CODEN:LUTEDX/(TEIE-7250)/1-48/(2015)

Industrial Electrical Engineering and Automation

Implementation of the Bürger-Diehl settler model on the Benchmark Simulation Platform

Magnus Arnell

Division of Industrial Electrical Engineering and Automation
Faculty of Engineering, Lund University

Implementation of the Bürger-Diehl settler model on the Benchmark Simulation Platform

MAGNUS ARNELL
IEA, Lunds University
E–mail: magnus.arnell@iea.lth.se

February 13, 2015

1. Background

Wastewater Treatment (WWT) is basically a separation problem. The incoming wastewater is contaminated by particulate and soluble substances, which are hazardous to the environment. Except for biological nitrogen removal all treatment technologies are based on the same principle – convert the contaminants to particulates and remove them as sludge. Therefore, sludge separation is a key process in WWT. Traditionally the most important sludge separation process is sedimentation. Even if different kinds of filtering techniques, such as sand filters, disc filters or even membrane filters start to become more common for final effluent polishing at wastewater treatment plants (WWTPs) sedimentation is still by far the most common separation technique. The different sedimentation steps have a great impact on the plant performance.

Mathematical process models of wastewater treatment plants have been developed since several decades and proven to be a very useful tool both for research and industrial application. The biological process models have reached an elevated level of detail describing all the important processes, such as microbial growth of different species under various conditions. These models, for example *IWA Activated Sludge Model no. 1* (ASM1) (Henze et al., 1987) are, carefully calibrated, very accurate and reliable. However, the biological model only describes one sub-process at the WWTP, and to be able to model a complete treatment process other interlinked subprocesses have to be modelled as well. To be able to model an activated sludge plant the most important subprocess is a sedimentation model for the secondary clarifier, see Figure 1. The main purpose of the secondary clarifier is solid-liquid separation giving clarification of effluent water, thickening of sludge and enabling return of activated sludge to the biological reactors. Although, some biological reactions (i.e. denitrification) can be expected to take place in the clarifier this report will only focus on the solids separation.

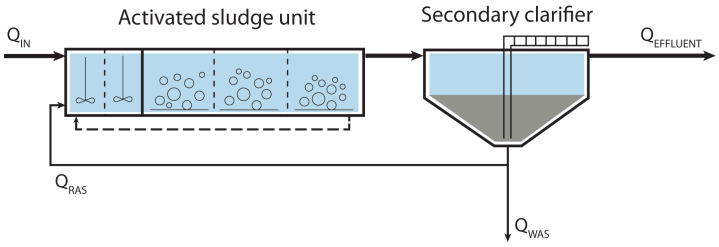

Figure 1. Process layout for an Activated Sludge Plant containing an activated sludge unit with anoxic and aerobic tanks followed by a secondary clarifier.

Sedimentation has been an important field of study through the history of WWT modelling with several models suggested and used (Makinia, 2010). One of the most common ones is the 10-layer, one dimensional model proposed by Takács et al. (1991). The model assumes uniform settling behaviour over the whole area of the settler and simplifies it to one dimension in hight. The hight is divided into 10 uniformly distributed layers. The solids flux between the layers is governed by the bulk flow, J_{bulk}, (upwards over the feed layer and downwards under the feed layer) and the gravity settling flux, J_S, determined by the settling velocity function, ν_{hs}, of choice.

$$J = J_S + J_{bulk} \tag{1}$$

The simplification to model the settler only in one dimension is justified in most applications. However, the Takács model first discretising the settler spatially in 10 fixed layers and then derive the equations directly from mass balance reasoning without considering the numerical flux in the process is shown to have several shortcomings, behave incorrectly and be numerically unstable under certain conditions. In this paper a new settling model presented by Bürger et al. (2011) (below referred to as the Bürger-Diehl model och just BD) is implemented and tested.

The test environment is the Matlab / Simulink version of the *Benchmark Simulation Model* (BSM) platform (Gernaey et al., 2014). The BSM platform is a simulation protocol for benchmarking of control strategies at WWTPs. The BSM1 describes an activated sludge plant similar to the one in Figure 1 and the BSM2 covers a full WWTP adding primary settling and sludge treatment with anaerobic digestion to the plant of BSM1. The platform consists of: *i)* a detailed plant layout for the two configurations; *ii)* a complete set of mathematical models for the included sub-processes describing physical, chemical and biological processes; *iii)* support systems such as sensors, actuators etc.; *iv)* a simulation procedure with 14 or 609 days of influent data for BSM1 and BSM2 respectively; and *v)* a performance evaluation procedure.

2. Method

2.1. The settler model

The Bürger-Diehl model is presented in a series of paper by Bürger et al. among which the latest are, Bürger et al. (2011), Bürger et al. (2012) and Bürger et al. (2013). The description in this section follows from those papers. The derivation of the model starts with the basic assumptions given above, 1-D along the depth, z, of the settler and the conservation of mass through the process. The derivation originates from the continuous version of Equation 1 and includes hindered settling, compression of sludge and dispersion due to mixing (i.e. at the inlet). This leads up to the partial differential equation (PDE):

$$\frac{\partial C}{\partial t} + \frac{\partial}{\partial z}F(C,z,t) = \frac{\partial}{\partial z}\left(\left\{\gamma(z)d_{comp}(C) + d_{disp}\big(z,Q_f(t)\big)\right\}\frac{\partial c}{\partial z}\right) + \frac{Q_f(t)C_f(t)}{A}\delta(z) \quad (2)$$

where,

C is the local TSS concentration;
t is time;
z is the depth from feed level in settler;
F is the convective flux function;
γ is the characteristic function, equals 1 inside settler and 0 outside;
d_{comp} is the compression function;
d_{disp} is the dispersion function;
Q_f is the feed flow;
C_f is the feed concentration of TSS;
A is the cross sectional area of the settler;
δ is the Dirac delta distribution.

The solution to Equation 2 may have discontinuities which means that it is not differentiable and traditional numerical methods can not be used. Rather, the weak sense of Equation 2 should be considered. For a more elaborated motivation of this and implications for the numerical solution, see Bürger et al. (2011).

The convective flux function, $F(C,z,t)$ includes the bulk flux and the hindered settling velocity function, $\nu_{hs}(C)$. Above a critical concentration, C_c, the flocs will be in constant contact and can in fact bear a solids stress leading to compression of the sludge. The compression function, d_{comp}, is a function of C and sludge density. Dispersion occurs in settling tanks due to mixing, particularly at the inlet. d_{disp} is a distribution around the inlet as

a function of Q_f. The method is flexible and allows for different choices for al of these consecutive functions, see section 2.2.

The numerical method for the PDE (2) has to include discretisation in both time (t) and space (z). For the latter the settler is divided into N layers, plus (in contrast to many other settler models) 2 extra layers at the top and bottom, respectively, representing the under- and overflow zones. These layers are needed for the discretisation scheme, but also to correctly determine effluent and underflow concentrations (Bürger et al., 2011). The spatial discretisation leads up to the finite difference form of the equation.

$$\begin{aligned}\frac{\mathrm{d}C_j}{\mathrm{d}t} = &-\frac{F\big(C(z_j,t),z_j,t\big)-F\big(C(z_{j-1},t),z_{j-1},t\big)}{\Delta z}\\ &+\frac{J_{disp}(z_j,t)-J_{disp}(z_{j-1},t)}{\Delta z}+\frac{J_{comp}(z_j,t)-J_{comp}(z_{j-1},t)}{\Delta z}\\ &+\frac{1}{\Delta z}\int_{z_{j-1}}^{z_j}\frac{Q_f(t)C_f(t)}{A}\delta(z)\mathrm{d}x \quad (3)\end{aligned}$$

Next step is to find reasonable numerical approximations of the consecutive functions to ensure that the solution converges to the exact solution. For $F(C,z,t)$ the Godunov numerical flux is chosen. The compressive and dispersive fluxes (J_{comp} and J_{disp} can be approximated with simple first-order forward approximations in space). Substituting the approximated functions into Equation 3 gives one ordinary differential equation (ODE) for each layer:

$$\begin{aligned}\frac{\mathrm{d}C_j}{\mathrm{d}t} = &-\frac{F_j^{num}-F_{j-1}^{num}}{\Delta z}+\frac{1}{\Delta z}\big(J_{disp,j}^{num}-J_{disp,j-1}^{num}+J_{comp,j}^{num}-J_{comp,j-1}^{num}\big)\\ &+\frac{Q_fC_f}{A\Delta z}\delta_{j,j_f},\quad j=-1,...,N+2 \quad (4)\end{aligned}$$

This is called a Method Of Lines (MOL) discretisation of the PDE. To assure stability of the numerical scheme, using MOL, it is important to consider the so called CFL condition, Equation 5. This states the largest allowed integration step size for the ODE solver based on the chosen layer thickness (Δz) in the settler.

$$\begin{aligned}\Delta t \le &\left[\frac{1}{\Delta z}\Big(\max_{0\le t\le T}\frac{Q_f(t)}{A}+\max_{0\le t\le T}\mid f_{bk}'(C)\mid\Big)\right.\\ &\left.+\frac{2}{(\Delta z)^2}\Big(\max_{0\le C\le C_{max}}d_{comp}(C)+\max_{-H\le z\le B,0\le t\le T}d_{disp}\big(z,Q_f(t)\big)\Big)\right]^{-1} \quad (5)\end{aligned}$$

2.2. Consecutive functions for settling velocity, compression and dispersion

The BD-model is flexible and allow for alternative functions for the different phenomena included, i.e. settling velocity, compression and dispersion. Different alternatives have been proposed for all three functions as described in this section. For the implmentation presented in this report the consecutive functions of Bürger et al. (2013) with corresponding parameter values have been used.

For the settling velocity both Vesilind (Eqn. 6) and Takács (Eqn. 7) settling velocity functions were tested during implementation. In the following description the Takács function is used to allow comparison with the standard BSM settler model.

$$\nu_{hs} = \nu_0 e^{-r_V C} \tag{6}$$

$$\nu_{hs} = \max\left\{0, \min\left[\nu_0', \nu_0(e^{-r_h(C-C_{min})} - e^{-r_p(C-C_{min})})\right]\right\} \tag{7}$$

where,

ν_{hs} is the hindered settling velocity;
r_V parameter in Vesilind settling velocity function;
ν_0 is the free settling velocity;
ν_0' is the maximum free settling velocity;
r_h parameter associated with hindered settling;
r_p parameter associated with low conc. and slow settling components;
C is the solids concentration;
C_{min} is the minimum solids concentration in the effluent.

For the compression and dispersion Bürger et al. (2013) proposed Equations 8 and 9 below with the parameter values given in Table 1.

$$d_{comp}(C) = \begin{cases} 0 & \text{for } C < C_c, \\ \frac{\rho_s \gamma \nu_{hs}}{g(\rho_s - \rho_f)(\beta + C - C_c)} & \text{for } C \geq C_c. \end{cases} \tag{8}$$

$$d_{disp} = \begin{cases} \alpha_1 Q_f \exp\left(\frac{-z^2/(\alpha_2 Q_f)^2}{1-|z|/(\alpha_2 Q_f)}\right) & \text{for } |z| < \alpha_2 Q_f, \\ 0 & \text{for } |z| \geq \alpha_2 Q_f. \end{cases} \tag{9}$$

where,

z	depth from feed level;
ρ_s, ρ_f	densities of sludge and fluid;
γ, β	parameters in solid stress function;
C_c	critical concentration;
α_1	parameter in dispersion function;
α_2	parameter in dispersion function, chosen so that the dispersion is zero outside the settler at all times.

Slightly simplified functions, Equations 10 and 11, are used in the implementation of the model in the commercial simulation package WEST by DHI.

$$d_{comp} = \begin{cases} 0 & \text{if } 0 \leq C < C_c, \\ \frac{\rho_s \gamma \nu_{hs}(C)}{g(\rho_s - \rho_f)} & \text{if } \geq C_c. \end{cases} \tag{10}$$

$$d_{disp} = \begin{cases} \frac{a}{A} Q_f \cos^2\left(\frac{\pi Z}{2bQ_f}\right) & \text{if } |z| < bQ_f, \\ 0 & \text{if } |z| \geq bQ_f. \end{cases} \tag{11}$$

2.3. Implementation in BSM

The implementation of the Bürger-Diehl model in the BSM platform follows the very detailed description in Bürger et al. (2013). The choice of consecutive functions, such as d_{comp}, d_{disp}, etc., also follows Bürger et al. (2013) with exception for ν_{hs} as described above. The interested reader is encouraged to read that paper since these details will not be repeated here. Furthermore, the Matlab code builds on the original Takács settler model in the BSM keeping the different options for handling of solubles and temperature. A transcript of the code is given in the Appendices.

