SimStruct *S, int T tid)
{
    int T
    int T
    int T
    int T
    int T
    int T
    int T
    int T
    int T
    int T

    count=work[0];
    ready=work[1];
    start=work[2];
    stop=work[3];
    offset=work[4];
    jold=work[60];
    sTime=1000;
}

45
temp=work[30];  /* end time cyclus in counts */

temdir4=work[7];
temdir2=work[14];
temdir4=work[22];

if (ready ==1)  /*Reset of Ready-bit*/
    {ready=0;}

if (count ==0)  /* Set Ready-bit*/
    {ready=1;}

if (ready ==1) {  /*Getting new input once a cycle*/
    work[2]=in3[0];
    start(work[2];
    for (i=0; i<31; i++)  /* Getting New Input Values */
        {
            work[i]=in[i];  /* = j[t][i]*/
            work[i+31]=in2[i];  /* = j[t][i]*/
        }
    }

if (ready ==1 && count == temdir2) {  /*Checking for ENTLP*/
    work[13]=in4[0];
    estop=work[13];
    if (estop ==1 && count == temdir2) {  /*option 1*/
        offset=1;
        work[13]=offset;}
    }

/* ------ Calculating Acc as functions of counts ------------------------ */
if ( count < work[1] )  /* j=0*/
    { 
        j=work[0+31];
        work[03]=work[03]+old/stime; 
    }
else if ( count < work[2] && count < (work[32]) )  /* j= 0*/
    { 
        j=work[1+31];
        work[03]=work[03]+old/stime; 
    }
else if ( count < work[3] )  /*store a for calc V*/
    { 
        j=work[2+31];
        work[03]=work[03]+old/stime; 
    }
else if ( count < work[4] )  /* j= 0 */
    { 
        j=work[3+31];
        work[03]=0; 
    }
    { 
        j=work[4+31];
        work[03]=work[03]+old/stime; 
    }
    { 
        j=work[5+31];
        work[03]=work[03]+old/stime; 
    }
else if ( count < work[7] && count < work[8] )  /* j=0*/
    { 
        j=work[6+31];
        work[03]=work[03]+old/stime; 
    }
else if ( count < work[8] && count < work[9] )  /* j=0*/
    { 
        j=work[7+31];
        work[03]=work[03]+old/stime; 
    }
else if ( count < work[9] && count < work[10] )  /* j=0*/
    { 
        j=work[8+31];
        work[03]=work[03]+old/stime; 
    }
    { 
        j=work[9+31];
        work[03]=work[03]+old/stime; 
    }
    { 
        j=work[1+31];
        work[03]=work[03]+old/stime; 
    }
else if ( count < work[12] && count < work[13] )  /* j=0*/
    { 
        j=work[2+31];
        work[03]=work[03]+old/stime; 
    }
else if ( count < work[13] && count < work[14] )  /* j=0*/
    { 
        j=work[3+31];
        work[03]=work[03]+old/stime; 
    }
    { 
        j=work[4+31];
        work[03]=work[03]+old/stime; 
    }
}
    { j-work[16-31];
      work[63]-work[63]+old/time;
    } /* j- j*/
else if ( count >= work[16] && count < work[17] )
    { j-work[16-31];
      work[63]-work[63]+old/time;
    } /* j- 0*/
else if ( count >= work[17] && count < work[18] )
    { j-work[17-31];
      work[63]-work[63]+old/time;
    } /* j- j*/
else if ( count >= work[18] && count < work[19] )
    { j-work[18-31];
      work[63]-work[63]+old/time;
    } /* j- 0*/
    { j-work[19-31];
      work[63]-work[63]+old/time;
    } /* j- j*/
    { j-work[20-31];
      work[63]-work[63]+old/time;
    } /* j- 0*/
else if ( count >= work[21] && count < work[22] )
    { j-work[21-31];
      work[63]-work[63]+old/time;
    } /* j- j*/
else if ( count >= work[22] && count < work[23] )
    { j-work[22-31];
      work[63]-work[63]+old/time;
    } /* j- j*/
else if ( count >= work[23] && count < work[24] )
    { j-work[23-31];
      work[63]-work[63]+old/time;
    } /* j- 0*/
    { j-work[24-31];
      work[63]-work[63]+old/time;
    } /* j- j*/
else if ( count >= work[25] && count < work[26] )
    { j-work[25-31];
      work[63]-work[63]+old/time;
    } /* j- 0*/
else if ( count >= work[26] && count < work[27] )
    { j-work[26-31];
      work[63]-work[63]+old/time;
    } /* j- j*/
else if ( count >= work[27] && count < work[28] )
    { j-work[27-31];
      work[63]-work[63]+old/time;
    } /* j- 0*/
else if ( count >= work[28] && count < work[29] )
    { j-work[28-31];
      work[63]-work[63]+old/time;
    } /* j- j*/
else if ( count >= work[29] && count < work[30]+1 )
    { /* ==test*/
      j-work[29-31];
      work[63]-work[63]+old/time;
    } /* j- 0*/

//* ------- generate outputs ---------------------------------------- */
  y+j;
  y+j=work[63];

//* ------- integrators --------------------------------------------- */
  work[64]=work[64]+y+j/time; /* velocity*/
  y+j=work[64];
  work[65]=work[65]+y+j/time; /* position*/
  y+j=work[65];

if ( count == 0 ) || count == tem+di+4 | count == tem+di+2 | count == tem+di+4 )
  { work[64]=0; y+j=work[64]; /* Reset Integrator velocity*/
    }
if ( count == 0 ) /* Reset Integrator position*/
  { work[63]=0; y+j=work[63];
    work[65]=0; y+j=work[65];
    }

//* ------- Opposition Count ---------------------------------------- */
  work[62]=j; /* storing old value*/

if ( count < tend )
  { (work[1][j]);
if ( count < tend )
  { work[0][j]=0; }
else if ( start == 0 | stop == 1 | offset == 0 ) { /* no offset of keermotor...Waiting for start*/ /option 2*/

