What is the best way to implement my algorithm in Simulink®?

By Giampiero Campa, PhD, Technical Evangelist
MathWorks, 970 W 190 ST, Suite 530,
Torrance, CA, 90502, USA

giampiero.campa@mathworks.com
Outline

- Implementing algorithms in Simulink: overview
- An Extended Kalman Filter (EKF) for GPS/IMU Fusion
- Case Study: Implementing the EKF as a Simulink block
- Informal performance comparison
- Conclusions
An overview of options

- MATLAB® based:
  - MATLAB S-functions
  - MATLAB functions
  - MATLAB System objects™

- C based:
  - C S-functions
  - S-Function Builder
  - Legacy Code Tool (LCT)

- Simulink based:
  - Assembling Simulink blocks
Automatic code (and executable) generation

- Code generation allowed toward any target:
  - MATLAB functions
  - MATLAB system objects
  - Legacy Code Tool
  - Assembling Simulink blocks

- Only toward targets supporting noninlined S-functions:
  - C S-functions

- No code generation allowed:
  - MATLAB S-functions
Outline

- Implementing algorithms in Simulink: overview
- An Extended Kalman Filter (EKF) for GPS/IMU Fusion
- Case Study: Implementing the EKF as a Simulink block
- Informal performance comparison
- Conclusions
A simplified EKF for GPS/IMU sensor fusion

Angular speed in body frame (from IMU)

Acceleration in body frame (from IMU)

Velocity in earth frame (from GPS)

Roll, Pitch and Yaw angles estimate
EKF for GPS/IMU sensor fusion: summary

- 3 inputs, each one of size 3x1
- 1 output, also having size 3x1
- Using simplified solution relying only on internal roll and pitch estimates (Kingston-Beard)
  - Internal states are: roll and pitch estimated, a 2x2 P matrix, and the previous velocity in body frame (3x1)
  - Only minor linear algebra required (few 2x2 matrix multiplications and one inversion), so manual coding in C is affordable
- So how do we implement this?
Outline

- Implementing algorithms in Simulink: overview
- An Extended Kalman Filter (EKF) for GPS/IMU Fusion
- Case Study: Implementing the EKF as a Simulink block
- Informal performance comparison
- Conclusions
MATLAB S-function block (level 2)

Setup

Post Propagation Setup

Initialize Conditions

Output (mandatory)

Update
MATLAB S-function block: pros and cons

- Allows **fine control** of sizes, inputs, outputs, states, work vectors, etc.

- Allows use of any MATLAB function, toolbox, or data structure (with few limitations).

- Is **interpreted** (may be slower).

- Does **not allow code generation** and targeting (may only be used for simulation).
MATLAB System object block

Object definition

Parameters (to be passed from Simulink)

Discrete States
MATLAB System object block

(Re)initialize conditions

Step (Output and Update)
MATLAB System object block: pros and cons

- The API is **simpler and more elegant** than S-functions.
- Allows **code generation** (and can be executed both in interpreted or compiled mode).
- The mask is generated automatically.
- Relies more heavily on OO concepts. Constrained structure may take some adjustment.
MATLAB function block

Calls the Data Manager

Function Definition

Discrete states as persistent variables
MATLAB function block: pros and cons

- Perhaps the **simplest** method once you know how to use the Data Manager.

- Allows **code generation**

- The default mask is not too descriptive, so a more descriptive mask must be manually added if needed.

- The lack of structure allows for a lot of flexibility and potentially simplifies things.
MATLAB function with external states

Discrete states

Updated values of the discrete States
MATLAB function with external states

function \([\mathbf{P}, \mathbf{x}] = \text{ekf2tx}(\mathbf{F}, \mathbf{X}, \mathbf{V}, \mathbf{W}, \mathbf{V}, \mathbf{I}, \mathbf{F}, \mathbf{I}_m, \mathbf{V}_e, \mathbf{u}, \mathbf{a})\)

% calculate gps acceleration
agps = (ve-veld)/T;

% calculate heading
psi=atan2(ve(2),ve(1));

% rotate gps acceleration
rx=-cos(psi)*agps(1)-sin(psi)*agps(2);
ry=sin(psi)*agps(1)-cos(psi)*agps(2);
rx=9.80665-agps(3);

% solve for theta and phi given imu and gps accelerations
sth=(rx*ab(1)+rz*(abs(rx^2+rz^2-abc(1)^2)^0.5))/(realmin+rx^2+rz^2);
theta=atan2(sth*rx-ab(1),sthrz);
rth=rx*sin(theta)-rz*cos(theta);
sph=(ry*ab(2)+rth*(abs(ry^2+rth^2-abc(2)^2)^0.5))/(realmin+ry^2+rth^2);
phi=atan2(-sph*ry-ab(2),sph*rth);

% this works as the "measured" output in the ekf
y = [phi; theta];

% states (note, phi and theta are re-initialized with the previous estimates)
MATLAB function with external states: pros and cons

- Is a more structured way of implementing the algorithm in which the states are externally held by unit delays and therefore clearly visible. This simplifies the MATLAB Function.