INITIALISATION AND PRECOMPUTATION

For solving the ODE:s the solver needs initial values for all states and defined values for all the parameter values in the model. For this purpose a script is run once before the start of the simulation. In the case of the BD model the initialisation process is also used to perform some precomputations in order to speed up the simulation. The script code is given in Appendices A and C.

To allow for the Godunov flux to be easily computed with algorithm 1 (Bürger et al., 2013) it is important to find the concentration $\hat{C}$ at which the batch flux density function 12 is at its local maximum. For 6 this is easily identified as $-1/r_V$. However for Eqn. 7 we will have to derive the equation obtained by substituting Eqn. 7 into Eqn. 12 getting Eqn. 13 and search for the upper zero derivative over a reasonably large range of possible C.

$$f_{bk}(C) = C\nu_{hs}(C) \tag{12}$$

$$f'_{bk} = \nu_0 \left(\frac{1}{e^{r_h(C-C_{min})}} - \frac{1}{e^{r_p(C-C_{min})}} \right) - C\nu_0 \left(\frac{r_h}{e^{r_h(C-C_{min})}} - \frac{r_p}{e^{r_p(C-C_{min})}} \right) \quad (13)$$

To speed up the simulations the tedious solving of the integral in d_{comp} can also be numerically approximated according to algorithm 2 in Bürger et al. (2013). This is done over a fine grid and sufficiently large interval of C. Furthermore, the CFL condition according to Eqn. 5 is computed in the initialisation script. For the calculation the maximum flow over the settler (Q_f) has to be known or guessed and the maximum f'_{bk}, d_{comp} and d_{disp} have to be found within the actual range of C, z and t respectively. For these calculations C_{max} has been set to 20 000 g.m^{-3} and Q_f to 200 000 m^3.d^{-1}.

The used parameter values for the consecutive functions are given in Table 1.

Table 1. Parameter values in consecutive functions Equation. 7 above and Equations. 13 and 14 in Bürger et al. (2013)

Settl. velocity	Value	Comp.	Value	Disp.	Value
v'_0 [m.d^{-1}]	250	γ [g.m^{-1}.d^{-2}]	$2.986e^{13}$	α_1 [m^{-1}]	0.0023
v_0 [m.d^{-1}]	474	β [g.m^{-3}]	4000	α_2 [d.m^{-2}]	$5e^{-6}$
r_h [m^3.g^{-1}]	0.000576	C_c [g.m^{-3}]	4000		
r_p [m^3.g^{-1}]	0.00286				
C_{min} [g.m^{-3}]	9				

ODE IMPLEMENTATION

As explained in section 1 and 2, one of the advantages of the Bürger-Diehl model over the original Takács model is that the correct numerical approximation of the batch flux allows for a finer spatial grid, i.e. more layers than 10, to be used. This increases the accuracy of the solution and also improves the estimation of the sludge blanket level in the settler. To include this feature in the BSM implementation of the BD model the S-function has been build with a variable number of layers. The discretisation of the settler in space is done in the initialisation script and sent to the S-function. This makes it possible to run faster simulations with a corse grid or to increase the number of layers if improved accuracy is needed. The later strategy

will also lead to slower simulations due to the lower Δt according to the CFL condition.

SUITABLE ODE SOLVERS

To solve the scheme of ODEs resulting from the MOL discretisation of space (Eqn. 4) is it necessary to apply some discretisation scheme in time and solve the resulting approximative difference equations, i.e. using an "ODE-solver". Since the spatial discretisation is only first-order accurate in space it is actually sufficient with a first-order accurate solver like Euler. However, due to use of the BD model within a WWTP model on commercial simulation platforms it is often not feasible to use fixed-step solvers like Euler. For practical matters and accuracy during long dynamic simulations variable step solvers are needed like the Runge-Kutta (RK) method, e.g. Matlab–ode45 RK(4,5) Dormand-Prince method of order 4, and Matlab–ode23, which is the RK(2,3) Bogacki-Shampine method of order 2. Diehl et al. (2014) investigated the impact of different solvers on over-all accuracy and simulation speed showing that the ode45 and ode23 are not superior choices but well functioning if the solutions do not contain discontinuities. The best solver was a specific semi-implicit method. However, it is not included in the Matlab-Simulink implementation used in this paper.

The simulations presented below were run on a PC with an AMD FX(tm)-8350 Eight-core processor at 4.00 GHz with 8 GB RAM running 64-bit Matlab-Simulink R2014a.

3. Simulation studies

3.1. BSM1

A couple of scenarios have been simulated to test the BSM1-BD model. Figures 2 and 3 show results from running the BSM1-BD with the dry and storm influent. As can be seen from Figure 2 the settler is in both cases low loaded and even the heavy storm events around days 9 and 11 only give a minor rise in the sludge blanket. However, from Figure 3 we can see that it results in a shorter rise in both effluent and underflow TSS. The increased effluent TSS gives an unintended sludge loss from the system.

The simulations are run with 30 internal layers, which gives a maximum step size of $\Delta t < 1.78e^{-5}$ according to the CFL condition. Reducing the number of layers to 10 gives $\Delta t < 1.43e^{-4}$, which is significantly larger and in fact larger than what the ODE:s in the bio-reactors can tolerate. In practice this means that a variable step solver like ode23 or ode45 will reduce the step size below the CFL-limit from time to time during the simulation. However, for 30 layers the CFL condition is the limiting factor for

the integrartion step size. The simulations still run reasonably fast, some simulation times are listed in Table 2. For BSM1 it can be seen that even if the simulation time increases many fold when the number of layers is increased to 30 or 100 it is still within reasonable limits for practical purposes. The ode23 is faster than the ode45 in all cases but the difference matters for practical use first at 100 layers.

Table 2. Simulation times for the different configurations. Open Loop (OL), Closed Loop (CL).

Model / solver	**N=10**	**N=30**	**N=100**
BSM1 CL / Euler1	8s	48s	33m:54s
BSM1 CL / ode23	17s	2m:32s	49m:55s
BSM1 CL / ode45	30s	4m:27s	1h:24m:19s
BSM2 OL / ode23	18m:33s	2h:36m:22s	40h:10m:49s
BSM2 OL / ode45	21m:04s	4h:34m:32s	
BSM2 CL / ode23	2h:10m:23s	3h:58m:49s	

3.2. BSM2

The BSM2-BD version has been tested in open loop with full 609 days simulations of which the last 365 have been used for evaluation (Gernaey et al., 2014). For comparison the standard BSM2 with the original Takács settler model has been simulated. Results can be seen in Figures 4 and 5. The first observation is that in comparison with BSM1-BD the BSM2-BD settler is much more highly loaded, which shows in higher sludge levels and higher effluent TSS during high flow. This is not surprising since the settler has the same dimensions but the plant load is about doubled. In Figure 5 the BD settler model is compared with an identical simulation using the Takács model. There are several differences in the results; the sludge blanket is generally higher, the effluent TSS peaks are significantly higher and the bottom TSS is somewhat lower – even if this differs depending on the number of layers used with the BD-model. These differences are mostly because compression is included in the BD model but also the dispersion contributes.

The simulation times for a few combinations of solvers and number of layers are given in Table 2. Simulating a full BSM2-BD with 10 layers does not give any significantly longer simulation times compared to the default Takács version. However, increasing the number of layers to 30 dramatically increases the simulation time as the temporal step-size is reduced, but for practical simulations a couple of hours is still reasonable if the increased accuracy is asked for. For the closed loop case the difference would

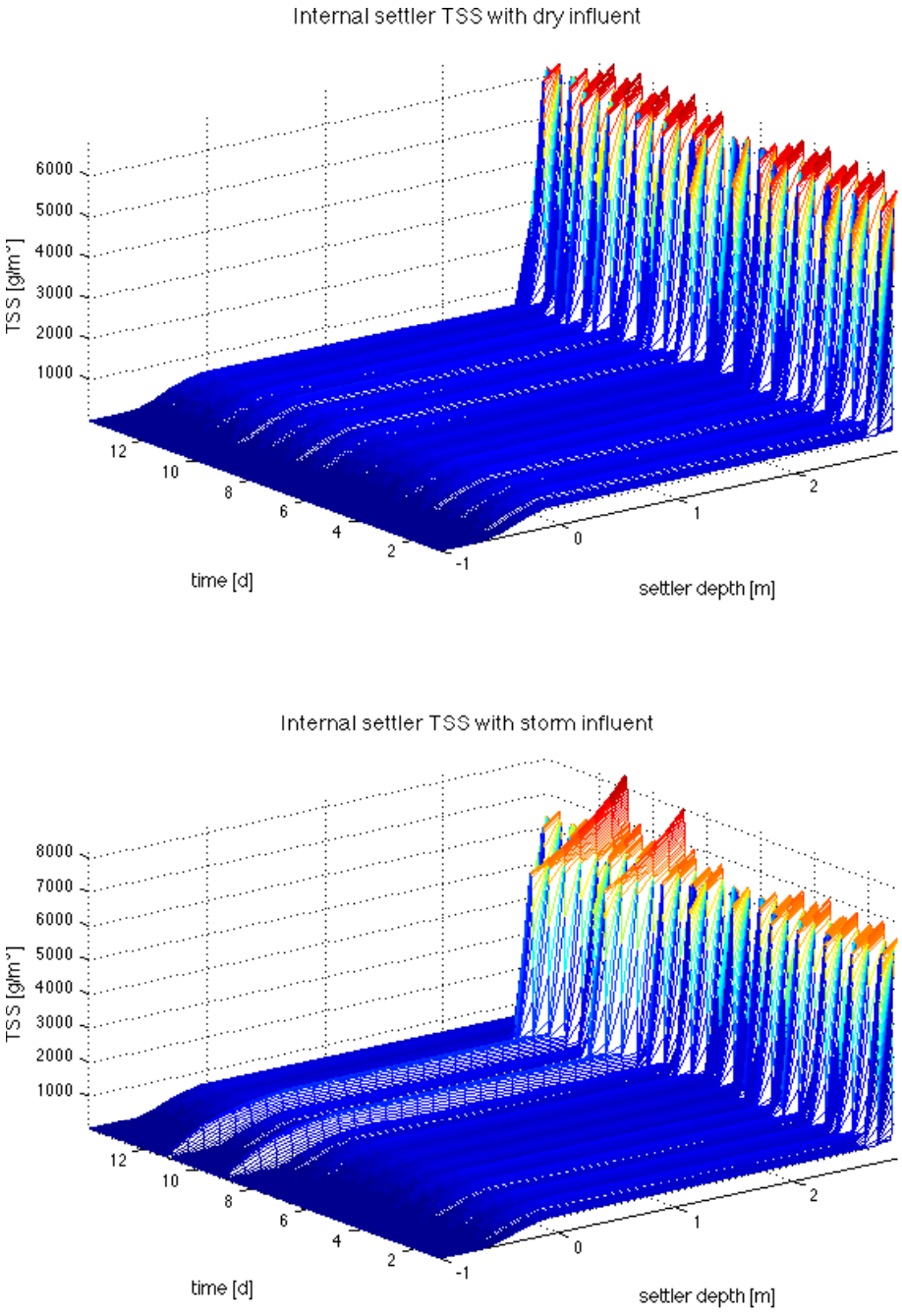

Figure 2. TSS concentrations over the hight of the settler for the duration of the simulation. Dry weather influent (top), storm weather influent (bottom). Simulation of BSM1 with 30 internal layers.