47
/*y1<0;
  y2<0;  work[63]=-0.0;
  y3<0;  work[64]=-0.0;
  y4<0;  work[65]=-0.0;*/
else if (offset == i) {  /*offset of kmotor...Waiting for resetting offset*/
  if (+int[0] == i) {  /* request for reset ?? */
    work[1][i] = 1;  /* reset -1*/
  } else if (work[1][i] == 0) {  /* if reset == 0*/
    y1<0;
    y2<0;  work[63]=0.0;
    y3<0;  work[64]=0.0;
    y4<0;  work[65]=0.0;
    jold=0;  work[66]=0;
  }
else if (work[1][i] == 1) {  /* Resetting offset*/
  }
/* count > work[14] & count < work[16] */
  {  j=work[14];
    work[63]=work[63]+jold/stime;
  } else if (count > work[15] & count < work[16] )
  {  j=work[15];
    work[63]=work[63]+jold/stime;
  } else if (count > work[16] & count < work[17] )
  {  j=work[16];
    work[63]=work[63]+jold/stime;
  } else if (count > work[17] & count < work[18] )
  {  j=work[17];
    work[63]=work[63]+jold/stime;
  } else if (count > work[18] & count < work[19] )
  {  j=work[18];
    work[63]=work[63]+jold/stime;
  } else if (count > work[19] & count < work[20] )
  {  j=work[19];
    work[63]=work[63]+jold/stime;
  {  j=work[20];
    work[63]=work[63]+jold/stime;
  } else if (count > work[21] & count < work[22] )
  {  j=work[21];
    work[63]=work[63]+jold/stime;
  } else if (count > work[22] & count < work[23] )
  {  j=work[22];
    work[63]=work[63]+jold/stime;
  } else if (count > work[23] & count < work[24] )
  {  j=work[23];
    work[63]=work[63]+jold/stime;
  {  j=work[24];
    work[63]=work[63]+jold/stime;
  } else if (count > work[25] & count < work[26] )
  {  j=work[25];
    work[63]=work[63]+jold/stime;
  } else if (count > work[26] & count < work[27] )
  {  j=work[26];
    work[63]=work[63]+jold/stime;
  } else if (count > work[27] & count < work[28] )
  {  j=work[27];
    work[63]=work[63]+jold/stime;
  } else if (count > work[28] & count < work[29] )
  {  j=work[28];
    work[63]=work[63]+jold/stime;
  } else if (count > work[29] & count < work[30]+1 )
  {  y1=-trest;
  }

/* ---- generes outputs -------------------------------------------------- */
  y1=-j;
  y2=work[63];

/* ---- integrators ----------------------------------------------------- */
  work[64]=work[64]+y2/stime;  /* velocity*/
  y3=work[64];
  work[65]=work[65]+y3/stime;  /* position*/
  y4=work[65];

if (count == 4 || count == tendiv1 || count == tendiv2 || count == tendiv4)
  {  work[64]=0;  y3=work[64];}  /*reset Integrator velocity*/
if (count == 0 )  /*reset Integrator position*/
  {  work[63]=0;  y2=work[63];
      work[65]=0;  y4=work[65];}

48
/* ========= oplogos Counts ============== */
work[62]-j;

    if ( count<end )
    { (work[0])=;}
    if ( count<end )
    { work[0]=0;
      work[1]=0;  /* offset =0 */
      work[2]=0;  /* reset =0 */
    } /* end of resetting offset*/
}

} /*end of output function*/

static void mdlTerminate(SimStruct *s)
{

#endif HML тысяч ^= /* Is this file being compiled as a MEX-file? */
#include "simlink.c" /* MEX-file interface mechanism */
#else
#include "eg_sim.h" /* Code generation registration function */
#endif
B.2 Feeler mechanism

B.2.1 Second order trajectory planning Feelers

/ * Trajectory Planning Calculation Feelers
  * Version 2.0
  * P.M. Ngelemanek nei '05
  *
  */

#define S_FUNCTION_NAME FTP2
#define S_FUNCTION_LEVEL 2
#define PI 3.14159265358979
#define stime 1000
#include "simstruc.h"
#include <math.h>

static void mlInitalizeSimTimes(SimStruct *s)
{
  ssSetNumSFcnParams(s, 0);
  if (ssGetNumSFcnParams(s) != ssGetSFcnParamsCount(s))
  {
    return;
  }
  if (ssGetNumInputPorts(s, 3)) return;

  ssSetInputPortWidth(s, 0, 1); /* desired deflection (max value = 18 deg) */
  ssSetInputPortWidth(s, 1, 1); /* vbound */
  ssSetInputPortWidth(s, 2, 1); /* absbound */
  ssSetInputPortDirectFeedthrough(s, 0, 1);
  ssSetInputPortDirectFeedthrough(s, 1, 1);
  ssSetInputPortDirectFeedthrough(s, 2, 1);

  if (ssGetNumOutputPorts(s, 3)) return;

  ssSetOutputPortWidth(s, 0, 6); /*et[6]*
  ssSetOutputPortWidth(s, 1, 6); /*et[6]*
  ssSetOutputPortWidth(s, 2, 4); /*et_in[4]*/

  ssSetNumWork(s, 3);
  /*Set number of real work factors*/
  ssSetNumSampleTimes(s, 1);

  ssSetOptions(s, SS_OPTION_EXCEPTION_FREE_CODE |
               SS_OPTION_DISABLE_WARNING_ALWAYS_WITH_NOCODECHECKER);
}

static void mlInitalizeSampleTimes(SimStruct *s)
{
  ssSetSampleTime(s, 0, INHERITED_SAMPLE_TIME);
  ssSetIfsetTime(s, 0, 0.0);
}

