Eindhoven University of Technology

Internship Project

Matlab/C++ toolbox for factor graph modeling

Author:
Timur Bagautdinov

Supervisor at EE department:
Prof. dr. de Vries

Supervisor at CS department:
Prof. dr. Pechenizkiy

January 30, 2013

Contents

Introduction **2**

1 Factor graphs **3**
- 1.1 Constructing factor graphs . . . 3
- 1.2 Sum-propagation algorithm . . . 5
- 1.3 Example. Kalman filtering . . . 6
- 1.4 Example. AR model parameter estimation . . . 7

2 Toolbox **10**
- 2.1 Package structure . . . 10
- 2.2 Installation and dependencies . . . 10
- 2.3 Example. Kalman filter . . . 10
- 2.4 Extending the toolbox . . . 16
- 2.5 Connecting Matlab and C++ . . . 18

3 Application example **20**
- 3.1 Object tracking . . . 20

Conclusion **23**

Introduction

The main goal of this work was to create a generic Matlab framework for factor graph modeling, that could be used in near real-time settings. Matlab was an initial requirement, since one of the envisioned application areas is adaptive sound processing, and Matlab has a very rich toolbox for that field - DSP toolbox. However, the framework can be applied to a wider range of problems, which will be shown in the Section 3. We were focusing on creating a piece of software that is flexible and generic, allowing fast prototyping without a knowledge of generic-purpose languages like C++ or Java. The expected performance is in between of single-purpose hard-coded models and generic tools written in pure Matlab.

There are several existing toolboxes for factor graph modeling, such as, for example, `SumProductLab` [9] and `LibDAI` [12]. The first one is written in pure Matlab, and, thus, does not provide the required speed capabilities: in [4], benchmarks are provided that show Matlab being slower than C by an order of magnitude for such basic operations as matrix multiplication, which is the most frequent operation used in factor graph inference (see e.g. [11, Table 4]). Mathworks provide means to convert a subset of Matlab language into C, and we have tried doing that. Unfortunately, these code generation experiments were not successful, since the core of the framework is using some unsupported features, such as cell arrays. Thus for this framework to have efficiency comparable to the C programming language, it should have been rewritten completely. This could not be expected since the project have not been updated for two years.

The second one, `LibDAI`, provides limited Matlab interface, but only supports discrete variables, which made it inapplicable to our target application area.

To overcome the aforementioned constraints of existing tools, to get a deeper understanding of factor graphs and also to have full control over implementation details, we have developed our toolbox written in C++, and supplied with a full-fledged Matlab interface.

The report is organized as follows. In Section 1 we give a short mathematical introduction into factor graphs based on some well-known signal processing models. In Section 2 we describe the structure and current capabilities of the developed framework, and in Section 3 we show some examples of applications developed with this framework. The full code for the developed toolbox and examples is available on GitHub http://github.com/psycharo/factor-graph.

1 Factor graphs

In general sense, a *graphical model* is a way of representing dependencies and independencies of variables for distributions. *Factor graphs* correspond to a specific type of graphical model that is used to express the factorization of functions. They were originally introduced by Frey et al. in [6], and the notation that we will be using in this work is known as *Forney-style factor graphs* (FFG, introduced as *normal graphs* in [5]). Such well-known graphical models as Markov and belief networks, according to Hammersley-Clifford theorem, can be expressed via factor graphs [2].

1.1 Constructing factor graphs

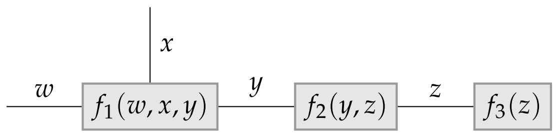


Figure 1: FFG for a factorization of $f(w, x, y, z)$.

We will now demonstrate on a small example how one can construct a Forney-Style factor graph given a factorization of a function. Consider a factorization of a multivariate function $f(w, x, y, z)$:

$$f(w, x, y, z) = f_1(w, x, y) f_2(y, z) f_3(z).$$

The FFG expressing this factorization is provided in the Figure 1. As one can see, the graph comprises three different types of elements:

- nodes, represented by f_1, f_2 and f_3
- edges, represented by y and z
- half-edges, represented by w and x

In general, FFG can be constructed out of a factorization according to the following three rules:

- for each factor, there is a node in the graph
- for each variable appearing in two factors, there is an edge in the graph
- for each variable appearing in one factor, there is a half-edge in the graph

Note that these rules imply that factorizations are restricted to those with no more than two factors sharing a variable. This restriction can be easily resolved though, by introducing additional variables and an equality constraint.

