Chapter 10: Appendix C - Computer Algorithm for Simulator

Time-related variables

t: time, in seconds after some base date/time
dt: basic time-step e.g. 2*s* for calculation of all actuator loops and plant models
dtc: time-step e.g. 10*s* for calculation of main control algorithm

Indices and numbers of indices

i: 1 to number of plant inputs
j: 1 to number of plant outputs
ku: 1 to number of time-shifts of length *dt* for dead-time states of plant inputs
kx: 1 to number of time-shifts of length *dt* for dead-time states of plant states
ky: 1 to number of time-shifts of length *dt* for dead-time states of plant outputs
mku: maximum number of dead-time states of length *dt* provided for storing plant inputs
mkx: maximum number of dead-time states of length *dt* provided for storing plant states
mky: maximum number of dead-time states of length *dt* provided for storing plant outputs

Conventions

The variables were given multi-character names (convenient for programming). Three basic variables were specified for the simulation of the plant:

u: Contains all input variables
x: Contains all state variables
y: Contains all output variables

Plant input variables (*u[i]*)

u[i].PI_Setpoint: main controller output, provided as a setpoint for a local PI controller
u[i].PI_last: value of *u[i].PI_Setpoint* at last local PI increment
u[i].local: signal to actuator from local PI controller
u[i].slack: modified *u[i].local* to provide another state incorporating actuator slack
u[i].state: 1st-order response of actuator to *u[i].slack*
u[i].smooth: noise-free input to plant, a non-linear function (valve characteristic) of *u[i].state*
u[i].noise: state for additive noise to measurement of input to plant
u[i].local_meas: local PI loop's measurement of input to plant
u[i].local_meas_last: value of *u[i].local_meas* at last local PI increment
u[i].meas: main controller's measurement of input to plant, filtered according to dtc
u[i].disturb: state for additive disturbance to plant input
u[i].input: final actual input to plant
u[i].input_last: final actual input to plant at time *dt* ago
u_dead[i,ku]: shift vector to provide for dead time of actuator

Plant input parameters (*u[i].p*)

u[i].p.knob:	multiplier "knob" (0 to 100 %) for all plant-input non-ideal parameters
u[i].p.base0:	initial base level for *u* used at initialisation
u[i].p.base:	drifting base level for *u*
u[i].p.drift_base:	drift rate of base level for *u*
u[i].p.min:	minimum *u*
u[i].p.max:	maximum *u*
u[i].p.soft0.min:	initial minimum achievable *u* used at initialisation
u[i].p.soft.min:	drifting minimum achievable *u*
u[i].p.drift.min:	drift rate of minimum achievable *u* (*u[i].p.soft.min*)
u[i].p.soft0.max:	initial maximum achievable *u* used at initialisation
u[i].p.soft.max:	drifting maximum achievable *u*
u[i].p.drift.max:	drift rate of maximum achievable *u* (*u[i].p.soft.max*)
u[i].PI.gain:	gain of local PI controller
u[i].PI.ti:	integral time constant of local PI controler
u[i].p_actuator.gain:	actuator gain
u[i].p_actuator.tau:	actuator time constant
u[i].p_actuator.tdead:	dead-time of actuator's response
u[i].p_slack:	two-sided dead-band to get from *u[i].state* to *u[i].slack*
u[i].p_nonlinear:	non-linearity of valve (-1 to +1, with 0 for linear) to get from *u[i].slack* to *u[i].smooth*
u[i].p_noise.gain:	magnitude of noise, to update *u[i].noise*
u[i].p_noise.tau:	time constant for filter of noise, to update *u[i].noise*
u[i].p_tdead:	dead-time of plant input (excluding actuator's)
u[i].p_disturb.gain:	magnitude of disturbance, to update *u[i].disturb*
u[i].p_disturb.tau:	time constant for filter of disturbance, to update *u[i].disturb*
u[i].p_offset:	offset to get *u[i].input*

Calculated plant-input parameters (*u[i].c_*)

u[i].c_exp:	0 if *u[i].p_actuator.tau* = 0, else *exp(-dt/ u[i].p_actuator.tau)*
u[i].c_exp_noise:	0 if *u[i].p_noise.tau* = 0, else *exp(-dt/ u[i].p_noise.tau)*
c_exp_u_dtc	exp(-dt/dtc)
u[i].c_exp_disturb:	0 if *u[i].p_disturb.tau* = 0, else *exp(-dt/ u[i].p_disturb.tau)*
u[i].c_ku_actuator:	*Minimum(Integer(1+u[i].p_actuator.tdead/dt), mku)*
u[i].c_ku_plant:	*Minimum(Integer (1+(u[i].p_actuator.tdead+u[i].p_tdead)/dt), mku)*