#define ML_START
static void mlStart(SimStruct *s)
{
  real_T
  /*work = ssSetWork(s);*/
  work[0]=0; /* desired deflection (max value = 18 deg) -- *in[0] */
  work[1]=0; /* vbound -- *in[1] */
}

static void mlOutputs(SimStruct *s, int T tid)
{
  int_T
    i, tr_izl, tr;
  real_T
    amp, v, ab, tar, t, tr_isltr, t1, t2, t3, t4, t5;
  real_T
    at[6], ax[6], at_in[4];
  int ptrType
    if = ssGetInputPortRealSignalPtrs(s, 0);
  real_T
    y1 = ssGetInputPortRealSignal(s, 0); /*et[6]*
  real_T
    y2 = ssGetInputPortRealSignal(s, 1); /*et[6]*
  real_T
    y3 = ssGetInputPortRealSignal(s, 2); /*et_in[4]*
}
real_t *workk = ssGetWork( S );

{
    ampl=fabs(*in[0])+PI/256/4;  /* abs values of input and converted to radians and compensated for transmission*/
    yb=fabs(*in[1]);
    ab=fabs(*in[2]);

    /* --- Calculating t_1 (Constant Acc phase) and t_2 (Constant Vel phase) --- */
    tar=sqrt( ampl/ab );  /* Const Acceleration Time */
    if ( yb<ab*tar )  /* y bound violated */
    {
        tar=yb/ab;
        ab=rb/tar;
    }

    t=-y/yb;
    t2=t1*tv;
    t3=t2*t2+ta;
    tv=t3*tv;
    tb=t4*ta;

    as[0]=ab;  as[1]=0.0;  as[2]=ab;  as[3]=0.0;  as[4]=ab;  as[5]=0.0;
    at_flight[0]=0.0;  at_flight[1]=ta;  at_flight[2]=ta+tv_init;  at_flight[3]=ta+tv_init+ta;

    for ( i=0; i<6; i++ )
    {
        y[i]=at[i];
        y2[i]=as[i];
    }
    for ( i=0; i<4; i++ )
    {
        y3[i]=at_flight[i];
    }
}

} /* end if (new data?) loop */

/* Storing old input ----------------------------------------------- */
workk[0]=in[0];
workk[1]=in[1];
workk[2]=in[2];

}

static void mlTerminate( SimStruct *s )
{

#endif MATLAB_HEX_FILE  /* Is this file being compiled as a HEX-file? */
#include "simlink.c"  /* HEX-file interface mechanism */

else
#include "g_nsfun.h"  /* Code generation registration function */

#endif

B.2.2 Trajectory output Feelers

/* Trajectory Output Calculation
 * Version 2.1
 * P.M. Nagelmeaker mei '05
 * */

#define S_FUNCTION_SAME FTID
#define S_FUNCTION_LEVEL 2
#define PI 3.14159265358979
#define sample 1000  /* sample frequency */
#define ampl 7.4  /* desired ampl used in pointing node */
# define ab 6.0 /* Round on cc. used in pointing node */
# define vb /* Round on cc. used in pointing node */

#include "sinstruct.h"

#include <math.h>

static void mLInitilizesizes(SinStruct *S)
{
    ssSetNumSFcnsParams(S, 0);
    if ((ssGetNumSFcnsParams(S) != ssGetSFcnsParamsCount(S))
        return;
    }

    if ((ssGetNumInputPorts(S, 6)) return;
    ssSetInputPortWidth(S, 0, 6); /* t */
    ssSetInputPortWidth(S, 1, 6); /* w */
    ssSetInputPortWidth(S, 2, 4); /* t_init[4] */
    ssSetInputPortWidth(S, 3, 1); /* scan node */
    ssSetInputPortWidth(S, 4, 1); /* startpointing node */
    ssSetInputPortWidth(S, 5, 1); /* startpointing angle */
    ssSetInputPortWidth(S, 6, 1); /* ax */
    ssSetInputPortWidth(S, 7, 1); /* ay */
    ssSetInputPortWidth(S, 8, 1); /* px */
    ssSetInputPortWidth(S, 9, 1); /* py */
    ssSetInputPortWidth(S, 10, 1); /* py */

    ssSetWork(S, 22);
    ssSetSumSample[ines(S, 1));
    ssSetOption(S, SS_OPTION_EXCEPTION_FREE_CODE | SS_OPTION_USE_TLC_WITH_LOADER);  
}

static void mLInitilizesampleTimes(SinStruct *S)
{
    ssSetSampleTime(S, 0, INSERTED_SAMPLE_TIME);
    ssSetWiffsetTime(S, 0, 0.0);
}

#define HDL_START

static void mLStart(SinStruct *S)
{
    int7
    int7 = work = ssNetWork(S);
    real7
    real7 = work = ssNetWork(S);

    work[0]=0; /* count_x */
    work[1]=0; /* startpointing */
    work[2]=0; /* start x */
    work[3]=0; /* initdone */
    work[4]=0; /* at_init[0] */
    work[5]=0; /* at_init[1] */
    work[7]=0; /* at_init[3] */
    work[8]=0; /* count_y */
    work[9]=0; /* initdone */
    work[10]=0; /* start_y */
    work[11]=0; /* stop2half */
    work[12]=0; /* stop2halfy */
    work[13]=0; /* startpointing */
    work[14]=0; /* pointingnode */
    52
work[i][j]=0;  /* stoppositioning 1 == start stoppositioning mode to return from pointingpos to zero */
work[i][j]=0;  /* pointingdone 1 == pointing done == x-axis @ xdes */
work[i][j]=0;  /* pointingdone 1 == pointing done == y-axis @ ydes */

/* startpointing */
if (startpointing==2)

/* startpointing */
if (startpointing==1)

/* startpointing */
if (startpointing==0)

