

How can WEST and MATLAB be coupled?

Below is a brief write-up of some of the interfacing possibilities between WEST (actually: Tornado, the engine underlying WEST) and MATLAB.

Calling the Tornado command-line experiment executor from MATLAB

The least advanced way of calling Tornado from MATLAB consist in running the Tornado command-line experiment executor (texec.exe) from MATLAB, e.g. through the “system()” function. In this way it is possible to specify initial values (through the `-i` option) to override the experiment’s settings, as part of the call. However, simulated time series will have to be read from the experiment’s output file afterwards. The Tornado command-line experiment executor is included in all WEST installers (R2011, R2012, R2014 and R2016) and can be used with any WEST license type (WESTforOPERATORS, WESTforDESIGN, WESTforOPTIMIZATION and WESTforAUTOMATION).

Example:

```
[status,cmdout] = system('texec -i ".Aeration.y_S=1.5;.InternRecycle.Q_Out2=55338"
                        TwoASU.Dynamic.ObjEval.Exp.xml');
```

Integration of Tornado with MATLAB through Tornado’s MEX wrapper

TornadoMEX is a DLL that acts as a MEX wrapper around the Tornado engine. It basically extends the MATLAB M language with one function, named TornadoMEX, which takes 5 arguments (mode, experiment file name, initial value specification, desired output specification, logging specification) and returns a MATLAB matrix containing the desired output time series as columns. TornadoMEX is included in the WEST installer as of R2016. For other WEST releases (R2011, R2012, R2014) TornadoMEX is only available upon special request (conditions to be negotiated with the WEST business area manager).

As of R2016 both a 32- and a 64-bit version of TornadoMEX are available. For other releases, only a 32-bit version exists.

Below is an example of the application of TornadoMEX to the TwoASU dynamic simulation experiment that is shipped with WEST as a sample.

```
% Run experiment

Mode = 'TimeSeries';
ExpFileName = 'TwoASU.Dynamic.ObjEval.Exp.xml|.Tornado.Exp.ObjEval.Obj';
Initial = '.Aeration.y_S=1.5;.InternRecycle.Q_Out2=55338';
Output =
'.AeratedTank.C(S_NH);.AeratedTank.C(S_NO);.AeratedTank.C(S_O);.AeratedTank.C(S_S);.AeratedTank.C
(X_BA);.AeratedTank.C(X_BH);.AeratedTank.X_TSS;.AnoxicTank.C(S_NH);.AnoxicTank.C(S_NO);.AnoxicTan
k.C(S_O);.AnoxicTank.C(S_S);.Clarifier.X_Layer(1);.Clarifier.X_Layer(10);.Clarifier.X_Layer(2);.C
larifier.X_Layer(3);.Clarifier.X_Layer(4);.Clarifier.X_Layer(5);.Clarifier.X_Layer(6);.Clarifier.
X_Layer(7);.Clarifier.X_Layer(8);.Clarifier.X_Layer(9);.Removal_COD;.out_1.Outflow(S_NH);.out_1.O
utflow(S_NO);.out_1.Outflow(S_S)';
```

```

Log = 'Console';

y = TornadoMEX(Mode, ExpFileName, Initial, Output, Log);

% Retrieve time series (t is always first; rest is in alphabetical order)

t = y(:,1);
AeratedTank_C_S_NH = y(:,2);
AeratedTank_C_S_NO = y(:,3);
AeratedTank_C_S_O = y(:,4);
AeratedTank_C_S_S = y(:,5);
AeratedTank_C_X_BA = y(:,6);
AeratedTank_C_X_BH = y(:,7);
AeratedTank_X_TSS = y(:,8);
AnoxicTank_C_S_NH = y(:,9);
AnoxicTank_C_S_NO = y(:,10);
AnoxicTank_C_S_O = y(:,11);
AnoxicTank_C_S_S = y(:,12);
Clarifier_X_Layer_1 = y(:,13);
Clarifier_X_Layer_10 = y(:,14);
Clarifier_X_Layer_2 = y(:,15);
Clarifier_X_Layer_3 = y(:,16);
Clarifier_X_Layer_4 = y(:,17);
Clarifier_X_Layer_5 = y(:,18);
Clarifier_X_Layer_6 = y(:,19);
Clarifier_X_Layer_7 = y(:,20);
Clarifier_X_Layer_8 = y(:,21);
Clarifier_X_Layer_9 = y(:,22);
Removal_COD = y(:,23);
out_1_Outflow_S_NH = y(:,24);
out_1_Outflow_S_NO = y(:,25);
out_1_Outflow_S_S = y(:,26);

Clarifier_X_Layer = [ Clarifier_X_Layer_1(end) Clarifier_X_Layer_2(end) Clarifier_X_Layer_3(end)
Clarifier_X_Layer_4(end) Clarifier_X_Layer_5(end) Clarifier_X_Layer_6(end)
Clarifier_X_Layer_7(end) Clarifier_X_Layer_8(end) Clarifier_X_Layer_9(end)
Clarifier_X_Layer_10(end) ];

% Plot time series

figure;
plot(t, [AnoxicTank_C_S_S AnoxicTank_C_S_NH AnoxicTank_C_S_NO]);
title('Anoxic: Solubles');
legend('S\S', 'S\NH', 'S\NO');

figure;
plot(t, [AeratedTank_C_S_S AeratedTank_C_S_NH AeratedTank_C_S_NO]);
title('Aerobic: Solubles');
legend('S\S', 'S\NH', 'S\NO');

figure;
plot(t, [AeratedTank_C_X_BH AeratedTank_C_X_BA AeratedTank_X_TSS]);
title('Biomass');
legend('X\BH', 'X\BA', 'X\TSS');

figure;
plot(t, [AnoxicTank_C_S_O AeratedTank_C_S_O]);
title('Oxygen');
legend('Anoxic', 'Aerated');

figure;
plot(t, [out_1_Outflow_S_S out_1_Outflow_S_NH out_1_Outflow_S_NO]);