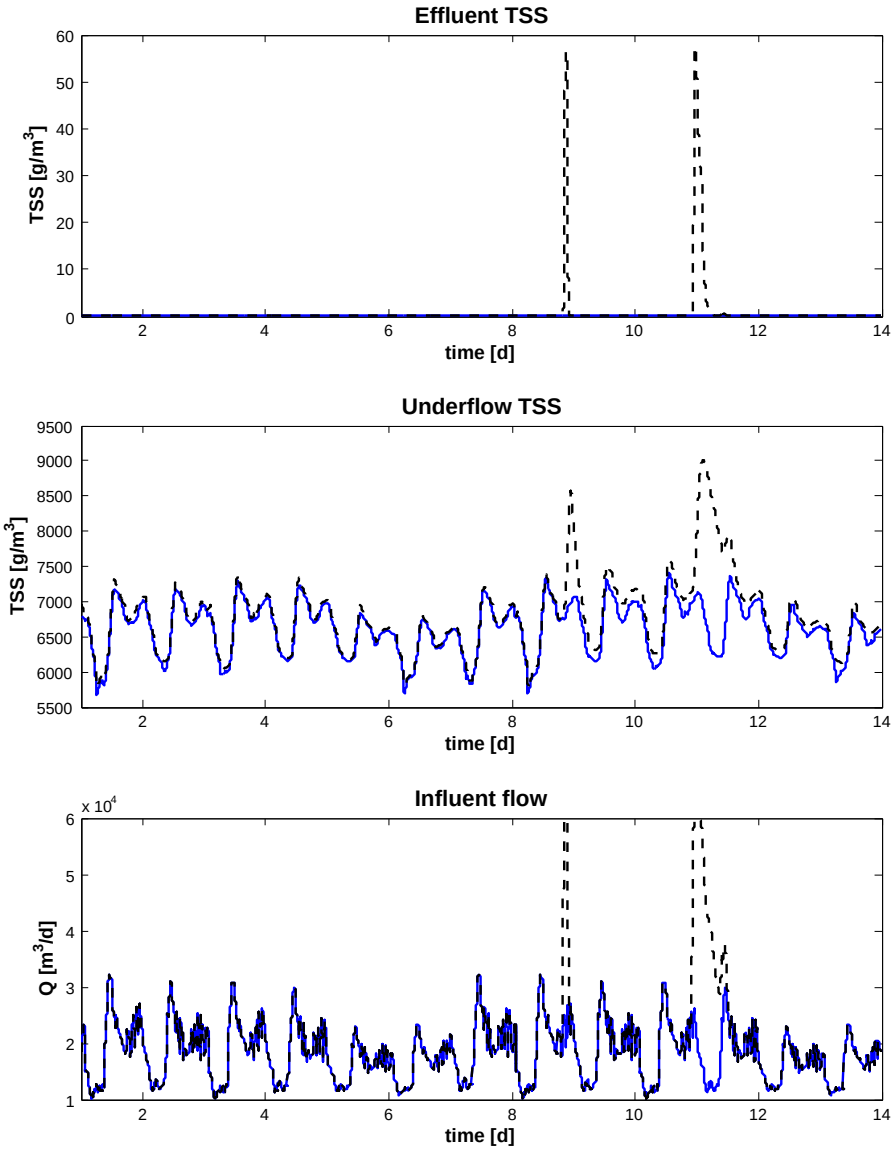

Figure 3. Settler performance during dry (solid blue line) and storm (dashed black line) weather conditions. Settler TSS in effluent (top), settler TSS in underflow (middle) and plant influent flow (bottom). Simulation of BSM1 with 30 internal layers.

be smaller since sensor noise and control loops will force the solver to take shorter integration steps regardless of the CFL-condition. The ode23 is as expected faster than the ode45 in all cases but the difference is notable first at 30 or more layers.

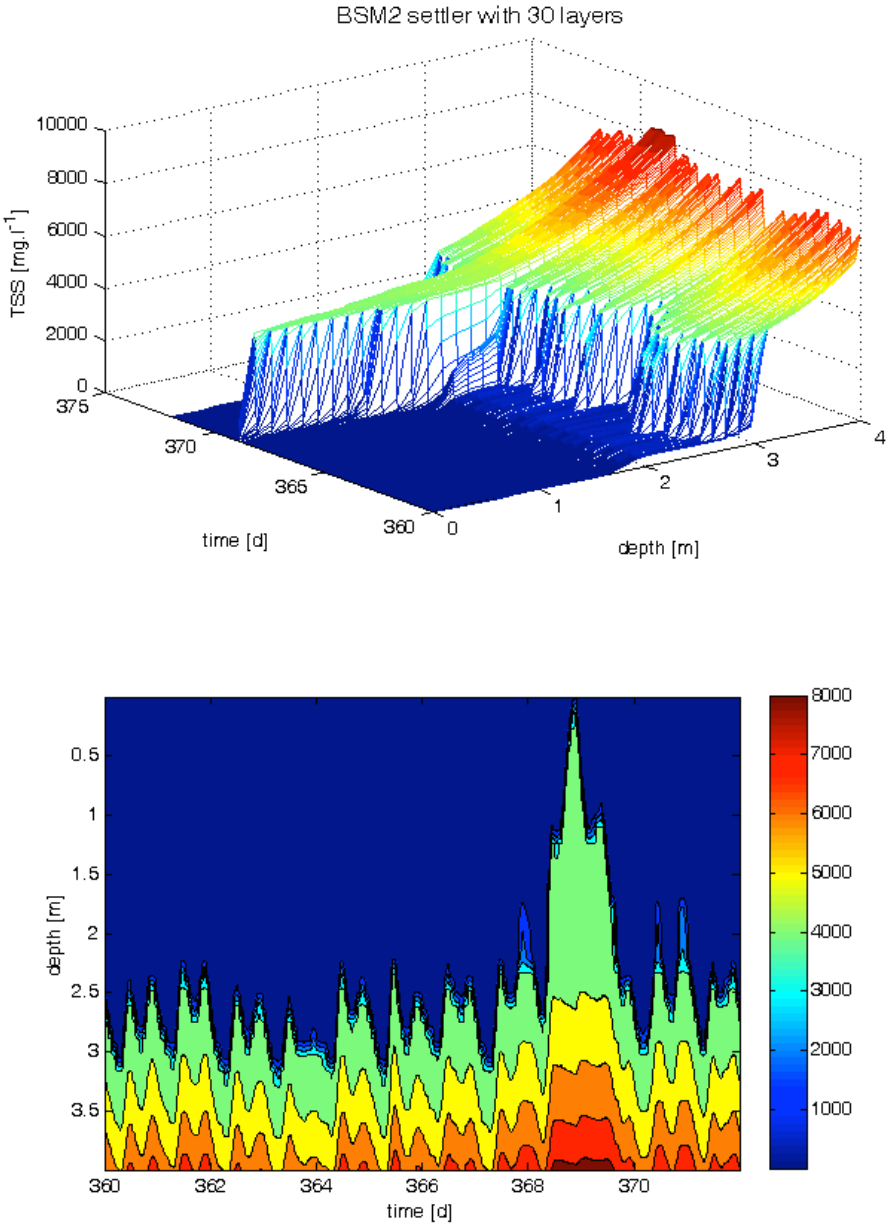

Figure 4. TSS concentrations over the hight of the settler for 7 days of simulation, a heavy rain event occurs during day 368. 3D plot (top), contour plot (bottom). Simulation of BSM2 with 30 internal layers.

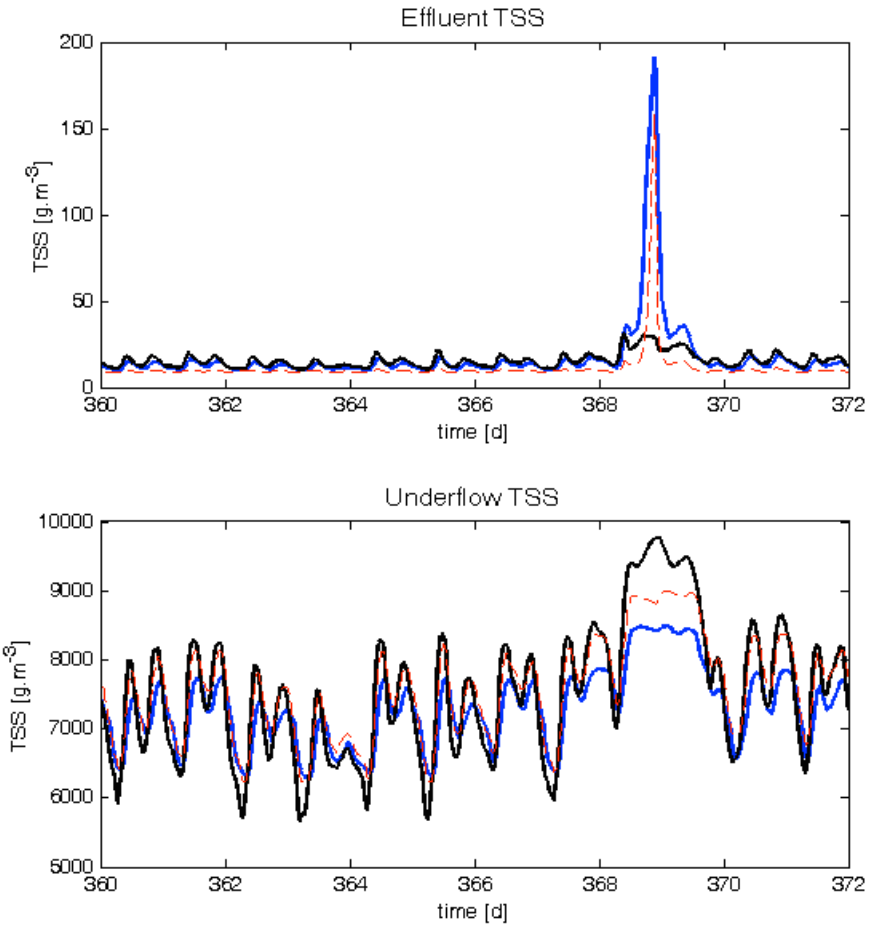

Figure 5. Compatarive plots for effluent TSS (top) and underflow TSS (bottom). Takács model (black solid line), BD model 10 layers (blue solid line) and BD model with 30 layers (red dashed line). Simulation of BSM2.

4. Acknowledgements

The author acknowledge the financial support for this project from Swedish Water and Wastewater Association (contract nr. 211-2010-141), Formas (contract nr 10-106) and Vinnova (project 2012-01243). The work has been done in close collaboration with Dr. Stefan Diehl and Sebastian Farås at Centre for Mathematical Sciences, Lund University and the author is grateful for all the support and help with the implementation.

References

Bürger, R., Diehl, S. and Nopens, I. (2011). A consistent modelling methodology for secondary settling tanks in wastewater treatment. Water Res., 45, 2247–2260.

Bürger, R., Diehl, S., Farås, S. and Nopens, I. (2012). On reliable and unreliable numerical methods for the simulation of secondary settling tanks in wastewater treatment. Comput. Chem. Eng., 41, 93–105.

Bürger, R., Diehl, S., Farås, S., Nopens, I. and Torfs, E. (2013). A consistent modelling methodology for secondary settling tanks: a reliable numerical method. Water Sci Technol., 68(1), 192–208.

Diehl, S., Farås, S. and Mauritsson, G. (2014). Fast reliable simulations of secondary settling tanks in wastewater treatment with semi-implicit time discretization. *Submitted to Elsevier*.

Gernaey, K.V., Jeppsson, U., Vanrolleghem, P.A. and Copp, J.B. (2014). Benchmarking of Control Strategies for Wastewater Treatment Plants. Scientific and Technical Report No. 23. IWA Publishing, London, UK.

Henze, M., Grady, Jr C.P.L., Gujer, W., Marais, G.v.R. and Matsuo, T. (1987). Activated Sludge Model n° 1. IAWQ Scientific and Technical Report n°1, IAWQ, London, UK.

Makinia, J. (2010). Mathematical Modelling and Computer Simulation of Activated Sludge systems. IWA Publishing, London, UK.

Takács, I., Patry, G.G. and Nolasco, D. (1991). A dynamic model of the clarification-thickening process. Water Res., 25(10), 1263–1271.