/* startpointing */
if (startpointing==0)

static void mdInputSp(SimStruct &s, int T tid)
{
int T

InputMotorType i1 = loadInputMotorType(5, 0, 0);  /* set 0 */
InputMotorType i2 = loadInputMotorType(5, 1, 0);  /* set 0 */
InputMotorType i3 = loadInputMotorType(5, 2, 0);  /* set 0 */
InputMotorType i4 = loadInputMotorType(5, 3, 0);  /* set 0 */
InputMotorType i5 = loadInputMotorType(5, 4, 0);  /* set 0 */
InputMotorType i6 = loadInputMotorType(5, 5, 0);  /* set 0 */

real T

*work I = ssGetWorkspace(0);  /* output acceleration */
*work I = ssGetWorkspace(0);  /* output velocity */
*work I = ssGetWorkspace(0);  /* input position */
*work I = ssGetWorkspace(0);  /* input velocity */

count x=work[i][0];
count y=work[i][1];
start x=work[i][2];
start y=work[i][3];
isitdone=work[i][4];
stopshalfy=work[i][5];
startpointing=work[i][6];
startpointing=work[i][7];
pointingmode=work[i][8];
pointingdone=work[i][9];
startposing=work[i][10];
startposing=work[i][11];
startposing=work[i][12];
startposing=work[i][13];
startposing=work[i][14];
startposing=work[i][15];
startposing=work[i][16];
startposing=work[i][17];
startposing=work[i][18];
startposing=work[i][19];
startposing=work[i][20];
startposing=work[i][21];

if (pointingmode == 0) {
if (isitdone == 0 && pointingdone == 0 && start x == 0 && startpointing == 0)

work[2][0]=ins[0];
/* wait for start scan mode */
if (*ins[0] == 1) {

work [2][0]=ins[0];
/* require desired pointing angle and save it*/
work[1][1]=startpointing==1;
work[1][1]=startpointing==1;
work[1][1]=startpointing==1;
work[1][1]=startpointing==1;
if (count_x == 0 ) {
    for (i=0; i<1; i++) {  /* Getting New Input Values*/
        work[1]=i*1[1];  /* = at[0]  */
    for (i=1, i<4, i++) {
        work[i]=i*1[3][1];  /* = at_init[0..3] = work[4..7] */
    }
}

if (count_x < work[0] )
    {as=work[0];}  /* a=ab*/
else if (count_x == work[0] & count_x < work[1])
    {as=work[1];}  /* a = 0*/
else if (count_x > work[1] & count_x-1 <= work[1] )
    {as=work[2];}  /* a=ab*/
else if (count_x > work[7] )
    {as=0.0;}  /* a=ab*/

/* --------- Integrators ------------------------------------------ */
work[14]=work[14]+(y2)/time;

/* --------- Reset Integrators -----------------------------------*/
if (count_x == 0 || count_x == work[7] )  /*Reset Integrator velocity*/
    {work[13]=0;}
else if (count_x == 0)  /*Reset Integrator position*/
    {work[14]=0;}

/* ------ generate outputs ---------------------------------------- */

  /* 0;  */
  }
  }
}
/* end hosing x procedure*/

if (inityline == 0 & (starts == 1 || startx == 1) && startpointing == 0 )  /* init procedure for y-axis. Starts second. */
tstarty=r-work[1]=0.5*(work[2]-work[1]);  /*ta=0.5*y -- time [count_x] to start ity and count_y*/
tstarty-floor(tstarty+.5);  /*ta=0.5*y -- time [count_x] to start ity and count_y*/

if (count_x == tstarty) {
    starty=1;  work[10]=starty;}

if (starty == 1) {
    if (count_y < work[0] )
        {as=work[0];}  /* a=ab*/
    else if (count_y == work[0] & count_y < work[1])
        {as=work[1];}  /* a = 0*/
    else if (count_y > work[1] & count_y-1 <= work[1] )
        {as=work[2];}  /* a=ab*/
    else if (count_y > work[7] )
        {as=0.0;}  /* a=ab*/

/* --------- Integrators ------------------------------------------ */
work[16]=work[16]+(y3)/time;  /* r_y*/
work[17]=work[17]+(y4)/time;  /* p_y*/

/* --------- Reset Integrators -----------------------------------*/
if (count_y == 0 || count_y == work[7] )  /*Reset Integrator velocity*/
    {work[16]=0;}

/* ------ generate outputs ---------------------------------------- */

  /* 0;  */
  }
  }
}
/* end hosing x procedure*/

if (count_y==work[7] )
    {work[8]=0;}  }
if (count_x==work[7] )
    {work[8]=0;}  }
54
{ work[8]=0; const_y=0; /* const_y=0 */
a=0.0; work[10]=a; y6=a;
work[9]=1; isnydone=1; }
} /* end starty */
} /* end init y procedure */

if ( isnydone = 1 ) { /* start scan for x-axis */
tend=work[6]; /* end time cyclus in const_x */
tend=2.floor(work[8]/2)=0.5; /* =0.5 == rounding */
if ( count_x = 0 ) { count_x = tend/2) { /* stop scan node if *in4[0] = 0 */
if ( *in4[0] = 0 ) { stopsx=0; work[2]=startx; /* stop scan node and start pointing node */
if ( startpointing = 0 ) { / * x-axis starts pointing first and x-axis will return to zero */
else if ( startpointing = -1 ) { count_s=0; work[7]=0; /* reset counter */
work[16]=work[14]; /* saving current amplitude in memory */
} }
if ( startx = 0 ) { (startpointing = -1 && startpointingfirst = -1) { /* stop scan node and return to zero (0,0) */
if ( count_x = tend/2 } work[8]=0; count_x=0; stophalftime=1; work[11]=1; }
if ( count_x = work[8] )
{ a=work[6]; } /* aab */
else if ( count_x = work[8] && count_x < work[8] }
{ a=work[7]; } /* a = 0 */
else if ( count_x = work[8] && count_x-1 = work[7] )
{ a=work[8]; } /* a = aab */
else if ( count_x = work[8] )
{ a=0.0; } /* aab */
/ * ----------- Integrators ------------------------------ */
/ * ----------- Reset Integrators ------------------------------- */
if ( count_x = 0 ) { /* reset integrator velocity */
work[13]=0; }
/ * ------------ generators outputs ---------------------------- */
if ( stophalftime = 0 ) { a=-a; }
*y1=a;
*y2=work[13]; /* y */
*y3=work[14]; /* p */
work[12]=a; /* storing old value of a*/
/ * ------------ othogen count_x ------------------------------- */
if ( count_x=work[8] )
{ work[10]=a; }
if ( count_x=0 )
{ work[0]=0; count_x=0; /* count_x=0 */
a=0.0; work[12]=a; y1=a; /* a = 0 */
work[13]=0.0; y2=-0.0; /* p = 0 */
work[14]=0.0; y3=0.0; /* p = 0 */
work[0]=0; isnxdone=0;
work[11]=0; stophalftime=0; }
} /* END startx = 0 - stop scan */