```

```

title('Effluent');
legend('S\_S', 'S\_NH', 'S\_NO');

figure;
plot(t, [Removal_COD]);
title('Evaluation');
legend('Removal COD');

figure;
h = axes();
barh(Clarifier_X_Layer);
title('Sludge Profile');
legend('Clarifier.X\_Layer');
set(h, 'YDir', 'reverse');
set(h, 'XScale', 'log');

```

Integration of Tornado with MATLAB through .NET

MATLAB supports the .NET framework since R2009a, which amongst other means that code implemented in .NET assemblies can be transparently called from the M language. This principle also allows for using the Tornado .NET API (named TornadoNET), in a way entirely similar to the use of this API in the context of C#, F#, VB.NET, IronPython, IronRuby and C++/CLI. The use of TornadoNET in MATLAB can therefore be regarded as a full-fledged alternative to TornadoMEX, which only offers limited access to Tornado. TornadoNET is included in all WEST installers (R2011, R2012, R2014 and R2016), but can only be used if one has a license for WESTforAUTOMATION. **The use of TornadoNET is by far the preferred, most powerful and most flexible way of integrating Tornado with MATLAB.**

Similarly to TornadoMEX, TornadoNET is available for 32- and 64-bit as of WEST R2016. For releases prior to that, it is only available for 32-bit. As Tornado references .NET 4.0, a MATLAB version is required that supports this platform, which is the case as of MATLAB 2011a.

Below is a complete example in M (again based on WEST's TwoASU dynamic simulation sample), including event handlers and exception handling. The program's structure and behaviour are entirely similar to implementations in C#, F#, VB.NET, IronPython, IronRuby and C++/CLI. One small difference however is the fact that MATLAB cannot handle "custom" events. "Standard" events were therefore added to TornadoNET, and can be activated through the `SetEnableStdEvents()` method.

```

NET.addAssembly('C:/Dev/Tornado/TornadoNET/bin/win32-msvc/DHI.Tornado.NET.dll');

try

    % Instantiate Tornado

    Tornado = TornadoNET.CTornado();
    Tornado.SetEnableStdEvents(true);

    % Set event handlers for Tornado

    addlistener(Tornado, 'EventSetMessageStd', @SetMessage);

    % Initialize Tornado