A. BSM1 Bürger-Diehl settler model initialisation m-script

```
% Initialisation file for all parameters and states in the B¸rger-Diehl model
% for secondary clarifiers. The Tak·cs settling velocity function is used.
%
% The model is implemented with variable number of layers.
%     set 'n' equal to the desired number of layers (10 minimum)
%
% Copyright: Magnus Arnell, IEA, Lund University, Sweden. 2014-12-01.


%% PARAMETERS
% Parameters in settling velocity (batch flux) %
v0_max = 250;
v0 = 474;
r_h = 0.000576;
r_p = 0.00286;
f_ns = 0.00228;
Cmin  = f_ns.*4000;                 % TSS cons below which settling velocity is zero

% Extra parameters in compression %
k    = 4*1e3;                       % parameter in solid stress [g/m^3]
s0   = (3600*24)^2*4*1e3;           % parameter in solid stress [g/(m*d^2)]
rhos = 1050*1e3;                    % Solids density        [g/m^3]
drho = rhos-998*1e3;                % Density difference    [g/m^3]
g    = 9.81*(3600*24)^2;            % Acceleration constant [m/d^2]


% Parameters in dispersion %
alph1 = 0.0023;                     % parameter in dispersion [1/m]
alph2 = 1/200000;                   % parameter in dispersion [d/m^2]


% Division into layers %
n = 100;                             % Number of layers in settler
N = n+4;                            % Number of computational cells in model
B = 2;                              % Hight of thickening zone
H = 2;                              % Hight of clarifications zone
dz = (B+H)/n;                       % Layer depth
feedlayer = ceil(H/dz);             % Index of feed layer
z  = linspace(-H-2*dz,B+2*dz,n+5);  % Layer boundaries
area = 1500;                        % Cross sectional area of settner
height = B+H;                       % Total hieght of settler
Z=linspace(-2*dz,height+2*dz,N);    % Z-variable for mesh of settler

%% PRECOMPUTATION
% Precomputation of Dtilde
Ccrit = 4000;
Cmax  = 20000;
M = n^2;
dC = (Cmax - Ccrit)/(M-1);
Dtilde(1) = 0;
d(1) = rhos*s0*max(0,min(v0_max,v0*(exp(-r_h.*(Ccrit-Cmin))...
    -exp(-r_p.*(Ccrit-Cmin)))))/(g*drho*(Ccrit-Ccrit+k));
```

```
for i=2:M
    d(i) = rhos*s0*max(0,min(v0_max,v0*(exp(-r_h.*((Ccrit+(i-1)*dC)-Cmin))...
        -exp(-r_p.*((Ccrit+(i-1)*dC)-Cmin)))))/(g*drho*((Ccrit+(i-1)*dC)-Ccrit+k));
    Dtilde(i) = Dtilde(i-1)+dC/2*(d(i-1)+d(i));
end


% Find local maximum of fbk, Chat
f1 = @(x)v0.*(1./exp(r_h.*(x - Cmin)) - 1./exp(r_p.*(x - Cmin))) ...
    - x.*v0.*(r_h./exp(r_h.*(x - Cmin)) - r_p./exp(r_p.*(x - Cmin)));
f2 = @(x)v0_max.*x - x.*(v0*(exp(-r_h.*(x-Cmin))-exp(-r_p.*(x-Cmin))));
Chat1 = fzero(f1,2000);             % Find zero of fbk_prime
Chat2 = fzero(f2,Cmax);             % Find upper zero of v0*C - fbk(C)
Chat = max(Chat1,Chat2);


% Calculate the CFL-condition %
C = 0:0.01:Cmax;
fbkPrime = v0.*(1./exp(r_h.*(C - Cmin)) - 1./exp(r_p.*(C - Cmin))) ...
    - C.*v0.*(r_h./exp(r_h.*(C - Cmin)) - r_p./exp(r_p.*(C - Cmin))); %derivative of fbk
fbkPrime_max = max(abs(fbkPrime)); % find maximum of fbkPrime
dcomp_max    = rhos*max(0,min(v0_max,v0*(exp(-r_h.*(Ccrit-f_ns.*4000))...
    -exp(-r_p.*(Ccrit-f_ns.*4000)))))/(g*drho*(Ccrit-Ccrit+k));
Qf_max       = 200000;
X = -z.^2./(alph2*Qf_max-abs(z));
ddisp_max    = max(alph1*Qf_max.*exp(X).*(abs(z)<alph2*Qf_max));

CFL = 0.98*dz^2/(dz*(fbkPrime_max+Qf_max/area)+2*(dcomp_max+ddisp_max));


%% INPUTVECTORS FOR S-FUNCTION
SETTLERPAR = [ alph1 alph2 v0_max v0 r_h r_p f_ns Chat Ccrit dC ];

DIM = [ area height ];

LAYER = [ feedlayer n dz M ];

DTILDE = Dtilde;

LAYERBOUNDARIES = z;

%% INITIALIZATION OF STATES %%%
% Initialization of TSS concentrations
zm = z(1:end-1)+dz/2;
SETTLERINIT = 7000*(zm>1.5);

% INITIALIZATION OF SI STATES %
for i=(N+1):(2*N)
    SETTLERINIT(i) = 30;
end

% INITIALIZATION OF SS STATES %
for i=(2*N+1):(3*N)
    SETTLERINIT(i) = 0.80801;
```

```
end

% INITIALIZATION OF SO STATES %
for i=(3*N+1):(4*N)
    SETTLERINIT(i) = 2;
end

% INITIALIZATION OF SNO STATES %
for i=(4*N+1):(5*N)
    SETTLERINIT(i) = 13.5243;
end

% INITIALIZATION OF SNH STATES %
for i=(5*N+1):(6*N)
    SETTLERINIT(i) = 0.67193;
end

% INITIALIZATION OFF SND STATES %
for i=(6*N+1):(7*N)
    SETTLERINIT(i) = 0.6645;
end

% INITIALIZATION OFF SALK STATES %
for i=(7*N+1):(8*N)
    SETTLERINIT(i) = 3.8277;
end

%% SET MODELTYPE
% to use model with 10 layers for solubles use type 0 (COST Benchmark)
% to use model with 1 layer for solubles use type 1 (GSP-X implementation)
% to use model with 0 layers for solubles use type 2 (WEST implementation)

MODELTYPE = [ 0 ];
```