else if ( startx = -1 && startpointing = -1 ) { /* NEXT scan node */
/ * ------ Calculating AC as functions of count_x --------------- */
if ( count_x = work[8] )
{ a=work[6]; } /* a = aab */
else if ( count_x = work[8] && count_x < work[8] )
{ a=work[7]; } /* a = 0 */
else if ( count_x = work[8] && count_x < work[8] )
{ a=work[8]; } /* a = aab */
else if ( count_x = work[8] && count_x < work[8] )
{ a=work[9]; } /* a = 0 */
else if ( count_x = work[8] && count_x < work[8] )
{ a=work[9]; } /* a = aab */
else if ( count_x = work[8] && count_x < work[8] )
{ a=work[9]; } /* a = 0 */
else if ( count_x = work[8] && count_x < work[8] )
{ a=work[9]; } /* a = aab */
else if ( count_x = work[8] && count_x < work[8] )
{ a=work[9]; } /* a = 0 */
else if ( count_x = work[8] && count_x < work[8] )
{ a=work[9]; } /* a = aab */
else if ( count_x = work[8] && count_x < work[8] )
{ a=work[9]; } /* a = 0 */
else if ( count_x = work[8] && count_x < work[8] )
{ a=work[9]; } /* a = aab */
else if ( count_x = work[8] && count_x < work[8] )
{ a=work[9]; } /* a = 0 */
else if ( count_x = work[8] && count_x < work[8] )
{ a=work[9]; } /* a = aab */
else if ( count_x = work[8] && count_x < work[8] )
{ a=work[9]; } /* a = 0 */
else if ( count_x = work[8] && count_x < work[8] )
{ a=work[9]; } /* a = aab */
else if ( count_x = work[8] && count_x < work[8] )
{ a=work[9]; } /* a = 0 */
else if ( count_x = work[8] && count_x < work[8] )
{ a=work[9]; } /* a = aab */
else if ( count_x = work[8] && count_x < work[8] )
{ a=work[9]; } /* a = 0 */
else if ( count_x = work[8] && count_x < work[8] )
{ a=work[9]; } /* a = aab */
else if ( count_x = work[8] && count_x < work[8] )
{ a=work[9]; } /* a = 0 */
else if ( count_x = work[8] && count_x < work[8] )
{ a=work[9]; } /* a = aab */
else if ( count_x = work[8] && count_x < work[8] )
{ a=work[9]; } /* a = 0 */
else if ( count_x = work[8] && count_x < work[8] )
{ a=work[9]; } /* a = aab */
else if ( count_x = work[8] && count_x < work[8] )
{ a=work[9]; } /* a = 0 */
else if ( count_x = work[8] && count_x < work[8] )
{ a=work[9]; } /* a = aab */
else if ( count_x = work[8] && count_x < work[8] )
{ a=work[9]; } /* a = 0 */
else if ( (count_x >= work[4] && count_x-1 <= tend) )
    { az=work[10]; } /* az = a*b*/
/* --------- Integrators x-axis ------------------------------------- */
work[13]=work[13]+x*work_x/stime; /* v_x*/
work[14]=work[14]+y*work_y/stime; /* p_x*/
/* --------- Reset Integrators -----------------------------------------*/
if (count_x == 0 || count_x == tend+2) /*Reset Integrator v_x*/
{ work[13]=0; }
/* ------ generator outputs ----------------------------------------- */
  v1=ax;
  v2=work[13];
  v3=work[14];
  work[12]=ax; /*Storing old value of ax*/
/* --------- Ophuges count_x ------------------------------------------*/
if (count_x==tend)
    { work[0]=1; } /*end startx =1*/
/*end initizone =-1*/
if (initzone == 1) {
    /*start scan Y axis*/
    if ((count_y == 0 || count_y == tend+2) && initzone == 1 && startpointingy == 0) {
        if (startx == 0) {
            starty=0; work[10]=starty; /*Stop scan mode if +1=0 */
            isin[0]=1; work[13]=startpointingy; work[20]=isin[0]; /* Require desired pointing angle and save it*/
            if (startpointingy == 0) {
                startpointingyfirst=1; work[19]=1; /* y-axis starts pointing first and y-axis will return to zero*/
                startpointingyfirst=0; work[18]=0; /* reset counter*/
                work[12]=work[17]; /* saving current amplitude in memory*/
                } /*Stop scan mode and start pointing mode*/
        }
    }
    /*stop scan mode and return 2 zero*/
    if (count_y == tend+2) {
        work[0]=0; count_y=0; /*count_y=0*/
        stophalfy=1; work[12]=stophalfy; }
    if (count_y < work[5])
        { ay=work[6]; } /* a=ab*/
    else if (count_y >= work[5] && count_y < work[6])
        { ay=work[7]; } /* a= 0*/
    else if (count_y >= work[6] && count_y-1 <= work[7])
        { ay=work[8]; } /* a=ab*/
    else if (count_y >= work[7])
        { ay=0.0; } /* a=ab*/
    /* --------- Integrators y-axis ------------------------------------- */
    work[16]=work[16]+y*work_y/stime; /* y_y*/
    work[17]=work[17]+y*stictime; /* p_y*/
    /* --------- Reset Integrators ----------------------------------------*/
    if (count_y == 0 || count_y == tend+2) /*Reset Integrator y_y*/
    { work[16]=0; }
    /* ------- generator outputs ---------------------------------------- */
    if (stophalfy == 0) {
        ay=ay; /*y*/
        y1=work[16];
        y2=work[17];
        work[16]=ay; /*Storing old value of ay*/
    } /* ------- Ophuges count_y -----------------------------------------*/
    if (count_y>=work[7])
        { work[0]=1; } /*count_y=0*/
    if (count_y>=work[7])
        { work[0]=0; count_y=0; } /**count_y=0*/
ay=0.0; work[16]=ay; y6=ay; /* a=0*/
work[16]=0.0; y6=0.0; /* t=0*/
work[17]=0.0; y6=0.0; /* p=0*/
work[9]=0; initxdone=0;
work[1]=0; stopxhalfy=0;
} /*END stop scan loop*/
else if (starty == 1 && startpointing == 0) { /*START scan mode*/
  /* ------ Calculating acc as functions of count_y ----------------- */
  if ( count_y < work[1] )
    { sy=work[6];
        sy=work[7];
        sy=work[8];
        sy=work[9];
      else if ( count_y >= work[4] && count_y-1 < tend )
        sy=work[10];
      /* a = -ab*/
    }
  } /* ------ Integrators y-axis --------------------------------- */
  work[16]=work[16]-y6/stime; /* v_y*/
  work[17]=work[17]+y6/stime; /* p_y*/
  */ /* ------ Reset Integrators -------------------------------------- */
  if (count_y == 0 || count_y == tend+1 ) /*Reset Integrator v_y*/
    {work[16]=0;
      /* ------ generate output ---------------------------------------- */
      ay=ay;
y5=work[16];
y6=work[17];
    work[16]=ay; /* storing old value of ay*/
  } /* ------ oploges count_y ---------------------------------------- */
  if (count_y>tend )
    {work[9]=s;}
  if (count_y==tend )
    {work[8]=s;}
} /*END initxdone */
if (startpointing == 1 && startpointing == 1) {
  if (initxdone == 0 || initxdone == 0 )
    { if (startpointingfirst == 0 )
      initxdone=0; work[3]=0; / initxdone has been reset in end of 'returning 2 zero' loop*/
      if (startpointingfirst == 0 )
        { initxdone=0; work[3]=0; / initxdone has been reset in end of 'returning 2 zero' loop*/
      }
  if (startpointingfirst == 0 )
    { startpointing2=1; work[20]=1; /x-axis @ max amplt. AND y-axis @ zero*/
      if (startpointingfirst == 1 && initxdone == 0 )
        { startpointing2=1; work[21]=1; /x-axis @ max amplt. AND y-axis @ zero*/
      if (startpointingfirst == 1 && initxdone == 0 )
        { startpointing2=1; work[21]=1; /x-axis @ max amplt.*/
      if (startpointingfirst == 1 && initxdone == 0 )
        { startpointing2=1; work[20]=1; /x-axis @ max amplt.*/
      if (startpointingfirst == 1 && initxdone == 0 )
        { startpointing2=1; work[20]=1; /x-axis @ max amplt.*/
      }
      if ( startpointing2 == 1 && pointsdone == 0 )
        /*start transition from standard pos. to desired angle*/
      xdes=amplsin(fabs(work[20]))/360*2*pi1-work[16]; / work[20]=alpha_des and work[16]=current ampl. */
      /* ------ Calculating ta (Constant Acc phase) and tv (Constant Vel phase) ------ */
      ta= sqrt(fabs(xdes)/ab); /sqrt(ax)/
      if ( y<eb+tar )
        { y bound violated /**/
        }
      y=eb+tar;
      tv=tar*(fabs(xdes)+eb*tar)/y; /* Const Velocity Time */
      if (floor(tar*1000)<0.5)
        { /*rounding*/
          tv=floor(tar*1000*0.999); /*rounding*/
        } else
          { /*rounding*/
            tv=floor(tar*1000*0.999); /*rounding*/
          }
at[0]=ta; at[1]=ta+tv; at[2]=ta+tv+ta;