Let's assume that we have a factorization:

$$f(x,y,z) = f_1(x,y)f_2(y,z)f_3(y)$$

where the variable y is shared by three factors f_1, f_2 and f_3. To overcome this, we can introduce auxiliary variables y' and y'', and an equality constraint $y = y' = y''$. The factor that enables this equality constraint is:

$$f_=(y,y',y'') = \delta(y-y')\delta(y-y'')$$

where δ is Dirac's delta function. The FFG of the modified factorization is presented in the Figure 2.

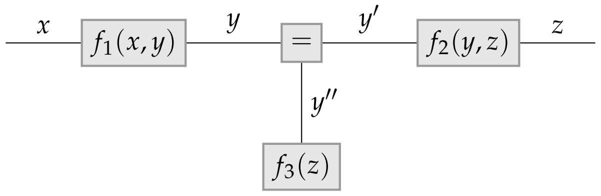


Figure 2: FFG for a modified factorization of $f(x,y,z)$.

Another useful and widely used factor is the one expressing zero-sum constraint $x + y + z = 0$:

$$f_+(x,y,z) = \delta(x+y+z),$$

which can of course be generalized to other similar constraints, e.g. the one expressing that one variable is the sum of two others $y = x + z$.

Factor graphs can be used to express a variety of models. For instance, linear space-state model is easily expressed using the factor nodes that we have discussed so far, see Figure 3. To understand the notations, let's see what the model equations look like:

$$X_t = AX_{t-1} + U_t \tag{1}$$

$$Y_t = HX_t + W_t \tag{2}$$

Where:

- Y_t is an observed variable, or measurement that is made on a timestep t;
- X_t and X_{t-1} are unobserved random variables on timestep $t-1$ and t correspondinly;
- U_t is the process noise, W_t is the observation noise (both usually assumed to be drawn from Gaussian distribution);

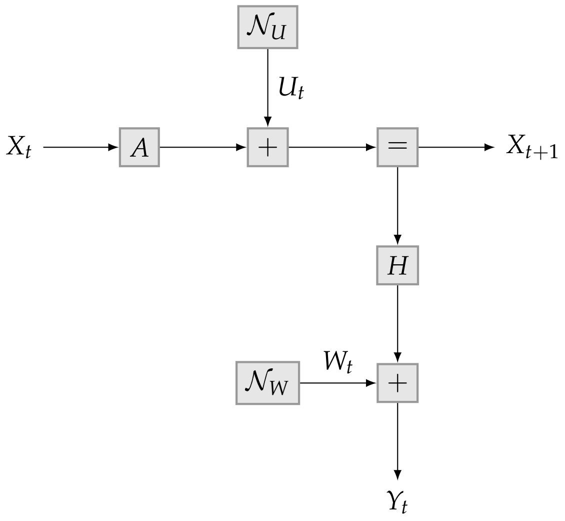


Figure 3: FFG for a state space model.

- A, H are the parameters of the model, A is sometimes called *state transition matrix* and H is referred to as an *observation matrix*.

Note that, in the Figure 3, we used nodes $\mathcal{N}_U$ and $\mathcal{N}_W$ that correspond to predefined distributions, in this case they are Gaussians with fixed means and variances.

1.2 Sum-propagation algorithm

When working with graphical models, there are two major steps [1]: *modeling*, i.e. formulating the problem as a graphical model, and *inference*, i.e. using the model to answer questions. Above we have discussed how can a problem be modeled using factor graphs, and now we will show how inference, or rather, its specific kind - *marginal inference*, can be done for this graphical model.