B. BSM1 Bürger-Diehl settler model s-function c-file

```
/*
 * SETTLER1DVBD is a C-file S-function for defining the B¸rger-Diehl
 * settler model. This implementation allows for a variable number of layers.
 * Can simulate 0, 1 or N layers for the solubles by using MODELTYPE
 *
 * Copyright: Magnus Arnell, IEA, Lund University, Lund, Sweden. 2014-12-01
 *
 * Note that wraped lines in this text is only for legibility.
 */

#define S_FUNCTION_NAME settler1dvBD

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

#define XINIT           ssGetArg(S,0)
#define PAR             ssGetArg(S,1)
#define DIM             ssGetArg(S,2)
#define LAYER           ssGetArg(S,3)
#define MODELTYPE       ssGetArg(S,4)
#define LAYERBOUNDARIES ssGetArg(S,5)
#define DTILDE          ssGetArg(S,6)

#define mymax(a,b) (a>=b ? a:b)
#define mymin(a,b) (a<=b ? a:b)


/*
 * mdlInitializeSizes - initialize the sizes array
 */
static void mdlInitializeSizes(SimStruct *S)
{
int nStates, nOutputs;

    int N        = (int)mxGetPr(LAYER)[1]+4;
    nStates  = 8*N;
    nOutputs = 8*N+33;


    ssSetNumContStates(    S, nStates);     /* number of continuous states           */
    ssSetNumDiscStates(    S, 0);           /* number of discrete states             */
    ssSetNumInputs(        S, 17);          /* number of inputs                      */
    ssSetNumOutputs(       S, nOutputs);    /* number of outputs                     */
    ssSetDirectFeedThrough(S, 1);           /* direct feedthrough flag               */
    ssSetNumSampleTimes(   S, 1);           /* number of sample times                */
    ssSetNumSFcnParams(    S, 7);           /* number of input arguments             */
    ssSetNumRWork(         S, 0);           /* number of real work vector ...
     * elements   */
    ssSetNumIWork(         S, 0);           /* number of integer work vector ...
     * elements*/
    ssSetNumPWork(         S, 0);           /* number of pointer work vector ...
     * elements*/
```

```
}

/*
 * mdlInitializeSampleTimes - initialize the sample times array
 */
static void mdlInitializeSampleTimes(SimStruct *S)
{
    ssSetSampleTime(S, 0, CONTINUOUS_SAMPLE_TIME);
    ssSetOffsetTime(S, 0, 0.0);
}


/*
 * mdlInitializeConditions - initialize the states
 */
static void mdlInitializeConditions(double *x0, SimStruct *S)
{
    int i;

    int N        = (int)mxGetPr(LAYER)[1]+4;

    for (i = 0; i < 8*N; i++) {
        x0[i] = mxGetPr(XINIT)[i];
    }

}

/*
 * mdlOutputs - compute the outputs
 */

static void mdlOutputs(double *y, double *x, double *u, SimStruct *S, int tid)
{
  double gamma, gamma_eff, modeltype;
  int i;

  int N        = (int)mxGetPr(LAYER)[1]+4;

  gamma = x[N-1]/u[13];
  gamma_eff = x[0]/u[13];

  modeltype = mxGetPr(MODELTYPE)[0];

  if (modeltype < 0.5) {
     /* underflow */
     y[0]=x[N+N-1];
     y[1]=x[2*N+N-1];
     y[2]=u[2]*gamma;
     y[3]=u[3]*gamma;
     y[4]=u[4]*gamma;
     y[5]=u[5]*gamma;
     y[6]=u[6]*gamma;
     y[7]=x[3*N+N-1]; /* use oxygen in return sludge flow */
     y[8]=x[4*N+N-1];
```

```
    y[9]=x[5*N+N-1];
    y[10]=x[6*N+N-1];
    y[11]=u[11]*gamma;
    y[12]=x[7*N+N-1];
    y[13]=x[N-1];
    y[14]=u[15];  /* Q_r */
    y[15]=u[16];  /* Q_w */

    /* effluent */
    y[16]=x[N];
    y[17]=x[2*N];
    y[18]=u[2]*gamma_eff;
    y[19]=u[3]*gamma_eff;
    y[20]=u[4]*gamma_eff;
    y[21]=u[5]*gamma_eff;
    y[22]=u[6]*gamma_eff;
    y[23]=x[3*N]; /* use oxygen in effluent flow */
    y[24]=x[4*N];
    y[25]=x[5*N];
    y[26]=x[6*N];
    y[27]=u[11]*gamma_eff;
    y[28]=x[7*N];
    y[29]=x[0];
    y[30]=u[14]-u[15]-u[16];  /* Q_e */

    /* internal TSS states */
    for (i = 0; i < N; i++) {
        y[i+31] = x[i];
    }

    y[N+31]=gamma;
    y[N+32]=gamma_eff;

    for (i = N; i < (8*N); i++)
       y[i+33] = x[i];
}

else if ((modeltype > 0.5) && (modeltype < 1.5)) {
    /* underflow */
    y[0]=x[N];
    y[1]=x[2*N];
    y[2]=u[2]*gamma;
    y[3]=u[3]*gamma;
    y[4]=u[4]*gamma;
    y[5]=u[5]*gamma;
    y[6]=u[6]*gamma;
    y[7]=x[3*N]; /* use oxygen in return sludge flow */
    y[8]=x[4*N];
    y[9]=x[5*N];
    y[10]=x[6*N];
    y[11]=u[11]*gamma;
    y[12]=x[7*N];
    y[13]=x[N-1];
    y[14]=u[15];  /* Q_r */
```

```
    y[15]=u[16];  /* Q_w */

    /* effluent */
    y[16]=x[N];
    y[17]=x[2*N];
    y[18]=u[2]*gamma_eff;
    y[19]=u[3]*gamma_eff;
    y[20]=u[4]*gamma_eff;
    y[21]=u[5]*gamma_eff;
    y[22]=u[6]*gamma_eff;
    y[23]=x[3*N]; /* use oxygen in effluent flow */
    y[24]=x[4*N];
    y[25]=x[5*N];
    y[26]=x[6*N];
    y[27]=u[11]*gamma_eff;
    y[28]=x[7*N];
    y[29]=x[0];
    y[30]=u[14]-u[15]-u[16];  /* Q_e */

    /* internal TSS states */
    for (i = 0; i < N; i++) {
        y[i+31] = x[i];
    }

    y[N+31]=gamma;
    y[N+32]=gamma_eff;

    for (i = N; i < (2*N); i++)
       y[i+33] = x[N];
    for (i = (2*N); i < (3*N); i++)
       y[i+33] = x[2*N];
    for (i = (3*N); i < (4*N); i++)
       y[i+33] = x[3*N];
    for (i = (4*N); i < (5*N); i++)
       y[i+33] = x[4*N];
    for (i = (5*N); i < (6*N); i++)
       y[i+33] = x[5*N];
    for (i = (6*N); i < (7*N); i++)
       y[i+33] = x[6*N];
    for (i = (7*N); i < (8*N); i++)
       y[i+33] = x[7*N];
 }

 else if (modeltype > 1.5) {
    /* underflow */
    y[0]=u[0];
    y[1]=u[1];
    y[2]=u[2]*gamma;
    y[3]=u[3]*gamma;
    y[4]=u[4]*gamma;
    y[5]=u[5]*gamma;
    y[6]=u[6]*gamma;
    y[7]=u[7]; /* use oxygen in return sludge flow */
    y[8]=u[8];
```

```
    y[9]=u[9];
    y[10]=u[10];
    y[11]=u[11]*gamma;
    y[12]=u[12];
    y[13]=x[N-1];
    y[14]=u[15];  /* Q_r */
    y[15]=u[16];  /* Q_w */

    /* effluent */
    y[16]=u[0];
    y[17]=u[1];
    y[18]=u[2]*gamma_eff;
    y[19]=u[3]*gamma_eff;
    y[20]=u[4]*gamma_eff;
    y[21]=u[5]*gamma_eff;
    y[22]=u[6]*gamma_eff;
    y[23]=u[7]; /* use oxygen in effluent flow */
    y[24]=u[8];
    y[25]=u[9];
    y[26]=u[10];
    y[27]=u[11]*gamma_eff;
    y[28]=u[12];
    y[29]=x[0];
    y[30]=u[14]-u[15]-u[16];  /* Q_e */

    /* internal TSS states */
    for (i = 0; i < N; i++) {
        y[i+31] = x[i];
    }

    y[N+31]=gamma;
    y[N+32]=gamma_eff;

    for (i = N; i < (2*N); i++)
       y[i+33] = u[0];
    for (i = (2*N); i < (3*N); i++)
       y[i+33] = u[1];
    for (i = (3*N); i < (4*N); i++)
       y[i+33] = u[7];
    for (i = (4*N); i < (5*N); i++)
       y[i+33] = u[8];
    for (i = (5*N); i < (6*N); i++)
       y[i+33] = u[9];
    for (i = (6*N); i < (7*N); i++)
       y[i+33] = u[10];
    for (i = (7*N); i < (8*N); i++)
       y[i+33] = u[12];
  }
}

/*
 * mdlUpdate - perform action at major integration time step
 */
```

```
static void mdlUpdate(double *x, double *u, SimStruct *S, int tid)
{
}

/*
 * mdlDerivatives - compute the derivatives
 */
static void mdlDerivatives(double *dx, double *x, double *u, SimStruct *S, int tid)
{

double area, feedlayer, dz, alph1, alph2, v0_max, v0, r_h, r_p, f_ns, Chat, fbkhat;
double Q_f, Q_e, Q_u, C_f, eps, Ccrit, dC, v_up, v_dn, v_in, h, volume, modeltype;
int i, k;

int N  = (int)mxGetPr(LAYER)[1]+4;
int M  = (int)mxGetPr(LAYER)[3];

double *dd       = mxCalloc(N,sizeof(double));
double *Dnum     = mxCalloc(N,sizeof(double));
double *fbk      = mxCalloc(N,sizeof(double));
double *G        = mxCalloc(N,sizeof(double));
double *z        = mxCalloc(N,sizeof(double));
double *Dtilde   = mxCalloc(M,sizeof(double));


/* Get inputs */
area      = mxGetPr(DIM)[0];
feedlayer = mxGetPr(LAYER)[0]+1;

dz        = mxGetPr(LAYER)[2];

for (i=0; i<N; i++) {
    z[i] = mxGetPr(LAYERBOUNDARIES)[i+1];
}

for (i=0; i<M; i++) {
    Dtilde[i] = mxGetPr(DTILDE)[i];
}

eps = 0.01;
Q_f = u[14];
Q_u = u[15] + u[16];
Q_e = Q_f - Q_u;
C_f = u[13];

alph1  = mxGetPr(PAR)[0];
alph2  = mxGetPr(PAR)[1];
v0_max = mxGetPr(PAR)[2];
v0     = mxGetPr(PAR)[3];
r_h    = mxGetPr(PAR)[4];
r_p    = mxGetPr(PAR)[5];
f_ns   = mxGetPr(PAR)[6];
Chat   = mxGetPr(PAR)[7];
Ccrit  = mxGetPr(PAR)[8];
```

```
dC     = mxGetPr(PAR)[9];


h = mxGetPr(DIM)[1]/mxGetPr(LAYER)[1];
modeltype = mxGetPr(MODELTYPE)[0];
volume = area*mxGetPr(DIM)[1];
v_in = Q_f/area;
v_up = Q_e/area;
v_dn = Q_u/area;

/* Dispersion*/
dd[0] = 0.0;
dd[N-1] = 0.0;
for (i=1; i<N-1 ; i++) {
    dd[i] = fabs(z[i])<alph2*Q_f ? alph1*Q_f*exp((-1.0*pow(z[i],2.0)...
            /pow((alph2*Q_f),2.0))/(1.0-fabs(z[i])/(alph2*Q_f))):0.0;
}



/* Compression */
Dnum[0] = 0.0;
Dnum[N-1] = 0.0;
for (i=1; i < (N-1); i++) {
    if (x[i] <= Ccrit)
        Dnum[i] = 0.0;
    else {
        k = floor((x[i]-Ccrit)/dC); /* ska det vara +2? */
        if(k>=M)
            mexPrintf("k to big, i=%u, x=%1.5f\n",i,x[i]);
        Dnum[i] = Dtilde[k] + ( Dtilde[k+1] - Dtilde[k] )...
                *(( x[i] - Ccrit )/dC - k);

    }
}

/* Godunov flux */
fbkhat = Chat*mymax(0,mymin(v0_max,v0*(exp(-r_h*(Chat-f_ns*C_f))...
        -exp(-r_p*(Chat-f_ns*C_f)))));


for (i = 0; i < N; i++) {
   fbk[i] = x[i]*mymax(0,mymin(v0_max,v0*(exp(-r_h*(x[i]-f_ns*C_f))...
            -exp(-r_p*(x[i]-f_ns*C_f)))));


}

G[0] = 0.0;
G[N-2] = 0.0;
G[N-1] = 0.0;
for (i=1; i<(N-2); i++){
    if (x[i] <= x[i+1])
        G[i] = mymin(fbk[i],fbk[i+1]);
```

```
    else{
        if ((Chat-x[i])*(Chat-x[i+1])<0)
            G[i] = fbkhat;
        else
            G[i] = mymax(fbk[i],fbk[i+1]);
    }
}

dx[0] = Q_e/(area*dz)*(x[1]-x[0]);
dx[1] = Q_e/(area*dz)*(x[2]-x[1])-G[1]/dz+1.0/(pow(dz,2))*(Dnum[2]-Dnum[1]);
dx[2] = Q_e/(area*dz)*(x[3]-x[2])-(G[2]-G[1])/dz...
        +1.0/(pow(dz,2))*(dd[2]*(x[3]-x[2])+Dnum[3]-2*Dnum[2]+Dnum[1]);
for (i = 3; i < (N-3); i++) {
    if (i < (feedlayer-1+eps))
        dx[i] = Q_e/(area*dz)*(x[i+1]-x[i])-(G[i]-G[i-1])/dz...
                +1.0/(pow(dz,2))*(dd[i]*(x[i+1]-x[i])-dd[i-1]*(x[i]-x[i-1]))...
                +Dnum[i+1]-2*Dnum[i]+Dnum[i-1]);
    else if (i > (feedlayer+eps))
        dx[i] = -Q_u/(area*dz)*(x[i]-x[i-1])-(G[i]-G[i-1])/dz...
                +1.0/(pow(dz,2))*(dd[i]*(x[i+1]-x[i])-dd[i-1]*(x[i]-x[i-1]))...
                +Dnum[i+1]-2*Dnum[i]+Dnum[i-1]);
    else
        dx[i] = -(Q_u+Q_e)/(area*dz)*x[i]-(G[i]-G[i-1])/dz...
                +1.0/(pow(dz,2))*(dd[i]*(x[i+1]-x[i])-dd[i-1]*(x[i]-x[i-1]))...
                +Dnum[i+1]-2*Dnum[i]+Dnum[i-1])+Q_f*C_f/(area*dz);
}
dx[N-3] = -Q_u/(area*dz)*(x[N-3]-x[N-4])-(G[N-3]-G[N-4])/dz...
        +1.0/(pow(dz,2))*(-dd[N-4]*(x[N-3]-x[N-4])...
        +Dnum[N-2]-2*Dnum[N-3]+Dnum[N-4]);
dx[N-2] = -Q_u/(area*dz)*(x[N-2]-x[N-3])+G[N-3]/dz...
        -1.0/(pow(dz,2))*(Dnum[N-2]-Dnum[N-3]);
dx[N-1] = -Q_u/(area*dz)*(x[N-1]-x[N-2]);



/* soluble component S_I */
if (modeltype < 0.5) {
   for (i = N; i < (2*N); i++) {
      if (i < (feedlayer-1+N+eps))
         dx[i] = (-v_up*x[i]+v_up*x[i+1])/h;
      else if (i > (feedlayer+N+eps))
         dx[i] = (v_dn*x[i-1]-v_dn*x[i])/h;
      else
         dx[i] = (v_in*u[0]-v_up*x[i]-v_dn*x[i])/h;
   }
}
else if ((modeltype > 0.5) && (modeltype < 1.5)) {
   dx[N] = (Q_f*(u[0]-x[N]))/volume;
   for (i = (N+1); i < (2*N); i++)
      dx[i] = 0;
}
else if (modeltype > 1.5) {
   for (i = N; i < 2*N; i++)
      dx[i] = 0;
```

```
}


/* soluble component S_S */
if (modeltype < 0.5) {
   for (i = (2*N); i < (3*N); i++) {
      if (i < (feedlayer-1+(2*N)+eps))
         dx[i] = (-v_up*x[i]+v_up*x[i+1])/h;
      else if (i > (feedlayer+(2*N)+eps))
         dx[i] = (v_dn*x[i-1]-v_dn*x[i])/h;
      else
         dx[i] = (v_in*u[1]-v_up*x[i]-v_dn*x[i])/h;
   }
}
else if ((modeltype > 0.5) && (modeltype < 1.5)) {
   dx[2*N] = (Q_f*(u[1]-x[2*N]))/volume;
   for (i = (2*N+1); i < (3*N); i++)
      dx[i] = 0;
}
else if (modeltype > 1.5) {
   for (i = (2*N); i < (3*N); i++)
      dx[i] = 0;
}

/* soluble component S_O */
if (modeltype < 0.5) {
   for (i = (3*N); i < (4*N); i++) {
      if (i < (feedlayer-1+(3*N)+eps))
         dx[i] = (-v_up*x[i]+v_up*x[i+1])/h;
      else if (i > (feedlayer+(3*N)+eps))
         dx[i] = (v_dn*x[i-1]-v_dn*x[i])/h;
      else
         dx[i] = (v_in*u[7]-v_up*x[i]-v_dn*x[i])/h;
   }
}
else if ((modeltype > 0.5) && (modeltype < 1.5)) {
   dx[3*N] = (Q_f*(u[7]-x[3*N]))/volume;
   for (i = (3*N+1); i < (4*N); i++)
      dx[i] = 0;
}
else if (modeltype > 1.5) {
   for (i = (3*N); i < (4*N); i++)
      dx[i] = 0;
}

/* soluble component S_NO */
if (modeltype < 0.5) {
   for (i = (4*N); i < (5*N); i++) {
      if (i < (feedlayer-1+(4*N)+eps))
         dx[i] = (-v_up*x[i]+v_up*x[i+1])/h;
      else if (i > (feedlayer+(4*N)+eps))
         dx[i] = (v_dn*x[i-1]-v_dn*x[i])/h;
      else
         dx[i] = (v_in*u[8]-v_up*x[i]-v_dn*x[i])/h;
```

```
   }
}
else if ((modeltype > 0.5) && (modeltype < 1.5)) {
   dx[4*N] = (Q_f*(u[8]-x[4*N]))/volume;
   for (i = (4*N+1); i < (5*N); i++)
      dx[i] = 0;
}
else if (modeltype > 1.5) {
   for (i = (4*N); i < (5*N); i++)
      dx[i] = 0;
}

/* soluble component S_NH */
if (modeltype < 0.5) {
   for (i = (5*N); i < (6*N); i++) {
      if (i < (feedlayer-1+(5*N)+eps))
         dx[i] = (-v_up*x[i]+v_up*x[i+1])/h;
      else if (i > (feedlayer+(5*N)+eps))
         dx[i] = (v_dn*x[i-1]-v_dn*x[i])/h;
      else
         dx[i] = (v_in*u[9]-v_up*x[i]-v_dn*x[i])/h;
   }
}
else if ((modeltype > 0.5) && (modeltype < 1.5)) {
   dx[5*N] = (Q_f*(u[9]-x[5*N]))/volume;
   for (i = (5*N+1); i < (6*N); i++)
      dx[i] = 0;
}
else if (modeltype > 1.5) {
   for (i = (5*N); i < (6*N); i++)
      dx[i] = 0;
}

/* soluble component S_ND */
if (modeltype < 0.5) {
   for (i = (6*N); i < (7*N); i++) {
      if (i < (feedlayer-1+(6*N)+eps))
         dx[i] = (-v_up*x[i]+v_up*x[i+1])/h;
      else if (i > (feedlayer+(6*N)+eps))
         dx[i] = (v_dn*x[i-1]-v_dn*x[i])/h;
      else
         dx[i] = (v_in*u[10]-v_up*x[i]-v_dn*x[i])/h;
   }
}
else if ((modeltype > 0.5) && (modeltype < 1.5)) {
   dx[6*N] = (Q_f*(u[10]-x[6*N]))/volume;
   for (i = (6*N+1); i < (7*N); i++)
      dx[i] = 0;
}
else if (modeltype > 1.5) {
   for (i = (6*N); i < (7*N); i++)
      dx[i] = 0;
}
```

```
/* soluble component S_ALK */
if (modeltype < 0.5) {
   for (i = (7*N); i < (8*N); i++) {
      if (i < (feedlayer-1+(7*N)+eps))
         dx[i] = (-v_up*x[i]+v_up*x[i+1])/h;
      else if (i > (feedlayer+(7*N)+eps))
         dx[i] = (v_dn*x[i-1]-v_dn*x[i])/h;
      else
         dx[i] = (v_in*u[12]-v_up*x[i]-v_dn*x[i])/h;
   }
}
else if ((modeltype > 0.5) && (modeltype < 1.5)) {
   dx[7*N] = (Q_f*(u[12]-x[7*N]))/volume;
   for (i = (7*N); i < (8*N); i++)
      dx[i] = 0;
}
else if (modeltype > 1.5) {
   for (i = (7*N); i < (8*N); i++)
      dx[i] = 0;
}

}


/*
 * mdlTerminate - called when the simulation is terminated.
 */
static void mdlTerminate(SimStruct *S)
{
}

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

C. BSM2 Bürger-Diehl settler model initialisation m-script