if (count_x < at[0] )
{ t=ab; /* t=ab*/
else if (count_x >= at[0] && count_x < at[1] )
{ t=0.0; /* t=0*/
{ t=ab; /* t=ab*/
else if ( count_x >= at[2] )
{ t=0.0; /* t=ab*/

/* ---------- integrators ----------------------------------------------- */

/* ------- generates outputs ------------------------------------------ */
if (ydes < 0) {
 z-=a1;

*y1=;
*y2=work[13]; /* r */
*y3=work[14]; /* p */
work[12]=a; /* storing old value of a*/

/* ------- opposes count_x ------------------------------------------- */
if (count_x<at[2] )
{ (work[10])+=1;
  if (count_x>=at[2] )
  { work[0]=0; count_x=0; /* count_x=0*/
    t=-0.0; /* t=-0*/
    work[13]=0.0; /* y2=0.0 */
    work[18]=work[14]; /* moving current ampl.*/
    pointing=1; work[16]=pointing;}
} /*END if startpointing/y2=1*/

if (startpointing/y2=1 && pointing2deme = 0) { /*start transition from standard pos. to desired angle*/
    ydes=ampl*cos(fabs(work[20])/360<180) -work[19];

    /* calculating ta (constant acc phase) and tv (constant vel phase)----*/
    tar=sqrt(fabs(ydes)/ab);

    if (y<ab*tar) /* y bound violated ? */
    { t=tv/ab;}

    y=ab*tar;
    tv=-fabs(ydes/ab)*tar*tar/ab; /* Const Velocity Time */

ta=floor(tar/1000.00); /* rounding*/

    tv=floor(tv/1000.00);

    at[0]=ta; at[1]=ta+tv; at[2]=ta+tv+ta;

    if (count_y < at[0] )
    { t=ab; /* t=ab*/
else if ( count_y >= at[0] && count_y < at[1] )
    { t=0.0; /* t=0*/
    { t=ab; /* t=ab*/
else if ( count_y >= at[2] )
    { t=0.0; /* t=ab*/

/* ---------- integrators y-axie ------------------------------------ */
work[16]=work[16]+old_y/y/time; /* t_y*/
work[17]=work[17]+y2/y/time; /* p_y*/

/* ------- generates outputs ----------------------------------------- */
if (ydes < 0) {
 z-=a1;

*y4=;
*y5=work[16]; /* y */
*y6=work[17];
work[16]=a; /* storing old value of ay*/