- Is only here for comparison purposes, probably not worth the extra work with respect to the previous method.

- However it might be useful to implement continuous time algorithms. This can be done by using integrators instead of unit delays and calculating (in the MATLAB function block) the state derivative instead of the state update.
C S-function (level 2)

S-function name

Check Parameters
C S-function (level 2)

Initialize Conditions

```
static void mdlInitializeConditions(SimStruct *S)
{
    int_T i;
    real_T *X0 = ssGetRealDiscStates(S);
    real_T *P0 = mxGetPr(ssGetSFcnParam(S,4));
    real_T *X0 = mxGetPr(ssGetSFcnParam(S,5));
    real_T *ve0 = mxGetPr(ssGetSFcnParam(S,6));
    for (i=0;i<4;i++) X0[0+i]=P0[i];
    for (i=0;i<2;i++) X0[4+i]=x0[i];
    for (i=0;i<3;i++) X0[6+i]=ve0[i];
}
```

Update discrete states

```
/* mdlStart - initialize work vectors *****************************/
#define MDL_START
static void mdlStart(SimStruct *S)
{
    int_T i;
    real_T *w = ssGetWork(S);
    for (i=0;i<33;i++) w[i]=0;
}

/* mdlUpdate - compute the states *****************************/
#define MDL_UPDATE
static void mdlUpdate(SimStruct *S, int_T tid)
```
C S-function (level 2)

Calculate Outputs

static void mdlOutputs(SimStruct *S, int_T tid)
{
    real_T X = ssGetRealDiscStates(S);
    real_T y = ssGetOutputPortRealSignal(S, 0);
    InputRealPtrsType ve = ssGetInputPortRealSignalPtrs(S, 2);

    /* outputs */
    y[0] = X[4];
    y[1] = X[5];
    y[2] = atan2(*ve[1], *ve[0]);
}

static void mdlTerminate(SimStruct *S) {}

for (i=0;i<4;i++) nrm+=P[i]*P[i]; nrm=sqrt(nrm);
for (i=0;i<4;i++) P[i]=MIN(nrm,Plim[0])/MAX(nrm,2.2251e-308);
C S-function : pros and cons

- Supports SimStruct and the entire S-function API (therefore is even more powerful than MATLAB S-functions).

- Is compiled.

- It must be handwritten in C (not feasible for large algorithms requiring linear algebra and/or MATLAB toolboxes functions).

- Allows code generation only for targets supporting noninlined S-functions (unless you write a TLC file).
S-Function Builder

Parameters

Initial Conditions and Sample Time

Must be constants cannot be variables, (rebuild necessary if they are changed).
S-Function Builder

Outputs pane

Outputs calculation (xD is the vector of discrete states and work variables)
S-Function Builder

Update pane

Calculation of the update for the discrete states (and work variables).
S-Function Builder: pros and cons

- Less flexible than handwritten S-function. Initial states and sample time **cannot be passed as parameters**. Also, masks are handled differently than other blocks.

- It is compiled. However the generated S-function code uses a wrapper function, which causes a small additional overhead in simulation mode.

- The builder automatically generates a TLC file, therefore it **allows code generation for any target**.

- It still requires some C and S-function knowledge. Initialization must be performed through update function.
Legacy Code Tool: the C code

Initialization function:

```c
void ekf_init(double *work1, double *work2, double *p5, double *p6, double *p7) {
    unsigned int i;
    for (i=0;i<4;i++) work1[i]=p5[i];
    for (i=0;i<2;i++) work1[i+p6]=p6[i];
    for (i=0;i<3;i++) work1[i+p7]=p7[i];
    for (i=0;i<33;i++) work2[i]=0;
}
```

Output (step) function:

```c
void cKf_out(double *y1, double *u1, double *u2, double *u3, double *work1, double *work2, double *p1, double *p2, double *p3, double *p4) {
    /* integers */
    unsigned int i, j, k;
    /* scalar variables */
    double rx, ry, rz, sth, theta, rth, sph, psi, p, q, r;
    double *p = &work1[0], *x = &work1[4], *void = &work1[6];
```
Legacy Code Tool: assembling the block

Basic block specifications

Function specifications

Generation of S-function, MEX and TLC files, Simulink block
Legacy Code Tool: pros and cons

- Completely programmatic interface (no GUI) oriented towards the integration of existing C code.