Initialisation of plant-input variables (at *t*=0)

u[i].p.base	= *u[i].p.base0*
u[i].PI_setpoint	= *u[i].p.base0*
u[i].PI_last	= *u[i].PI_setpoint*
u[i].local	= *u[i].PI_setpoint*
u[i].slack	= *u[i].PI_setpoint*
u[i].state	= *u[i].PI_setpoint*
u[i].smooth	= *u[i].PI_setpoint*

u[i].noise = 0
u_dead(i,ku) = *u[i].PI_setpoint*, for all *ku*
u[i].local_meas = *u[i].PI_setpoint*
u[i].local_meas_last = *u[i].PI_setpoint*
u[i].meas = *u[i].PI_setpoint*
u[i].disturb = 0
u[i].input = *u[i].PI_setpoint*

Plant-input dynamics (to be executed every dt seconds)

```
//Calculate PI control for future over dt
if (u->PI.gain == 0)
        u->local += (u->PI_setpoint-u->PI_last);
else {
        if (u->PI.ti == 0)
                u->local+=u->PI.gain*((u->PI_setpoint-u->PI_last)-(u->local_meas-u->local_meas_last));
        else
                u->local += u->PI.gain*((u->PI_setpoint-u->PI_last)-(u->local_meas-u->local_meas_last)+
                (u->PI_setpoint-u->local_meas)*sim_param.dt/u->PI.ti);
}

u->local_meas_last = u->local_meas;
u->PI_last = u->PI_setpoint;

//If not generating the model for the MPC-controller
if (!model){

        //Update soft maximum and restrict to hard limits
        u->p.soft.max += (u->p.drift.max * sim_param.dt) * u->p.knob;
        if (u->p.soft.max > u->p.max) u->p.soft.max = u->p.max;
        if (u->p.soft.max < u->p.min) u->p.soft.max = u->p.min;

        //Update soft minimum and restrict to between hard minimum and soft maximum
        u->p.soft.min += (u->p.drift.min * sim_param.dt) * u->p.knob;
        if (u->p.soft.min < u->p.min) u->p.soft.min = u->p.min;
        if (u->p.soft.min > u->p.soft.max) u->p.soft.min = u->p.soft.max;

        //Update base level drift and restrict to soft limits
        u->p.base += (u->p.drift_base * sim_param.dt) * u->p.knob;
        if (u->p.base < u->p.soft.min) u->p.base = u->p.soft.min;
        if (u->p.base > u->p.soft.max) u->p.base = u->p.soft.max;

        //Limit actuator, allowing slack on both sides
        if (u->local > u->p.max + u->p_slack) u->local = u->p.max + u->p_slack;
        if (u->local < u->p.min - u->p_slack) u->local = u->p.min - u->p_slack;

        //Calculate slack state (Unchanged if within the slack band)
        if (u->local > u->slack + u->p_slack) u->slack = u->local - u->p_slack;
        if (u->local < u->slack - u->p_slack) u->slack = u->local + u->p_slack;

}
else
        u->slack = u->local;

//Update state
u->state = u->state * u->c_exp + (u->slack*u->p_actuator.gain)*(1 - u->c_exp);
```

```
//Calculate nonlinearity and limit it to soft limits
//Function: y = (1+a)x + ax^2.      a between -1 and 1
if (u->p.soft.max == u->p.soft.min)
        temp = 1;
else
        temp = (u->state - u->p.soft.min) / (u->p.soft.max - u->p.soft.min); //Scale variable between 0 and 1;
u->smooth = (1.0 + u->p_nonlinear) * temp - u->p_nonlinear * temp * temp;

//Scale back to normal values
u->smooth = u->smooth * (u->p.soft.max - u->p.soft.min) + u->p.soft.min;

//Store in shift vector (dead time)
for (long ku=(u->c_ku_plant-1); ku>0; ku--)
        u_dead[ku] = u_dead[ku-1];
u_dead[0] = u->smooth;

//Measurement noise
if (!model)
        u->noise = u->noise*u->c_exp_noise + ( (double(rand())/RAND_MAX)*2 - 1)*u->p_noise.gain*
        (1 - u->c_exp_noise);

//Extract local measurement (with dead time) and add noise
u->local_meas = u_dead[u->c_ku_actuator-1] + u->noise;

//Update disturbance
if (!model)
        u->disturb = u->disturb + ( (double(rand())/RAND_MAX)*2 -1)*u->p_disturb.gain*(1 - u-
>c_exp_disturb);

//Extract plant input (with dead time) and add disturbance
u->input_last = u->input;
u->input = u_dead[u->c_ku_plant-1] + u->disturb + u->p_offset;

//Update measurement for main controller
u->meas = u->meas*u->c_exp_dtc + u->local_meas*(1 - u->c_exp_dtc);
```