```
% Initialisation file for all parameters and states in the B¸rger-Diehl model
% for secondary clarifiers. The Tak·cs settling velocity function is used.
%
% The model is implemented with variable number of layers.
%     set 'n' equal to the desired number of layers (10 minimum)
%
% Copyright: Magnus Arnell, IEA, Lund University, Sweden. 2014-12-01.

%% PARAMETERS
% Parameters in settling velocity (batch flux)
v0_max = 250;
v0 = 474;
r_h = 0.000576;
r_p = 0.00286;
f_ns = 0.00228;
Cmin  = f_ns.*4000;                % TSS cons below which settling velocity
                                   % is zero

% Extra parameters in compression
k    = 4*1e3;                      % parameter in solid stress [g/m^3]
s0   = (3600*24)^2*4*1e3;          % parameter in solid stress [g/(m*d^2)]
rhos = 1050*1e3;                   % Solids density        [g/m^3]
drho = rhos-998*1e3;               % Density difference    [g/m^3]
g    = 9.81*(3600*24)^2;           % Acceleration constant [m/d^2]


% Parameters in dispersion
alph1 = 0.0023;                    % parameter in dispersion [1/m]
alph2 = 1/200000;                  % parameter in dispersion [d/m^2]


%% Division into layers
n = 10;                            % Number of layers in settler
N = n+4;                           % Number of computational cells in model
B = 2;                             % Hight of thickening zone
H = 2;                             % Hight of clarifications zone
dz = (B+H)/n;                      % Layer depth
feedlayer = ceil(H/dz);            % Index of feed layer
z  = linspace(-H-2*dz,B+2*dz,n+5); % Layer boundaries
area = 1500;                       % Cross sectional area of settner
height = B+H;                      % Total hieght of settler
Z=linspace(-2*dz,height+2*dz,N);   % Z-variable for mesh of settler


%% PRECOMPUTATIONS
% Precomputation of Dtilde
Ccrit = 4000;
Cmax  = 20000;
M = n^2;
dC = (Cmax - Ccrit)/(M-1);
Dtilde(1) = 0;
d(1) = rhos*s0*max(0,min(v0_max,v0*(exp(-r_h.*(Ccrit-Cmin))-exp(-r_p...
```

```
    .*(Ccrit-Cmin))))/(g*drho*(Ccrit-Ccrit+k));
for i=2:M
    d(i) = rhos*s0*max(0,min(v0_max,v0*(exp(-r_h.*((Ccrit+(i-1)*dC)-Cmin))...
        -exp(-r_p.*((Ccrit+(i-1)*dC)-Cmin)))))/(g*drho*((Ccrit+(i-1)*dC)-Ccrit+k));
    Dtilde(i) = Dtilde(i-1)+dC/2*(d(i-1)+d(i));
end


% Find local maximum of fbk, Chat
f1 = @(x)v0.*(1./exp(r_h.*(x - Cmin)) - 1./exp(r_p.*(x - Cmin))) - x.*v0...
    .*(r_h./exp(r_h.*(x - Cmin)) - r_p./exp(r_p.*(x - Cmin)));
f2 = @(x)v0_max.*x - x.*(v0*(exp(-r_h.*(x-Cmin))-exp(-r_p.*(x-Cmin))));
Chat1 = fzero(f1,2000);             % Find zero of fbk_prime
Chat2 = fzero(f2,Cmax);             % Find upper zero of v0*C - fbk(C)
Chat = max(Chat1,Chat2);


% Calculate the CFL-condition %
C = 0:0.01:Cmax;
fbkPrime = v0.*(1./exp(r_h.*(C - Cmin)) - 1./exp(r_p.*(C - Cmin))) - C.*v0...
    .*(r_h./exp(r_h.*(C - Cmin)) - r_p./exp(r_p.*(C - Cmin))); %derivative of fbk
fbkPrime_max = max(abs(fbkPrime)); % find maximum of fbkPrime
dcomp_max    = rhos*max(0,min(v0_max,v0*(exp(-r_h.*(Ccrit-f_ns.*4000))...
    -exp(-r_p.*(Ccrit-f_ns.*4000)))))/(g*drho*(Ccrit-Ccrit+k));
Qf_max       = 200000;
X = -z.^2./(alph2*Qf_max-abs(z));
ddisp_max    = max(alph1*Qf_max.*exp(X).*(abs(z)<alph2*Qf_max));

CFL = 0.98*dz^2/(dz*(fbkPrime_max+Qf_max/area)+2*(dcomp_max+ddisp_max));

%% INPUTVECTORS FOR S-FUNCTION
SETTLERPAR = [ alph1 alph2 v0_max v0 r_h r_p f_ns Chat Ccrit dC ];

DIM = [ area height ];

LAYER = [ feedlayer n dz M ];

DTILDE = Dtilde;

LAYERBOUNDARIES = z;

%% INITIALIZATION OF SETTLER STATES
% Initialization of TSS concentrations
zm = z(1:end-1)+dz/2;
SETTLERINIT = 7000*(zm>1.5);

% INITIALIZATION OF SI STATES %
for i=(N+1):(2*N)
    SETTLERINIT(i) = 30;
end

% INITIALIZATION OF SS STATES %
for i=(2*N+1):(3*N)
    SETTLERINIT(i) = 0.80801;
```

```
end

% INITIALIZATION OF SO STATES %
for i=(3*N+1):(4*N)
    SETTLERINIT(i) = 2;
end

% INITIALIZATION OF SNO STATES %
for i=(4*N+1):(5*N)
    SETTLERINIT(i) = 13.5243;
end

% INITIALIZATION OF SNH STATES %
for i=(5*N+1):(6*N)
    SETTLERINIT(i) = 0.67193;
end

% INITIALIZATION OFF SND STATES %
for i=(6*N+1):(7*N)
    SETTLERINIT(i) = 0.6645;
end

% INITIALIZATION OFF SALK STATES %
for i=(7*N+1):(8*N)
    SETTLERINIT(i) = 3.8277;
end

% INITIALIZATION OFF SOLUBLE DUMMY 1 STATES %
for i=(8*N+1):(9*N)
    SETTLERINIT(i) = 0;
end

% INITIALIZATION OFF SOLUBLE DUMMY 2 STATES %
for i=(9*N+1):(10*N)
    SETTLERINIT(i) = 0;
end

% INITIALIZATION OFF SOLUBLE DUMMY 3 STATES %
for i=(10*N+1):(11*N)
    SETTLERINIT(i) = 0;
end

% INITIALIZATION OFF TEMP STATES %
for i=(11*N+1):(12*N)
    SETTLERINIT(i) = 14.8581;
end

%% SET MODELTYPE
% to use model with 10 layers for solubles use type 0 (COST Benchmark)
% to use model with 1 layer for solubles use type 1 (GSP-X implementation)
% to use model with 0 layers for solubles use type 2 (WEST implementation)

MODELTYPE = [ 0 ];
```

D. BSM2 Bürger-Diehl settler model s-function c-file

```
/*
 * SETTLER1DVBD is a C-file S-function for defining the B¸rger-Diehl
 * settler model with a variable number of layers.
 * can simulate 0, 1 or N layers for the solubles by using MODELTYPE
 *
 * Copyright: Magnus Arnell, IEA, Lund University, Lund, Sweden. 2014-12-01
 *
 * Note that wraped lines in this text is only for legibility.
 */

#define S_FUNCTION_NAME settler1dvBD_bsm2

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

#define XINIT           ssGetArg(S,0)
#define PAR             ssGetArg(S,1)
#define DIM             ssGetArg(S,2)
#define LAYER           ssGetArg(S,3)
#define LAYERBOUNDARIES ssGetArg(S,4)
#define DTILDE          ssGetArg(S,5)
#define MODELTYPE       ssGetArg(S,6)
#define TEMPMODEL       ssGetArg(S,7)
#define ACTIVATE        ssGetArg(S,8)

#define mymax(a,b) (a>=b ? a:b)
#define mymin(a,b) (a<=b ? a:b)


/*
 * mdlInitializeSizes - initialize the sizes array
 */
static void mdlInitializeSizes(SimStruct *S)
{
int nStates, nOutputs;

    int N    = (int)mxGetPr(LAYER)[1]+4;
    nStates  = 12*N;
    nOutputs = 12*N+45;


    ssSetNumContStates(    S, nStates);     /* number of continuous states           */
    ssSetNumDiscStates(    S, 0);           /* number of discrete states             */
    ssSetNumInputs(        S, 23);          /* number of inputs                      */
    ssSetNumOutputs(       S, nOutputs);    /* number of outputs                     */
    ssSetDirectFeedThrough(S, 1);           /* direct feedthrough flag               */
    ssSetNumSampleTimes(   S, 1);           /* number of sample times                */
    ssSetNumSFcnParams(    S, 9);           /* number of input arguments             */
    ssSetNumRWork(         S, 0);           /* number of real work vector ...
     * elements   */
    ssSetNumIWork(         S, 0);           /* number of integer work vector ...
     * elements*/
```

```
    ssSetNumPWork(         S, 0);           /* number of pointer work vector ...
     * elements*/
}

/*
 * mdlInitializeSampleTimes - initialize the sample times array
 */
static void mdlInitializeSampleTimes(SimStruct *S)
{
    ssSetSampleTime(S, 0, CONTINUOUS_SAMPLE_TIME);
    ssSetOffsetTime(S, 0, 0.0);
}


/*
 * mdlInitializeConditions - initialize the states
 */
static void mdlInitializeConditions(double *x0, SimStruct *S)
{
int i;

int N    = (int)mxGetPr(LAYER)[1]+4;

for (i = 0; i < 12*N; i++) {
   x0[i] = mxGetPr(XINIT)[i];
}

}

/*
 * mdlOutputs - compute the outputs
 */

static void mdlOutputs(double *y, double *x, double *u, SimStruct *S, int tid)
{
  double gamma, gamma_eff, modeltype, tempmodel;
  int i;

  int N        = (int)mxGetPr(LAYER)[1]+4;

  gamma = x[N-1]/u[13];
  gamma_eff = x[0]/u[13];

  modeltype = mxGetPr(MODELTYPE)[0];
  tempmodel = mxGetPr(TEMPMODEL)[0];

  if (modeltype < 0.5) {
     /* underflow */
     y[0]=x[N+N-1];
     y[1]=x[2*N+N-1];
     y[2]=u[2]*gamma;
     y[3]=u[3]*gamma;
     y[4]=u[4]*gamma;
     y[5]=u[5]*gamma;
```