Let's consider a probability mass distribution (FFG in the Figure 4):

$$f(u,v,x,y,z) = f_1(u,v)f_2(v,x,z)f_3(x,y)f_4(y)f_5(z),$$

and assume that we are interested in how a specific variable u is distributed, or in other words, in a marginal distribution $p(u)$:

$$p(u) = \int_{v,x,y,z} f(u,v,x,y,z) = \int_{v,x,y,z} f_1(u,v)f_2(v,x,z)f_3(x,y)f_4(y)f_5(z).$$

The algorithm for computing this marginal on factor graphs is called *sum-propagation* algorithm, which boils down to substituting the global marginal

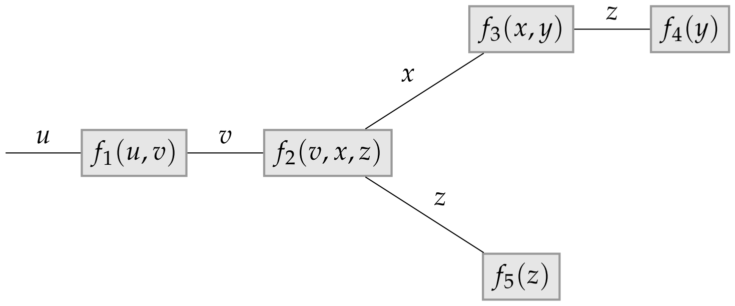


Figure 4: FFG for $f(u,v,x,y,z)$.

by a sequence of local marginals [10]. For our small example, it could be done by first computing the following local marginals:

$$\mu_1(x) = \int_z f_3(x,y) f_4(y)$$

$$\mu_2(v) = \int_{x,z} f_2(v,x,z)\mu_1(x) f_5(z)$$

And after that the target marginal distribution:

$$p(u) = \int_v f_1(u,v)\mu_2(v)$$

The local summaries $\mu_1(x)$ and $\mu_2(v)$ are often called *messages*, and the whole process can be seen as a sequence of messages passing from one node of the factor graph to another. What we have done to compute $p(u)$ is then can be generalized as the *sum-product rule*: the message from a node f along some edge x is formed as the product of f and all incoming messages along all edges except x, integrated over all involved variables except x itself. Note that for discrete variables integration is substituted by summation.

For cycle-free graph, the direct application of this rule leads to the sum-product algorithm. For graphs with cycles, the computation can be done iteratively, according to some message-passing schedule that is specified in advance.

1.3 Example. Kalman filtering

A well known algorithm Kalman filtering (smoothing) can be viewed as a sum-propagation algorithm applied to the FFG of the linear state space model (Figure 3), where all factors are multivariate Gaussians.

Messages in this settings are Gaussian distributions, which are represented by a pair of parameters, typically the mean vector m and the variance matrix V. It is also often useful to use the precision matrix $W = V^{-1}$ instead of the variance, and the weighted mean Wm instead of the mean.

The rules for actual message computations can be derived based on the factor's function and the selected message type. An example of such computational rules for several Gaussian factor nodes is given in the Table 1. Rules for other node and message types, and complete mathematical derivations of the rules can be found in [7], [11].

$\delta(x-y)\delta(x-z)$	$m_z = (W_x + W_y)^{\#}(W_x m_x + W_y m_y)$ $W_z = (W_x + W_y)$ $V_z = V_x(V_x + V_y)^{\#} V_y$
$\delta(x+y+z)$	$m_z = -m_x - m_y$ $V_z = V_x + V_y$ $V_z = W_x(W_x + W_y)^{\#} W_y$
$\delta(y - Ax)$	$m_y = Am_x$ $V_y = AV_x A^T$

Table 1: Message computation rules. $(.)^{\#}$ denotes Moore-Penroise pseudo-inverse.

To do the actual filtering, it is necessary to specify the message passing schedule. On the Figure 8 schedule is shown with ordinal numbers and arrows above them specifying the direction of messages. For instance, message from the zero-sum constraint node to equality constraint node has number 4, and an arrow pointing to the equality node.

Having the schedule, it is finally possible to run the sum-propagation algorithm on the factor graph, and use the latest available message for internal state X as an estimation (both for mean and variance) of the real value.

1.4 Example. AR model parameter estimation

The previous example can be further extended to estimate the model parameters [8]. In the Figure 6 we show the modified FFG for auto-regressive model, which is a special case of state-space model:

$$\mathbf{X}_t = \mathbf{A}\mathbf{X}_{t-1} + \mathbf{U}_t \tag{3}$$

$$Y_t = \mathbf{H}\mathbf{X}_t + W_t \tag{4}$$

where:

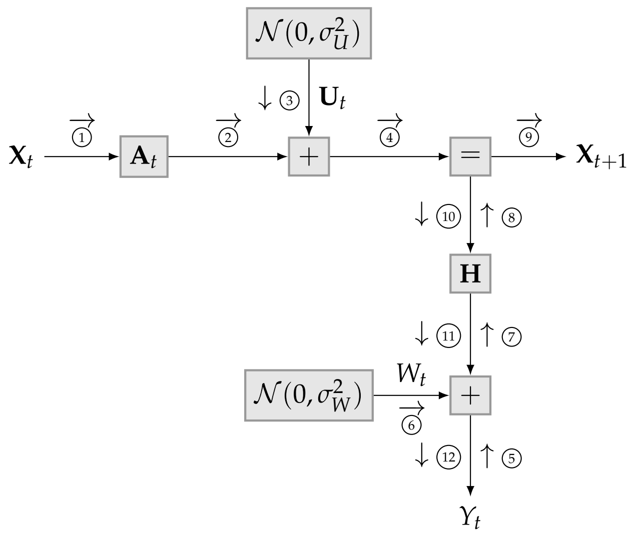


Figure 5: Message schedule for a Kalman filter.

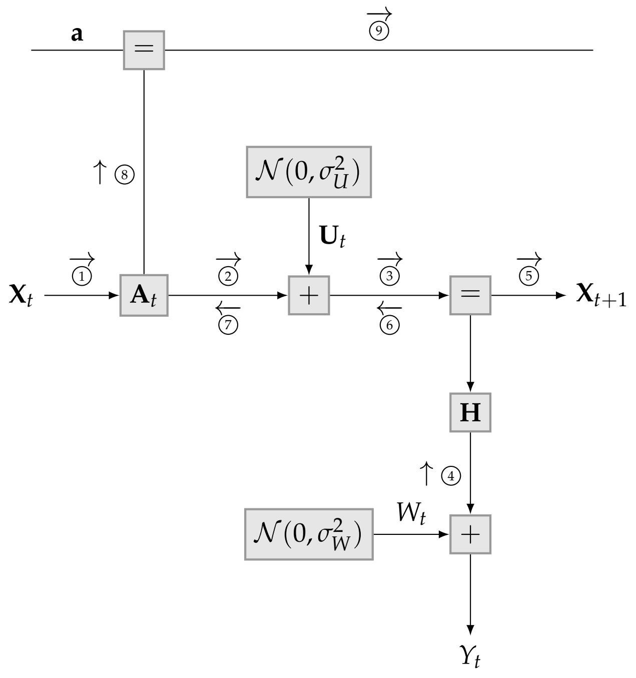


Figure 6: FFG and message schedule for AR model.

$$\mathbf{X}_t = [X_t, X_{t-1} \ldots, X_{t-M-1}]^T$$

$$\mathbf{A} = \begin{bmatrix} \mathbf{a^T} \\ \mathbf{I} \quad \mathbf{0} \end{bmatrix}$$

$$\mathbf{H} = [1, 0 \ldots, 0]$$

$$\mathbf{a}^T = [a_1, \ldots, a_M]^T$$

Having the FFG, we can now specify message schedule and run the sum-propagation algorithm. Messages in the Figure 6 model are specified by circumscribed numbers with arrows above them indicating the direction of the message. The full description of the messages, including message update formulas can be found in [8]. Messages 1 - 5 correspond to standard Kalman filtering: predicting the state and then correcting it with the observed evidence. Messages 6 and 7 correspond to Kalman smoothing, and messages 8 and 9 make an update of the AR coefficients estimation. Note that in our model observation and process noise variances are assumed to be constant, however, it is also possible to estimate those.

Now we can apply the sum-propagation algorithm to the FFG, and use the value of message 9 as an estimate of the model parameters.

2 Toolbox

In this section we present the Matlab toolbox that was developed and show how it can be used to define factor graphs and doing inference. We first take a look at the structure of the module, then consider the use-case of the framework on the example of Kalman filtering (Section 1.3), and finally show how the framework can be extended to add more nodes and message types.