/* ------- opposes count_y ------------------------------------------ */
}
if (count_y<at[2])
   { count_y+=1; }
if (count_y>at[2])
   { count_y-=1; }
if (xspec==)
   { 
      xspec[8]=0;
      count_x=0; /*count_x= */
      yspec[16]=0;
      yspec[16]=0;
      yspec[16]=0;
      yspec[16]=0;
      yspec[16]=0;
      yspec[16]=0;
      yspec[16]=0;
      /* saving current amplitude in memory*/
      printingdone=1; work[17]=printingdone;
   }
} /*END if startpointings=1*/
} /*END if startpointings && startpointing =1*/

if (printings done =1 && pointings done =1) {
   pointings done =0;
   work[16]=0;
   startx=0;
   work[16]=0;
   starty=0;
   work[1]=0;
   startpointings 2=0;
   work[20]=0;
   startpointing first=0;
   work[18]=0;
   
   printingdone=1;
   work[17]=printingdone;
   }
} /*END pointings done=0 loop*/
else if (pointings done =1) {
   /*pointing...Wait for new pointing angle*/
if (startpointings2 == 0 && startpointings2 == 0) { /* saving new alpha_des only once @ start of pointings=1 loop*/
if (ins[0] == 0) { /*stop pointings*/
   stoppointings=1;
   work[1]=1;
   startpointings2=1;
   work[20]=1;
   startpointing first=1;
   work[21]=1;
   }
else if (work[20] != ins[0]) { /* new pointing angle ?*/
   work[20]=ins[0];
   yspec+=ampl*cos(fabs(work[20])/360.0+1); /* new desired absolute y-pos*/
if (work[19] > 0.0) {
   /* current y-pos. > 0 ? */
   if (ydes > work[19]) {
      /* current y-pos*/
      startpointings first=1;
      work[18]=1;
      startpointings 2=1;
      work[20]=1;
      /* Start x*/
      if (ydes < work[19]) {
         /* current y-pos*/
         startpointings first=1;
         work[19]=1;
         startpointings 2=1;
         work[21]=1;
         /* Start y*/
   }
} else if (work[19] < 0.0) {
   /* current y-pos. < 0 ? */
   if (ydes > work[19]) {
      /* current y-pos*/
      startpointings first=1;
      work[19]=1;
      startpointings 2=1;
      work[21]=1;
      /* Start x*/
   } else if (work[19] < 0.0) {
      /* current y-pos*/
      startpointings first=1;
      work[21]=1;
      startpointings 2=1;
      work[21]=1;
      /* Start y*/
   }
} /* END*/
} else if (count x != at[0] )
else if (count x > 0.0) { /* constant x */
   startpointings 1=1;
   work[18]=1;
   startpointings 2=1;
   work[20]=1;
   /* Start x*/
} /*END if startpointings 1*/

if (startpointings 2 == 1 && pointings done == 0 && stoppointings == 0) { /*start transition from old pos. to new desired angle*/
   xdes=ampl*sin(fabs(work[20])/360.0+1)*work[18]/relative new x-pos... work[20]=alpha_des and work[18]=current ampl. */
   tar=qrt(fabs(xdes/ab));
if (yhcab+tar) { /* y bound violated ? */
   {tar=m/abs;}
   yb=tar;
   ytr= fabs(xdes)=-+tar+tar;r; /*Const Velocity Time */
   ta= floor(tar/1000.0+0.5); /*rounding*/
   tr= floor(ytr/1000.0+0.5); /*rounding*/
   at[0]=ta; at[1]=ta; at[2]=ta; tr=ta; pt, tend= floor(at[2]/DIV+0.5);
```c
{ a = sb; }
else if ( count_x >= at[0] && count_x < at[1] )
    { a=0.0; } /* a=ab*/
    { a=0; } /* a=0*/
else if ( count_x >= at[2] )
    { a=0.0; } /* a=ab*/
/* ----------------- integrators ----------------------------- */

/* ------ generators outputs ----------------------------------- */
if (ydes < 0) { a=a1; }
  *y1=sa;
  *y2=work[13]; /* y*/
  *y3=work[14]; /* p */
  work[12]=a; /* Storing old value of a*/
/* ------ opgesum count_x -------------------------------------- */
if ( count_x(at[2]) )
{ (work[0])=++;
  work[0]=0.0; count_x=0; /* count_x<0*/
  a=0.0; /* a=0*/
  work[13]=0.0; /* y2=0.0*/
  work[16]=work[14]; /* saving current ampl.*/
  *pointingdone=1;
  work[16]=pointingdone;
}
} /*END if startpointing2 = 1*/

if (startpointing2 = 1 && pointingdone = 0 && stoppointing = 0) { /*start transition from old pos. to new desired angle*/
  ydes=ampl*cos(fabs(work[201])/360+0+1)-work[19];
/* ---------------------- Calculating ta (Constant Acc phase) and tv (Constant Vel phase) ---------------- */
  tar=sqrt(fabs(ydes)/ab);
if (y<ab+tar) /* y bound violated ? */
  { tar=y/ab; }
  *y=ab-tar;
  tv=sqrt(fabs(ydes)-ab+tar+tar)/r; /* Const Velocity Time */
  ta=floor(tar+0.00+0.5); /* rounding*/
  tv=floor((tv+0.00+0.5); 
  at[0]=ta; at[1]=at+tv; at[2]=ta+tv+ta; 
  pntena=pnten(at[2])/div+0.5;
if (count_y < at[0] )
  { a=ab; } /* a=ab*/
else if ( count_y >= at[0] && count_y < at[1] )
  { a=0.0; } /* a=0*/
  { a=ab; } /* a=ab*/
else if ( count_y >= at[2] )
  { a=0.0; } /* a=ab*/
/* ------------------ z-axis integrators ------------------------ */
vork[16]=vork[16]+old_y/time; /* y*/

/* ------ generators outputs ----------------------------------- */
if (ydes < 0) { a=a1; }
  *y1=sa;
  *y1=work[16];
  *y1=work[17]; /* Storing old value of y*/
/* ------ opgesum count_y -------------------------------------- */
if ( count_y <at[2] )
{ (work[6])=++;
}
```
if ( count_y==at[2] ) {
    work[8]=0;
    count_y=0; /*count_y=0*/
    a=0.0;
    work[15]=a; /*a=0*/
    work[16]=0.0; /*y=0*/
    work[19]=work[17]; /* saving current amplitude in memory */
    pointingdone=1; work[17]=pointingdone;
} //END if startpointing2 == 1*/