- It is compiled. It does not use any wrapper. Supports less features than the S-Function Builder.

- S-function and TLC files are automatically generated. **Code generation is allowed for any target** and optimized for faster execution on embedded systems.

- It still requires some C knowledge (but no S-function knowledge).
Pure Simulink

- Only one model file required thus easy to ship. Only Simulink knowledge required.

- It is compiled.

- S-function, MEX and TLC files are not required. **Code generation is allowed for any target.**

- Harder for large algorithms requiring linear algebra, a lot of logic, and/or MATLAB toolbox functions. Harder to deal with the initialization function.
Outline

- Implementing algorithms in Simulink: overview
- An Extended Kalman Filter (EKF) for GPS/IMU Fusion
- Case Study: Implementing the EKF as a Simulink block
- Informal performance comparison
- Conclusions
Informal performance comparison

- Simulink blocks were created using the methods previously described (one for each method).

- Simulations were then run to verify that blocks reproduced the same outputs from the same inputs, and starting from the same initial conditions.

- Simple simulations (containing just a source and the EKF blocks, see next page) were then run programmatically on an Intel i7-3520, 2.90GHz, 4-Cores, 8GB RAM, Win64 laptop.
Example model for performance comparison

- Simulation time was set to 1e5 seconds, and the sampling time was 0.05 seconds.

- Elapsed time was measured using `tic` and `toc`, and averaged over 4 different executions (so, not rigorous).

- Maximum achievable frequency was calculated dividing the number of steps (1e5/0.05) by the elapsed time.
Simulation only

How fast are the blocks (KHz):

<table>
<thead>
<tr>
<th>Block Type</th>
<th>Speed (KHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MATLAB S-function</td>
<td>0</td>
</tr>
<tr>
<td>MATLAB System object</td>
<td>0</td>
</tr>
<tr>
<td>MATLAB function</td>
<td>50</td>
</tr>
<tr>
<td>MATLAB function ext. states</td>
<td>55</td>
</tr>
<tr>
<td>C S-function</td>
<td>120</td>
</tr>
<tr>
<td>S-Function Builder</td>
<td>180</td>
</tr>
<tr>
<td>Legacy Code Tool</td>
<td>190</td>
</tr>
<tr>
<td>Pure Simulink</td>
<td>70</td>
</tr>
</tbody>
</table>
GRT executables

How fast are the blocks (KHz):

- MATLAB System object
- MATLAB function
- MATLAB fcn. ext. states
- C S-function
- S-Function Builder
- Legacy Code Tool
- Pure Simulink

[Bar chart showing the speed of different blocks in KHz]
ERT executables

How fast are the blocks (KHz):
Arduino Uno

The previous Simulink models were augmented with digital output blocks to light up a LED after 5 minutes:
Arduino Uno

- Up until a sampling frequency of 100Hz the execution was fine, and the LED on pin 9 actually lit up after exactly 300s (as measured with a stopwatch).

- Whenever the frequency was pushed to 125Hz (base sample time $T=0.008$ sec) the different executables started to accumulate different delays (so termination happened 30-60s later than 5 minutes).

- Maximum achievable frequency was calculated dividing the number of steps ($300/0.008$) by the total elapsed time (e.g. 337 sec for the S-Function Builder block)
Arduino Uno

How Fast are the blocks (Hz):

- MATLAB System object: 114 Hz
- MATLAB function: 114 Hz
- MATLAB Fcn. Ext. States: 116 Hz
- S-Function Builder: 112 Hz
- Pure Simulink: 102 Hz
Arduino Uno

Executable size (bytes):

- MATLAB System object: 22000 bytes
- MATLAB function: 18000 bytes
- MATLAB Fcn. Ext. States: 18000 bytes
- S-Function Builder: 15000 bytes
- Pure Simulink: 18000 bytes
Outline

- Implementing algorithms in Simulink: overview
- An Extended Kalman Filter (EKF) for GPS/IMU Fusion
- Case Study: Implementing the EKF as a Simulink block
- Informal performance comparison
- Conclusions
Conclusions

- MATLAB System object, MATLAB function, S-Function Builder, and pure Simulink work for any kind of target.

- The performance comparison was somewhat informal (and compiler-dependent), however:
  - Methods based on C tend to be faster in simulation
  - Methods based on MATLAB tend to be faster for on-target execution (this could actually be important).
  - The MATLAB function block was consistently the fastest, while the pure Simulink block was the consistently the slowest.

- If you are starting from scratch, the MATLAB function block is probably your best option.