```

```

Tornado.Initialize('$ (TORNADO_DATA_PATH)/etc/Tornado.Main.xml|.Tornado', true);

% Load experiment

Exp = Tornado.ExpLoad('TwoASU.Dynamic.ObjEval.Exp.xml');

% Set event handlers for experiment

addListener(Exp, 'EventSetMessageStd', @SetMessage);

% Set initial values

Exp.ExpSetInitialValue('Aeration.y_S', 1.5);
Exp.ExpSetInitialValue('InternRecycle.Q_Out2', 55338);

% Initialize & run experiment

Exp.Initialize();
Exp.Run();

% Retrieve timeseries

ExpSimul = Exp.ExpGetSimul();
Buffer = ExpSimul.OutputBufferGet('Buffer');

t = Buffer.GetTimes().ToArray().double;

AnoxicTank_C_S_S = Buffer.GetValues('AnoxicTank.C(S_S)').ToArray().double;
AnoxicTank_C_S_NH = Buffer.GetValues('AnoxicTank.C(S_NH)').ToArray().double;
AnoxicTank_C_S_NO = Buffer.GetValues('AnoxicTank.C(S_NO)').ToArray().double;

AeratedTank_C_S_S = Buffer.GetValues('AeratedTank.C(S_S)').ToArray().double;
AeratedTank_C_S_NH = Buffer.GetValues('AeratedTank.C(S_NH)').ToArray().double;
AeratedTank_C_S_NO = Buffer.GetValues('AeratedTank.C(S_NO)').ToArray().double;

AeratedTank_C_X_BH = Buffer.GetValues('AeratedTank.C(X_BH)').ToArray().double;
AeratedTank_C_X_BA = Buffer.GetValues('AeratedTank.C(X_BA)').ToArray().double;
AeratedTank_X_TSS = Buffer.GetValues('AeratedTank.X_TSS').ToArray().double;

AnoxicTank_C_S_O = Buffer.GetValues('AnoxicTank.C(S_O)').ToArray().double;
AeratedTank_C_S_O = Buffer.GetValues('AeratedTank.C(S_O)').ToArray().double;

out_1_Outflow_S_S = Buffer.GetValues('out_1.Outflow(S_S)').ToArray().double;
out_1_Outflow_S_NH = Buffer.GetValues('out_1.Outflow(S_NH)').ToArray().double;
out_1_Outflow_S_NO = Buffer.GetValues('out_1.Outflow(S_NO)').ToArray().double;

Removal_COD = Buffer.GetValues('Removal_COD').ToArray().double;

% Retrieve final values

for i=1:10
    Clarifier_X_Layer(i) = Exp.ExpGetValue(strcat(strcat('Clarifier.X_Layer(',
        int2str(i)), ' '));
end;

% Retrieve objective values

ExpObjEval = Exp;
ObjMap = containers.Map;

QuantityNames = ExpObjEval.ObjQuantityEnumerate();
for i = 0:QuantityNames.Count() - 1

```

```

ObjNames = ExpObjEval.ObjQuantityObjValueEnumerate(QuantityNames.Item(i));
for j = 0:ObjNames.Count() - 1
    ObjMap(strcat(char(ObjNames.Item(j)), '(', char(QuantityNames.Item(i)), ')') =
        ExpObjEval.ObjQuantityObjValueGet(QuantityNames.Item(i), ObjNames.Item(j));
end
end
ObjMap('ObjValue') = ExpObjEval.ObjValueGet('ObjValue');

% Destroy experiment

Exp.Destroy();

% Plot timeseries

figure;
plot(t, [AnoxicTank_C_S_S' AnoxicTank_C_S_NH' AnoxicTank_C_S_NO]);
title('Anoxic: Solubles');
legend('S\_S', 'S\_NH', 'S\_NO');

figure;
plot(t, [AeratedTank_C_S_S' AeratedTank_C_S_NH' AeratedTank_C_S_NO]);
title('Aerobic: Solubles');
legend('S\_S', 'S\_NH', 'S\_NO');

figure;
plot(t, [AeratedTank_C_X_BH' AeratedTank_C_X_BA' AeratedTank_X_TSS]);
title('Biomass');
legend('X\_BH', 'X\_BA', 'X\_TSS');

figure;
plot(t, [AnoxicTank_C_S_O' AeratedTank_C_S_O]);
title('Oxygen');
legend('Anoxic', 'Aerated');

figure;
plot(t, [out_1_Outflow_S_S' out_1_Outflow_S_NH' out_1_Outflow_S_NO]);
title('Effluent');
legend('S\_S', 'S\_NH', 'S\_NO');

figure;
plot(t, [Removal_COD]);
title('Evaluation');
legend('Removal COD');

figure;
h = axes();
barh(Clarifier_X_Layer);
title('Sludge Profile');
legend('Clarifier.X\_Layer');
set(h, 'YDir', 'reverse');
set(h, 'XScale', 'log');

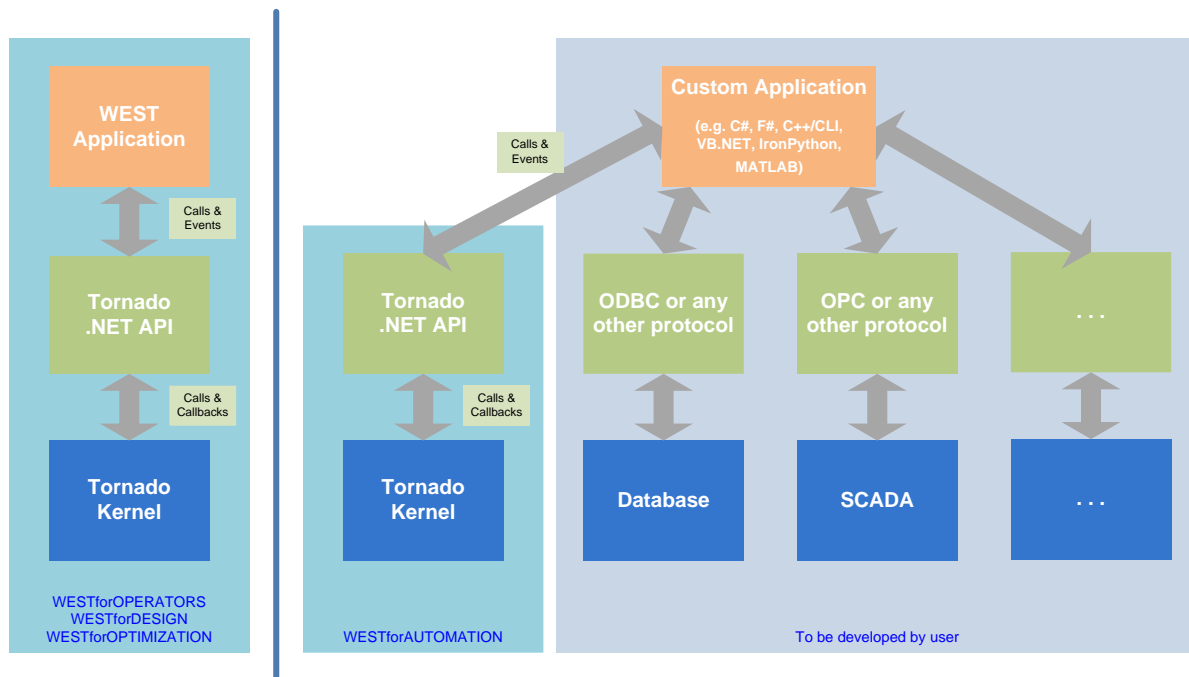
catch Ex
    if (isa(Ex, 'NET.NetException'))
        disp(Ex.ExceptionObject.ToString());
    else
        disp(Ex.message);
    end
end
----