Plant-state module

Plant states (*x[i,j]*)

x[i,j].base:	base level for state *x*
x[i,j].xx:	plant state associated with input *i* and output *j*
x[i,j].disturb:	state for additive noise to measurement of state *x*
x_dead(i,j,kx):	shift vector to provide for dead time of state *x*

Plant-state parameters (*x[i,j].p)*

x[i,j].knob:	multiplier "knob" (0 to 100 %) for all plant-state non-ideal parameters
x[i,j].p.base0:	initial base level for *x* used at initialisation
x[i,j].p.base:	drifting base level for *x*
x[i,j].p.drift_base:	drift rate of base level for *x*
x[i,j].p.min:	minimum *x*
x[i,j].p.max:	maximum *x*
x[i,j].p.soft0.min:	initial minimum achievable *x* used at initialisation
x[i,j].p.soft.min:	drifting minimum achievable *x*

x[i,j].p.drift.min: drift rate of minimum achievable *x* (*x[i,j].p.soft.min*)
x[i,j].p.soft0.max: initial maximum achievable *x* used at initialisation
x[i,j].p.soft.max: drifting maximum achievable *x*
x[i,j].p.drift.max: drift rate of maximum achievable *x* (*x[i,j].p.soft.max*)
x[i,j].p_tf.a: steady-state gain for state *x*
x[i,j].p_tf.b: immediate-response gain for state *x*
x[i,j].p_tf.tau: 1st-order time constant for state *x*
x[i,j].p_tf.ti: integrator time constant for state *x* (0 for no integrator)
x[i,j].p_tf.tdead: dead-time for state *x*
x[i,j].p_disturb.gain: magnitude of disturbance, to update state *x*
x[i,j].p_disturb.tau: time constant for filter of disturbance, to update state *x*

Calculated plant-state parameters (*x[i,j].c_*)

x[i,j].c_exp: 0 if *x[i,j].p_tf.tau* = 0, else *exp(-dt / x[i,j].p_tf.tau)*
x[i,j].c_exp_disturb: 0 if *x[i,j].p_disturb.tau* = 0, else *exp(-dt / x[i,j].p_disturb.tau)*
x[i,j].c_kx: *Minimum(Integer (1+ (x[i,j].p_tf.tdead)/dt), mkx)*

Initialisation of plant-state variables (at *t*=0)

x[i,j].p.base = *x[i,j].p.base0*
x[i,j].base = *x[i,j].p.base*
x[i,j].xx = *x[i,j].p.base*
x_dead(i,j,kx) = *x[i,j].p.base*, for all *kx*
X[I,J].DISTURB = 0