```
y[6]=u[6]*gamma;
y[7]=x[3*N+N-1]; /* use oxygen in return sludge flow */
y[8]=x[4*N+N-1];
y[9]=x[5*N+N-1];
y[10]=x[6*N+N-1];
y[11]=u[11]*gamma;
y[12]=x[7*N+N-1];
y[13]=x[N-1];
y[14]=u[21];  /* Q_r */

if (tempmodel < 0.5)  /* Temp */
   y[15]=u[15];
else
   y[15]=x[11*N+N-1];

/* Dummy states */
y[16]=x[8*N+N-1];
y[17]=x[9*N+N-1];
y[18]=x[10*N+N-1];
y[19]=u[19]*gamma;
y[20]=u[20]*gamma;
y[21]=u[22];  /* Q_w */


/* effluent */
y[22]=x[N];
y[23]=x[2*N];
y[24]=u[2]*gamma_eff;
y[25]=u[3]*gamma_eff;
y[26]=u[4]*gamma_eff;
y[27]=u[5]*gamma_eff;
y[28]=u[6]*gamma_eff;
y[29]=x[3*N]; /* use oxygen in effluent flow */
y[30]=x[4*N];
y[31]=x[5*N];
y[32]=x[6*N];
y[33]=u[11]*gamma_eff;
y[34]=x[7*N];
y[35]=x[0];
y[36]=u[14]-u[21]-u[22];  /* Q_e */

if (tempmodel < 0.5)      /* Temp */
   y[37]=u[15];
else
   y[37]=x[11*N];

/* dummy states */
y[38]=x[8*N];
y[39]=x[9*N];
y[40]=x[10*N];
y[41]=u[19]*gamma_eff;
y[42]=u[20]*gamma_eff;
```

```
    /* internal TSS states */
    for (i = 0; i < N; i++) {
        y[i+43] = x[i];
    }

    y[N+43]=gamma;
    y[N+44]=gamma_eff;

    for (i = N; i < (12*N); i++)
       y[i+45] = x[i];
}

else if ((modeltype > 0.5) && (modeltype < 1.5)) {
    /* underflow */
    y[0]=x[N];
    y[1]=x[2*N];
    y[2]=u[2]*gamma;
    y[3]=u[3]*gamma;
    y[4]=u[4]*gamma;
    y[5]=u[5]*gamma;
    y[6]=u[6]*gamma;
    y[7]=x[3*N]; /* use oxygen in return sludge flow */
    y[8]=x[4*N];
    y[9]=x[5*N];
    y[10]=x[6*N];
    y[11]=u[11]*gamma;
    y[12]=x[7*N];
    y[13]=x[N-1];
    y[14]=u[21];  /* Q_r */

    if (tempmodel < 0.5)  /* Temp */
       y[15]=u[15];
    else
       y[15]=x[11*N];

    /* Dummy states */
    y[16]=x[8*N];
    y[17]=x[9*N];
    y[18]=x[10*N];
    y[19]=u[19]*gamma;
    y[20]=u[20]*gamma;
    y[21]=u[22];  /* Q_w */

    /* effluent */
    y[22]=x[N];
    y[23]=x[2*N];
    y[24]=u[2]*gamma_eff;
    y[25]=u[3]*gamma_eff;
    y[26]=u[4]*gamma_eff;
    y[27]=u[5]*gamma_eff;
    y[28]=u[6]*gamma_eff;
    y[29]=x[3*N]; /* use oxygen in effluent flow */
    y[30]=x[4*N];
    y[31]=x[5*N];
```

```
    y[32]=x[6*N];
    y[33]=u[11]*gamma_eff;
    y[34]=x[7*N];
    y[35]=x[0];
    y[36]=u[14]-u[21]-u[22];  /* Q_e */

    if (tempmodel < 0.5)      /* Temp */
       y[37]=u[15];
    else
       y[37]=x[11*N];

    /* dummy states */
    y[38]=x[8*N];
    y[39]=x[9*N];
    y[40]=x[10*N];
    y[41]=u[19]*gamma_eff;
    y[42]=u[20]*gamma_eff;

   /* internal TSS states */
    for (i = 0; i < N; i++) {
        y[i+43] = x[i];
    }

    y[N+43]=gamma;
    y[N+44]=gamma_eff;

    for (i = N; i < (2*N); i++)
       y[i+45] = x[N];
    for (i = (2*N); i < (3*N); i++)
       y[i+45] = x[2*N];
    for (i = (3*N); i < (4*N); i++)
       y[i+45] = x[3*N];
    for (i = (4*N); i < (5*N); i++)
       y[i+45] = x[4*N];
    for (i = (5*N); i < (6*N); i++)
       y[i+45] = x[5*N];
    for (i = (6*N); i < (7*N); i++)
       y[i+45] = x[6*N];
    for (i = (7*N); i < (8*N); i++)
       y[i+45] = x[7*N];
    for (i = (8*N); i < (9*N); i++)
       y[i+45] = x[8*N];
    for (i = (9*N); i < (10*N); i++)
       y[i+45] = x[9*N];
    for (i = (10*N); i < (11*N); i++)
       y[i+45] = x[10*N];
}

else if (modeltype > 1.5) {
    /* underflow */
    y[0]=u[0];
    y[1]=u[1];
    y[2]=u[2]*gamma;
    y[3]=u[3]*gamma;
```

```
y[4]=u[4]*gamma;
y[5]=u[5]*gamma;
y[6]=u[6]*gamma;
y[7]=u[7]; /* use oxygen in return sludge flow */
y[8]=u[8];
y[9]=u[9];
y[10]=u[10];
y[11]=u[11]*gamma;
y[12]=u[12];
y[13]=x[N-1];
y[14]=u[21];  /* Q_r */

if (tempmodel < 0.5)                /* Temp */
   y[15]=u[15];
else
   y[15]=x[11*N];

/* Dummy states */
y[16]=u[16];
y[17]=u[17];
y[18]=u[18];
y[19]=u[19]*gamma;
y[20]=u[20]*gamma;
y[21]=u[22];  /* Q_w */

/* effluent */
y[22]=u[0];
y[23]=u[1];
y[24]=u[2]*gamma_eff;
y[25]=u[3]*gamma_eff;
y[26]=u[4]*gamma_eff;
y[27]=u[5]*gamma_eff;
y[28]=u[6]*gamma_eff;
y[29]=u[7]; /* use oxygen in effluent flow */
y[30]=u[8];
y[31]=u[9];
y[32]=u[10];
y[33]=u[11]*gamma_eff;
y[34]=u[12];
y[35]=x[0];
y[36]=u[14]-u[21]-u[22];  /* Q_e */

if (tempmodel < 0.5)      /* Temp */
   y[37]=u[15];
else
   y[37]=x[11*N];

/* dummy states */
y[38]=u[16];
y[39]=u[17];
y[40]=u[18];
y[41]=u[19]*gamma_eff;
y[42]=u[20]*gamma_eff;
```

```
     /* internal TSS states */
     for (i = 0; i < N; i++) {
         y[i+43] = x[i];
     }

     y[N+43]=gamma;
     y[N+44]=gamma_eff;

     for (i = N; i < (2*N); i++)
        y[i+45] = u[0];
     for (i = (2*N); i < (3*N); i++)
        y[i+45] = u[1];
     for (i = (3*N); i < (4*N); i++)
        y[i+45] = u[7];
     for (i = (4*N); i < (5*N); i++)
        y[i+45] = u[8];
     for (i = (5*N); i < (6*N); i++)
        y[i+45] = u[9];
     for (i = (6*N); i < (7*N); i++)
        y[i+45] = u[10];
     for (i = (7*N); i < (8*N); i++)
        y[i+45] = u[12];
     for (i = (8*N); i < (9*N); i++)
        y[i+45] = u[16];
     for (i = (9*N); i < (10*N); i++)
        y[i+45] = u[17];
     for (i = (10*N); i < (11*N); i++)
        y[i+45] = u[18];
   }
}

/*
 * mdlUpdate - perform action at major integration time step
 */

static void mdlUpdate(double *x, double *u, SimStruct *S, int tid)
{
}

/*
 * mdlDerivatives - compute the derivatives
 */
static void mdlDerivatives(double *dx, double *x, double *u, SimStruct *S, int tid)
{

double area, feedlayer, dz, alph1, alph2, v0_max, v0, r_h, r_p, f_ns, Chat, fbkhat;
double Q_f, Q_e, Q_u, C_f, eps, Ccrit, dC, v_up, v_dn, v_in, h, volume, modeltype;
double tempmodel, activate;
int i, k;

int N  = (int)mxGetPr(LAYER)[1]+4;
int M  = (int)mxGetPr(LAYER)[3];

double *dd       = mxCalloc(N,sizeof(double));
```

```
double *Dnum     = mxCalloc(N,sizeof(double));
double *fbk      = mxCalloc(N,sizeof(double));
double *G        = mxCalloc(N,sizeof(double));
double *z        = mxCalloc(N,sizeof(double));
double *Dtilde   = mxCalloc(M,sizeof(double));


/* Get inputs */
area      = mxGetPr(DIM)[0];
feedlayer = mxGetPr(LAYER)[0]+1;

dz        = mxGetPr(LAYER)[2];

for (i=0; i<N; i++) {
    z[i] = mxGetPr(LAYERBOUNDARIES)[i+1];
}

for (i=0; i<M; i++) {
    Dtilde[i] = mxGetPr(DTILDE)[i];
}

eps = 0.01;
Q_f = u[14];
Q_u = u[21] + u[22];
Q_e = u[14] - Q_u;
C_f = u[13];

alph1  = mxGetPr(PAR)[0];
alph2  = mxGetPr(PAR)[1];
v0_max = mxGetPr(PAR)[2];
v0     = mxGetPr(PAR)[3];
r_h    = mxGetPr(PAR)[4];
r_p    = mxGetPr(PAR)[5];
f_ns   = mxGetPr(PAR)[6];
Chat   = mxGetPr(PAR)[7];
Ccrit  = mxGetPr(PAR)[8];
dC     = mxGetPr(PAR)[9];


h = mxGetPr(DIM)[1]/mxGetPr(LAYER)[1];  //uppdatera
modeltype = mxGetPr(MODELTYPE)[0];
tempmodel = mxGetPr(TEMPMODEL)[0];
activate = mxGetPr(ACTIVATE)[0];
volume = area*mxGetPr(DIM)[1]; //uppdatera
v_in = Q_f/area;
v_up = Q_e/area;
v_dn = Q_u/area;



/* Dispersion ska detta vara ekv. 13 & 14*/
dd[0] = 0.0;
dd[N-1] = 0.0;
```

```
for (i=1; i<N-1 ; i++) {
    dd[i] = fabs(z[i])<alph2*Q_f ? alph1*Q_f*exp((-1.0*pow(z[i],2.0)...
            /pow((alph2*Q_f),2.0))/(1.0-fabs(z[i])/(alph2*Q_f))):0.0;
}

/* Compression */
Dnum[0] = 0.0;
Dnum[N-1] = 0.0;
for (i=1; i < (N-1); i++) {
    if (x[i] <= Ccrit)
        Dnum[i] = 0.0;
    else {
        k = floor((x[i]-Ccrit)/dC); /* ska det vara +2? */
        if(k>=M)
            mexPrintf("k to big, i=%u, x=%1.5f\n",i,x[i]);
        Dnum[i] = Dtilde[k] + ( Dtilde[k+1] - Dtilde[k] )*(( x[i] - Ccrit )...
                /dC - k);
    }
}


/* Godunov flux */
fbkhat = Chat*mymax(0,mymin(v0_max,v0*(exp(-r_h*(Chat-f_ns*C_f))...
        -exp(-r_p*(Chat-f_ns*C_f)))));


for (i = 0; i < N; i++) {
   fbk[i] = x[i]*mymax(0,mymin(v0_max,v0*(exp(-r_h*(x[i]-f_ns*C_f))...
           -exp(-r_p*(x[i]-f_ns*C_f)))));