if ( startpointingfirst == 0 && startpointingy2 == 1 && stoppingy == 0 ) {
    if ( count_y == pn_temdy ) {
        startpointing2=1; work[20]=i; /* CASE 1 - Determine when x-axis starts (y-axis already running)*/
    }
    else if ( startpointingfirst == 0 && startpointingy2 == 1 && stoppingy == 0 ) {
        if ( count_x == pn_temdx ) {
            startpointing2=1; work[21]=i; /* CASE 2 - Determine when y-axis starts (x-axis already running)*/
        }
    }
}

if ( pointingdone == 1 && pointingdone == 1 && stoppingy == 0 ) { /*both axis ready with pointing move; Reset flags*/
    pointingdone=0; work[16]=0;
    startpointingx=0; work[1]=0;
    startpointingy=0; work[2]=0;
    startpointingfirst=0; work[18]=0;
    pointingdy=0; work[17]=0;
    stoppointingx=0; work[20]=0;
    stoppointingy=0; work[21]=0;
    stoppointingfirst=0; work[19]=0;
}

if ( stoppingy == 1 ) {
    /* Start transition from pointing pos. to zero */
    if ( startpointingy2 == 1 && pointingdone == 0 ) { /* start transition for y-axis*/
        zdes=0.0-work[15];
        /* ----- Calculating ta (Constant Acc phase) and tv (Constant Vel phase) ----- */
        tar=gtar(fabs(zdes)/ab);
        if ( yb<ab*tar ) /* y bound violated ? */
            {tar=yb/ab;}
        y=ab*tar;
        tv=(fabs(zdes)-ab*tar*tar)/y; /* Const Velocity Time */
        ta-tar*1000/0.5; /*rounding*/
        tv-tar*1000/0.5;
        pn_temdx=at[2]/0110+0.5;
        if ( count_x < at[0] )
            {a=ab;} /* a=ab*/
        else if ( count_x >= at[0] && count_x < at[1] )
            {a=0.0;} /* a= 0*/
            {a=-ab;} /* a= -ab*/
        else if ( count_x > at[2] )
            {a=0.0;} /* a= 0*/
        /------------------ Integrators ------------------------------*/
        /---------------------- generators outputs ----------------------*/
        if ( zdes < 0 )
            a=a+1;
        y1=1; /*y1=work[13]; /* y */
        y2=work[14]; /* p */
        work[12]=a; /* Storing old value of a*/
        /---------------------- oplogens count_x ----------------------*/
        if ( count_x<at[2] )
            {work[12]=work[12];}
        else (count_x=at[2] )
        }

61
```c
work[0]=0;  count_x=0;  /*count_x=0*/
a=0.0;  /* a=0.0*/
work[1]=0.0;  /* y=0.0*/
work[1][0]=0.0;  /* y2=0.0*/
work[1][1]=work[14];  /* moving current ampl.*/
pointingdone=1;  work[16]=pointingdone;
} /* END if startpointing2 == 1 && pointingdone == 0 */

if (startpointing2 == 1 && pointingdone == 0) {  /*start transition for y-axis*/
  ydes=0.0-work[19];
  /* --------- Calculating ta (Constant Acc phase) and tv (Constant Vel phase) --------- */
ta=sqrt(fabs(ydes)/a);
  if (yv<atar)  /* v bound violated ? */
    {tar=atavb;}
  yv=atavb;
  tv=(fabs(ydes)-atavb-tar)/v;  /* Const Velocity Time */
ta=fabs(tar);tv=fabs(tv);at=st[0]-ta;st[1]-ta-st[2];st[3]-ta-st[4];
  if (count_y < at[0] )
    {a=ab;}
  else if (count_y > at[0] && count_y < at[1])
    {a=atavb;}
    {a=ab;}
  else if (count_y > at[2] )
    {a=atavb;}
  /* --------- integrators y-axis ------------------------------- */
  work[16]=work[16]+old_y/stime;  /* y_y*/
  work[17]=work[17]+yv/stime;  /* p_y*/
  /* --------- generates outputs ------------------------------- */
  if (ydes < 0) {
    a=a-
  }
  yv-y;  /*y=work[16];*/
  yv=work[17];
  work[16]=a;  /*Storing old value of yv*/
  /* --------- opcodes count_y ------------------------------- */
  if (count_y < at[2] )
    { (work[1][8]++)}
  else if (count_y > at[2])
    { (work[1][8]++)}
    work[8]=0;  count_y-0;  /*count_x=0*/
    a=0.0;  /* a=0.0*/
    work[16]=0.0;  /* y2=0.0*/
    work[14]=work[17];  /* y=0.0*/
    work[16]=0.0;  /* y2=0.0*/
    work[17]=work[17];  /* moving current amplitude in memory*/
    pointingdone=0;  work[17]=pointingdone;
} /* END if startpointing2 == 1*/

if (pointingdone == 1 && pointingdone == 0) {  /*transition to zero ready -> Stop pointing mode and reset flags*/
  pointingdone=0;  work[14]=0;
  stoppointing=0;  work[16]=0;
  startpointing2=0;  work[10]=0;
  startpointing2=0;  work[12]=0;
  pointingdone=0;  work[16]=0;
  pointingdone=0;  work[17]=0;
} /* END else if pointingdone == 1 */
/* END m3I1Output*/

/* Function: m3I1Terminate -------------------------------------*/
/* Abstract: */
/*  No termination needed, but we are required to have this routine.*/
static void nlTerminate(SimStruct *S)
{
}

#ifdef MATLAB_MEX_FILE   /* Is this file being compiled as a MEX-file? */
#include "simlink.h"       /* MEX-file interface mechanism */
#else
#include "cgsfun.h"        /* Code generation registration function */
#endif