Plant-state dynamics (to be executed every *dt* seconds)

```
//If not generating the model for the MPC-controller
if (!model) {

        //Update soft maximum and restrict to hard limits
        x[0].p.soft.max += (x[0].p.drift.max * sim_param.dt) * x[0].p.knob;
        if (x[0].p.soft.max > x[0].p.max) x[0].p.soft.max = x[0].p.max;
        if (x[0].p.soft.max < x[0].p.min) x[0].p.soft.max = x[0].p.min;

        //Update soft minimum and restrict to between hard minimum and soft maximum
        x[0].p.soft.min += (x[0].p.drift.min * sim_param.dt) * x[0].p.knob;
        if (x[0].p.soft.min < x[0].p.min) x[0].p.soft.min = x[0].p.min;
        if (x[0].p.soft.min > x[0].p.soft.max) x[0].p.soft.min = x[0].p.soft.max;

}

for (j=0; j<tfdim; j++)
{

        if (!model) {

                //Update base level drift and restrict to soft limits
                x[j].p.base += (x[j].p.drift_base * sim_param.dt) * x[j].p.knob;
                if (x[j].p.base < x[j].p.soft.min) x[j].p.base = x[j].p.soft.min;
                if (x[j].p.base > x[j].p.soft.max) x[j].p.base = x[j].p.soft.max;
        }
```

```
        //Update state
        if (x[j].p_tf.ti == 0)
                x[j].xx = x[j].xx*x[j].c_exp + (x[j].p.base + x[j].p_tf.a * (u->input - u->p.base)) *
                (1 - x[j].c_exp) + x[j].p_tf.b * x[j].c_exp * (u->input - u->input_last);
        else
                x[j].xx = x[j].xx + x[j].p_tf.a*(u->input - u->p.base)*sim_param.dt;

        //Update disturbance
        if (!model)
                x[j].disturb = x[j].disturb * x[j].c_exp_disturb + ( (double(rand())/RAND_MAX)*2 - 1) *
                x[j].p_disturb.gain*(1 - x[j].c_exp_disturb);

        x[j].xx += x[j].disturb;

        //Store in shift vector (dead time)
        for (kx=(x[j].c_kx-1); kx>0; kx--)
                x_dead[j*sim_param.max_kx+kx] = x_dead[j*sim_param.max_kx+kx-1];
        x_dead[j*sim_param.max_kx+0] = x[j].xx;
}
```

Plant-output module

Plant output variables (*y[j]*)

y[j].y_output:	plant output
y[j].noise:	state for additive measurement noise to output of plant
y[j].yy:	plant output as measured
y[j].setpoint:	setpoint of output for main controller
y_dead(j,ky):	shift vector to provide for dead time of plant measurement

Plant-output parameters (*y[j].p*)

y[j].knob:	multiplier "knob" (0 to 100 %) for all plant-output non-ideal parameters
y[j].p.base0:	initial base level for *y* used at initialisation
y[j].p.base:	drifting base level for *y*
y[j].p.drift_base:	drift rate of base level for *y*
y[j].p.min:	minimum *y*
y[j].p.max:	maximum *y*
y[j].p.soft0.min:	initial minimum achievable *y* used at initialisation
y[j].p.soft.min:	drifting minimum achievable *y*
y[j].p.drift.min:	drift rate of minimum achievable *y* (*y[j].p.soft.min*)
y[j].p.soft0.max:	initial maximum achievable *y* used at initialisation
y[j].p.soft.max:	drifting maximum achievable *y*
y[j].p.drift.max:	drift rate of maximum achievable *y* (*y[j].p.soft.max*)
y[j].p_tf.tau:	steady-state gain for output *y*
y[j].p_tf.tau:	1st-order time constant for output *y*
y[j].p_tf.tdead:	dead-time for output *y*
y[j].p_noise.gain:	magnitude of noise, to update output *y*
y[j].p_noise.tau:	time constant for filter of noise, to update output *y*
y[j].p_parabola:	0 for linear, otherwise y(j)=p_y_base(j) - Σ(x(I,j)-p_x_base(i,j))^2

Calculated plant-output parameters (*y[i].c_*)

y[j].c_exp: 0 if *y[j].p_tf.tau* = 0, else *exp(-dt / y[j].p_tf.tau)*
y[j].c_exp_noise: 0 if *y[j].p_tau_noise* = 0, else *exp(-dt / y[j].p_tau_noise)*
y[j].c_ky: *Minimum(Integer(1+ y[j].p_tf.tdead / dt), mky)*

Initialisation of plant-output variables (at *t*=0)

y[j].p.base = y[j].p.base0
y[j].y_output = y[j].p.base
y_dead(j,ky) = y[j].p.base, for all *ky*
y[j].noise = 0
y[j].yy = y[j].p.base