}

G[0] = 0.0;
G[N-2] = 0.0;
G[N-1] = 0.0;
for (i=1; i<(N-2); i++){
    if (x[i] <= x[i+1])
        G[i] = mymin(fbk[i],fbk[i+1]);
    else{
        if ((Chat-x[i])*(Chat-x[i+1])<0)
            G[i] = fbkhat;
        else
            G[i] = mymax(fbk[i],fbk[i+1]);
    }
}

dx[0] = Q_e/(area*dz)*(x[1]-x[0]);
dx[1] = Q_e/(area*dz)*(x[2]-x[1])-G[1]/dz+1.0/(pow(dz,2))*(Dnum[2]-Dnum[1]);
dx[2] = Q_e/(area*dz)*(x[3]-x[2])-(G[2]-G[1])/dz...
        +1.0/(pow(dz,2))*(dd[2]*(x[3]-x[2])+Dnum[3]-2*Dnum[2]+Dnum[1]);
for (i = 3; i < (N-3); i++) {
    if (i < (feedlayer-1+eps))
        dx[i] = Q_e/(area*dz)*(x[i+1]-x[i])-(G[i]-G[i-1])/dz...
                +1.0/(pow(dz,2))*(dd[i]*(x[i+1]-x[i])-dd[i-1]*(x[i]-x[i-1]))...
```

```
                +Dnum[i+1]-2*Dnum[i]+Dnum[i-1]);
    else if (i > (feedlayer+eps))
        dx[i] = -Q_u/(area*dz)*(x[i]-x[i-1])-(G[i]-G[i-1])/dz...
                +1.0/(pow(dz,2))*(dd[i]*(x[i+1]-x[i])-dd[i-1]*(x[i]-x[i-1]))...
                +Dnum[i+1]-2*Dnum[i]+Dnum[i-1]);
    else
        dx[i] = -(Q_u+Q_e)/(area*dz)*x[i]-(G[i]-G[i-1])/dz...
                +1.0/(pow(dz,2))*(dd[i]*(x[i+1]-x[i])-dd[i-1]*(x[i]-x[i-1]))...
                +Dnum[i+1]-2*Dnum[i]+Dnum[i-1])+Q_f*C_f/(area*dz);
}
dx[N-3] = -Q_u/(area*dz)*(x[N-3]-x[N-4])-(G[N-3]-G[N-4])/dz...
        +1.0/(pow(dz,2))*(-dd[N-4]*(x[N-3]-x[N-4])...
        +Dnum[N-2]-2*Dnum[N-3]+Dnum[N-4]);
dx[N-2] = -Q_u/(area*dz)*(x[N-2]-x[N-3])+G[N-3]/dz...
        -1.0/(pow(dz,2))*(Dnum[N-2]-Dnum[N-3]);
dx[N-1] = -Q_u/(area*dz)*(x[N-1]-x[N-2]);



/* soluble component S_I */
if (modeltype < 0.5) {
   for (i = N; i < (2*N); i++) {
      if (i < (feedlayer-1+N+eps))
         dx[i] = (-v_up*x[i]+v_up*x[i+1])/h;
      else if (i > (feedlayer+N+eps))
         dx[i] = (v_dn*x[i-1]-v_dn*x[i])/h;
      else
         dx[i] = (v_in*u[0]-v_up*x[i]-v_dn*x[i])/h;
   }
}
else if ((modeltype > 0.5) && (modeltype < 1.5)) {
   dx[N] = (Q_f*(u[0]-x[N]))/volume;
   for (i = (N+1); i < (2*N); i++)
      dx[i] = 0;
}
else if (modeltype > 1.5) {
   for (i = N; i < 2*N; i++)
      dx[i] = 0;
}


/* soluble component S_S */
if (modeltype < 0.5) {
   for (i = (2*N); i < (3*N); i++) {
      if (i < (feedlayer-1+(2*N)+eps))
         dx[i] = (-v_up*x[i]+v_up*x[i+1])/h;
      else if (i > (feedlayer+(2*N)+eps))
         dx[i] = (v_dn*x[i-1]-v_dn*x[i])/h;
      else
         dx[i] = (v_in*u[1]-v_up*x[i]-v_dn*x[i])/h;
   }
}
else if ((modeltype > 0.5) && (modeltype < 1.5)) {
   dx[2*N] = (Q_f*(u[1]-x[2*N]))/volume;
```

```
   for (i = (2*N+1); i < (3*N); i++)
      dx[i] = 0;
}
else if (modeltype > 1.5) {
   for (i = (2*N); i < (3*N); i++)
      dx[i] = 0;
}

/* soluble component S_O */
if (modeltype < 0.5) {
   for (i = (3*N); i < (4*N); i++) {
      if (i < (feedlayer-1+(3*N)+eps))
         dx[i] = (-v_up*x[i]+v_up*x[i+1])/h;
      else if (i > (feedlayer+(3*N)+eps))
         dx[i] = (v_dn*x[i-1]-v_dn*x[i])/h;
      else
         dx[i] = (v_in*u[7]-v_up*x[i]-v_dn*x[i])/h;
   }
}
else if ((modeltype > 0.5) && (modeltype < 1.5)) {
   dx[3*N] = (Q_f*(u[7]-x[3*N]))/volume;
   for (i = (3*N+1); i < (4*N); i++)
      dx[i] = 0;
}
else if (modeltype > 1.5) {
   for (i = (3*N); i < (4*N); i++)
      dx[i] = 0;
}

/* soluble component S_NO */
if (modeltype < 0.5) {
   for (i = (4*N); i < (5*N); i++) {
      if (i < (feedlayer-1+(4*N)+eps))
         dx[i] = (-v_up*x[i]+v_up*x[i+1])/h;
      else if (i > (feedlayer+(4*N)+eps))
         dx[i] = (v_dn*x[i-1]-v_dn*x[i])/h;
      else
         dx[i] = (v_in*u[8]-v_up*x[i]-v_dn*x[i])/h;
   }
}
else if ((modeltype > 0.5) && (modeltype < 1.5)) {
   dx[4*N] = (Q_f*(u[8]-x[4*N]))/volume;
   for (i = (4*N+1); i < (5*N); i++)
      dx[i] = 0;
}
else if (modeltype > 1.5) {
   for (i = (4*N); i < (5*N); i++)
      dx[i] = 0;
}

/* soluble component S_NH */
if (modeltype < 0.5) {
   for (i = (5*N); i < (6*N); i++) {
      if (i < (feedlayer-1+(5*N)+eps))
```

```
         dx[i] = (-v_up*x[i]+v_up*x[i+1])/h;
      else if (i > (feedlayer+(5*N)+eps))
         dx[i] = (v_dn*x[i-1]-v_dn*x[i])/h;
      else
         dx[i] = (v_in*u[9]-v_up*x[i]-v_dn*x[i])/h;
   }
}
else if ((modeltype > 0.5) && (modeltype < 1.5)) {
   dx[5*N] = (Q_f*(u[9]-x[5*N]))/volume;
   for (i = (5*N+1); i < (6*N); i++)
      dx[i] = 0;
}
else if (modeltype > 1.5) {
   for (i = (5*N); i < (6*N); i++)
      dx[i] = 0;
}

/* soluble component S_ND */
if (modeltype < 0.5) {
   for (i = (6*N); i < (7*N); i++) {
      if (i < (feedlayer-1+(6*N)+eps))
         dx[i] = (-v_up*x[i]+v_up*x[i+1])/h;
      else if (i > (feedlayer+(6*N)+eps))
         dx[i] = (v_dn*x[i-1]-v_dn*x[i])/h;
      else
         dx[i] = (v_in*u[10]-v_up*x[i]-v_dn*x[i])/h;
   }
}
else if ((modeltype > 0.5) && (modeltype < 1.5)) {
   dx[6*N] = (Q_f*(u[10]-x[6*N]))/volume;
   for (i = (6*N+1); i < (7*N); i++)
      dx[i] = 0;
}
else if (modeltype > 1.5) {
   for (i = (6*N); i < (7*N); i++)
      dx[i] = 0;
}

/* soluble component S_ALK */
if (modeltype < 0.5) {
   for (i = (7*N); i < (8*N); i++) {
      if (i < (feedlayer-1+(7*N)+eps))
         dx[i] = (-v_up*x[i]+v_up*x[i+1])/h;
      else if (i > (feedlayer+(7*N)+eps))
         dx[i] = (v_dn*x[i-1]-v_dn*x[i])/h;
      else
         dx[i] = (v_in*u[12]-v_up*x[i]-v_dn*x[i])/h;
   }
}
else if ((modeltype > 0.5) && (modeltype < 1.5)) {
   dx[7*N] = (Q_f*(u[12]-x[7*N]))/volume;
   for (i = (7*N+1); i < (8*N); i++)
      dx[i] = 0;
}
```

```
else if (modeltype > 1.5) {
   for (i = (7*N); i < (8*N); i++)
      dx[i] = 0;
}

/* soluble dummy 1  */
if (activate > 0.5) {
if (modeltype < 0.5) {
   for (i = (8*N); i < (9*N); i++) {
      if (i < (feedlayer-1+(8*N)-eps))
         dx[i] = (-v_up*x[i]+v_up*x[i+1])/h;
      else if (i > (feedlayer+(8*N)-eps))
         dx[i] = (v_dn*x[i-1]-v_dn*x[i])/h;
      else
         dx[i] = (v_in*u[16]-v_up*x[i]-v_dn*x[i])/h;
   }
}
else if ((modeltype > 0.5) && (modeltype < 1.5)) {
   dx[8*N] = (Q_f*(u[16]-x[8*N]))/volume;
   for (i = (8*N+1); i < (9*N); i++)
      dx[i] = 0;
}
else if (modeltype > 1.5) {
   for (i = (8*N); i < (9*N); i++)
      dx[i] = 0;
}
}
else if (activate < 0.5) {
   for (i = (8*N); i < (9*N); i++)
      dx[i] = 0;
}

/* soluble dummy 2  */
if (activate > 0.5) {
if (modeltype < 0.5) {
   for (i = (9*N); i < (10*N); i++) {
      if (i < (feedlayer-1+(9*N)-eps))
         dx[i] = (-v_up*x[i]+v_up*x[i+1])/h;
      else if (i > (feedlayer+(9*N)-eps))
         dx[i] = (v_dn*x[i-1]-v_dn*x[i])/h;
      else
         dx[i] = (v_in*u[17]-v_up*x[i]-v_dn*x[i])/h;
   }
}
else if ((modeltype > 0.5) && (modeltype < 1.5)) {
   dx[9*N] = (Q_f*(u[17]-x[9*N]))/volume;
   for (i = (9*N+1); i < (10*N); i++)
      dx[i] = 0;
}
else if (modeltype > 1.5) {
   for (i = (9*N); i < (10*N); i++)
      dx[i] = 0;
}
}
```

```
else if (activate < 0.5) {
   for (i = (9*N); i < (10*N); i++)
      dx[i] = 0;
}

/* soluble dummy 3  */
if (activate > 0.5) {
if (modeltype < 0.5) {
   for (i = (10*N); i < (11*N); i++) {
      if (i < (feedlayer-1+(10*N)-eps))
         dx[i] = (-v_up*x[i]+v_up*x[i+1])/h;
      else if (i > (feedlayer+(10*N)-eps))
         dx[i] = (v_dn*x[i-1]-v_dn*x[i])/h;
      else
         dx[i] = (v_in*u[18]-v_up*x[i]-v_dn*x[i])/h;
   }
}
else if ((modeltype > 0.5) && (modeltype < 1.5)) {
   dx[10*N] = (Q_f*(u[18]-x[10*N]))/volume;
   for (i = (10*N+1); i < (11*N); i++)
      dx[i] = 0;
}
else if (modeltype > 1.5) {
   for (i = (10*N); i < (11*N); i++)
      dx[i] = 0;
}
}
else if (activate < 0.5) {
   for (i = (10*N); i < (11*N); i++)
      dx[i] = 0;
}

/* Temp */
if (tempmodel < 0.5) {
   for (i = (11*N); i < (12*N); i++)
      dx[i] = 0;
}
else if ((tempmodel > 0.5) && (modeltype < 0.5)) {
   for (i = (11*N); i < (12*N); i++) {
      if (i < (feedlayer-1+(11*N)-eps))
         dx[i] = (-v_up*x[i]+v_up*x[i+1])/h;
      else if (i > (feedlayer+(11*N)-eps))
         dx[i] = (v_dn*x[i-1]-v_dn*x[i])/h;
      else
         dx[i] = (v_in*u[15]-v_up*x[i]-v_dn*x[i])/h;
      }
   }
else {
dx[11*N] = (Q_f*(u[15]-x[11*N]))/volume;
   for (i = (11*N+1); i < (12*N); i++)
      dx[i] = 0;
}

}
```

```
/*
 * mdlTerminate - called when the simulation is terminated.
 */
static void mdlTerminate(SimStruct *S)
{
}

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