```

```

function SetMessage(Obj, EventArgs)
    if EventArgs.Type == 0
        fprintf('I | ');
    elseif EventArgs.Type == 1
        fprintf('W | ');
    elseif EventArgs.Type == 2
        fprintf('E | ');
    else
        fprintf('U | ');
    end
    fprintf('%s\n', char(EventArgs.Message));
end

```



Conversion of flat MSL / Modelica models to S-functions (for use with MATLAB/Simulink)

Tornado's TMOF model compiler normally generates executable Tornado C code from flat Modelica models. However, it is also able to generate S-function code from the same input, which can hence be used as a block in the scope of Simulink. In addition to this, we are also able to convert flat MSL to flat Modelica (through a side-effect of the application of the TMSL model compiler). We can therefore ultimately go from flat MSL to S-functions as well.

The approach depicted above is currently experimental, and is in principle only available internally (meaning DHI + WEST Development Centers). Main reason for this is that TMOF (and some other software components that it depends on) is not included in the WEST installer. It is only included in the Tornado installer, which is normally restricted to internal use.

Conversion of a MATLAB fuzzy inference system (FIS) to executable code for Tornado

As of R2014, the WEST installer contains 2 command-line tools (tfis2t.exe and tfuzzy.exe) that can be used jointly to convert a MATLAB .fis file (which describes a fuzzy inference system) to executable C code that can be called from MSL (or Modelica) models. One requires a WESTforOPTIMIZATION or WESTforAUTOMATION license in order to be able to use tfis2t.exe and tfuzzy.exe.

Generating a C-callable DLL from MATLAB M code that can be invoked from MSL / Modelica models

The MATLAB Compiler is a product from the MathWorks that allows for building standalone applications and software components from MATLAB programs. If one has access to the MATLAB Compiler, it therefore becomes possible to convert an existing MATLAB program to a C-callable DLL that can hence be invoked from MSL or Modelica models (through Tornado's external function interface).

It should be noted that MATLAB is an interpreted language, which means that it is inherently slow in terms of performance. When generating a DLL from a MATLAB program, a wrapper is created around a (Java) virtual machine that executes the original MATLAB program. The resulting DLL will therefore more or less have the same performance as the original MATLAB program. Calling a MATLAB-generated DLL from a MSL / Modelica model therefore means that a component which is inherently slow is combined with one that is inherently fast, the run-time performance of the whole will therefore not be as fast as a full MSL / Modelica implementation.