Plant-output dynamics (to be executed every dt seconds)

```
//If not generating the model for the MPC-controller
if (!model) {
        //Update soft maximum and restrict to hard limits
        y->p.soft.max += (y->p.drift.max * sim_param->dt) * y->p.knob;
        if (y->p.soft.max > y->p.max) y->p.soft.max = y->p.max;
        if (y->p.soft.max < y->p.min) y->p.soft.max = y->p.min;

        //Update soft minimum and restrict to between hard minimum and soft maximum
        y->p.soft.min += (y->p.drift.min * sim_param->dt) * y->p.knob;
        if (y->p.soft.min < y->p.min) y->p.soft.min = y->p.min;
        if (y->p.soft.min > y->p.soft.max) y->p.soft.min = y->p.soft.max;

        //Update base level drift and restrict to soft limits
        y->p.base += (y->p.drift_base * sim_param->dt) * y->p.knob;
        if (y->p.base < y->p.soft.min) y->p.base = y->p.soft.min;
        if (y->p.base > y->p.soft.max) y->p.base = y->p.soft.max;
}

//Extract plant outputs
if (y->p_parabola == 0) {           //If not quadratic function
        temp = y->p.base;
        for (long i=0; i<tfdim; i++)
                temp += (x_dead[i*sim_param->max_kx*tfdim+x[i*tfdim].c_kx-1] - x[i*tfdim].p.base);
}
else {                              //If quadratic function
        temp = 0;
        for (long i=0; i<tfdim; i++)
                temp+=(x_dead[i*sim_param->max_kx*tfdim+x[i*tfdim].c_kx-1]-x[i*tfdim].p.base);
        temp = y->p.base - pow(temp, 2);
}
y->y_output = y->y_output * y->c_exp + temp * (1 - y->c_exp);

//Store in shift vector (dead time)
for (long ky=(y->c_ky-1); ky>0; ky--)  y_dead[ky] = y_dead[ky-1];
y_dead[0] = y->y_output;

//Update measurement noise
if (!model)
        y->noise = y->noise * y->c_exp_noise + ( (double(rand())/RAND_MAX)*2 - 1) *
        y->p_noise.gain*(1 - y->c_exp_noise);

//Produce plant measurement with dead-time and noise
```

```
y->yy = y_dead[y->c_ky-1] + y->noise;

//Calculates performance index
if (y->use_mpc) {
        if (mpc->perform == 1)
                mpc->p_index += (pow( (y->yy - y->setpoint), 2) * y->tune_w);
        if (mpc->perform == 2)
                mpc->p_index += (fabs(y->yy - y->setpoint) * y->tune_w);
        if (mpc->perform == 3)
                mpc->p_index += (fabs(y->yy - y->setpoint) * (time*sim_param->dt) * y->tune_w);
}
```

Simulation Parameters (*sim_param*)

sim_param.simtime:	Time of simulation
sim_param.dt:	Time interval of simulation of plant models
sim_param.steptime[i]:	Time when step change is applied
sim_param.finalSP[i]:	Size of step change
sim_param.initialSP[i]:	Initial setpoint
sim_param.recordstep:	Size of interval (time) for recording the output
sim_param.max_ku:	Maximum input dead time steps
sim_param.max_kx:	Maximum state dead time steps
sim_param.max_ky:	Maximum output dead time steps
sim_param.no_inputs:	Number of inputs
sim_param.no_outputs:	Number of outputs
sim_param.accuracy:	Accuracy of simulation for steady state
sim_param.dt_power:	Time interval for power controller
sim_param.load_change:	Load change for power controller at each control interval

MPC Parameters (*mpc*)

mpc.TT[j]:	Model horizon(T) for each output
mpc.UU[i]:	Control Horizon(U) for each input
mpc.VV[j]:	Prediction horizon(V) for each output
mpc.delt:	Time interval of MPC controller
mpc.f1[j]:	Weighting factors for controlled variables (multiple of identity matrix)
mpc.f2[i]:	Weighting for manipulated variables (multiple of identity matrix)
mpc.udim:	Dimension for inputs of MPC controller
mpc.ydim:	Dimension for outputs of MPC controller
mpc.steptime[j]:	Time when step change is applied
mpc.finalSP[j]:	Final value of setpoint
mpc.initialSP[j]:	Initial value of setpoint
mpc.qpdim:	Number of controlled variables to be constrained when QP is used
mpc.qp[j].constraint:	Specifies if output should be constrained (when not controlled)
mpc.qp[j].T:	Model horizon for output that should be constrained
mpc.qp[j].V:	Control horizon for output that should be constrained
mpc.qp[j].Vb:	Beginning of constraint window
mpc.qp[j].Ve:	End of constraint window

PI-Controller Parameters (*PI*)

PI.gain:	Controller gain
PI.ti:	Integral action (ti) of PI-controller
PI.delt:	Time interval for PI-control action
PI.output:	Output number to be controlled with current input