2.1 Package structure

We tried to make the structure similar to the toolboxes that are widely used by Matlab users, such as `dsp` and `vision` system toolboxes. The toolbox is currently distributed as a source package, with the following contents

- `+ffg/` - core Matlab module, contains all factor node definitions
- `+ffg/+messages` - Matlab submodule, contains functions to create messages
- `+ffg/+tests` - Matlab submodule, contains unit tests written for `XUnit` framework
- `src/` - core sources, written in C++ (stl and mex)
- `lib/` - dependencies, currently contain the testing framework `XUnit`
- `examples/` - example applications and a simple graph drawing library `graphlayout`
- `installFFG.m` - installation script

2.2 Installation and dependencies

To install the framework, simply run the installation script: `installFFG.m`. This will compile the toolbox mex library and run unit tests. Note that you will need a supported compiler installed on your system. On Windows, it was tested with Windows SDK 7.1, on Linux with GCC C++ 4.6 compiler. Supported Matlab versions include 2012b and 2013a prerelease. For some example applications, the `vision` system toolbox is required.

2.3 Example. Kalman filter

We will show how to use the toolbox on the example of Kalman filter. First of all, we will need to construct the graph itself. In the toolbox, `ffg.Network` is the class that represents the graph (network) as a whole. So, first we need to create our empty network.

```
nwk = ffg.Network;
```

Now, we can add various factor nodes to this network. Every factor node in the toolbox is represented by a separate class, each being a descendant of the basic class `ffg.FactorNode`, and each factor node has the following common methods:

- `function result = id(this)` - get the identifier of the node within the network
- `function result = messages(this)` - get all the incoming messages for this node
- `function result = message(this, from)` - get the latest incoming message from the node `from`
- `function setMessage(this, message)` - manually set the message from the node specified in the `from` field of `message`

Currently, the following factor nodes are supported by the framework:

- `ffg.EqualityNode` - implements equality constraint
- `ffg.AddNode` - implements zero-sum constraint
- `ffg.MultiplicationNode` - implements matrix multiplication constraint
- `ffg.EvidenceNode` - corresponds to constant nodes (e.g. $\mathcal{N}_u$ in Figure 3) and half-edges (such as X_{t-1}, Y_t, X_t in Figure 3)
- `ffg.EquMultNode` - implements compound node that mimicks equality and multiplication nodes in one (see [10, Table 4.5]).
- `ffg.EstimateMultiplicationNode` - multiplication node enabling matrix estimation (such as $\mathbf{A}_t$ in Figure 6)
- `ffg.CustomNode` - enables custom message update rules

As one might have noticed, in the toolbox we do not make a distinction between a half-edge and a node, i.e. half-edge is just a separate type of node: `ffg.EvidenceNode`.

According to the Figure 3, we will need four types of factor nodes to define a Kalman filter:

- `ffg.MultiplicationNode` - for $\mathbf{A}$ and $\mathbf{H}$
- `ffg.EvidenceNode` - for X_{t-1}, X_t, Y_t and noise
- `ffg.AddNode` - for “$+$” nodes
- `ffg.EqualityNode` - for the “$=$” node

Every node class has a constructor that takes at least one parameter - the network to which this node belongs. The network assigns a unique identifier to every factor node, which can be retrieved by a method `function id(this)`. This identifier is very useful when debugging the network, as all the messages on the graph are indexed by sender's identifier. Creating all the nodes of the factor graph will look as follows:

```
% hidden variable, X_{t−1}
xin = ffg.EvidenceNode(nwk);
% hidden variable, X_t
xout = ffg.EvidenceNode(nwk);
% state transition matrix
A = ffg.MultiplicationNode(nwk);
A.setMatrix([1 1;
             0 1]);
% observation, corresponds to a half−edge y
y = ffg.EvidenceNode(nwk);
% equality node, connects a, b, xout
equ = ffg.EqualityNode(nwk);
% process noise, U_t
u = ffg.EvidenceNode(nwk);
% observation noise, W_t
w = ffg.EvidenceNode(nwk);
% observation model matrix, H
H = ffg.MultiplicationNode(nwk);
H.setMatrix([1 0]);
% zero−sum node, connects xin + u = equ
add_xin_u = ffg.AddNode(nwk);
% zero−sum node, connects equ + w = y
add_w_y = ffg.AddNode(nwk);
```

Here, we have used the method:

```
function setMatrix(this, matrix)
```

of `ffg.MultiplicationNode` to set the state transition and observation models:

$$\mathbf{A} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \tag{5}$$

$$\mathbf{H} = \begin{bmatrix} 1 & 0 \end{bmatrix} \tag{6}$$

Now that all the nodes are created, we need to specify how these nodes are connected with each other. This is done via the method:

```
function addEdge(from, to)
```

of `ffg.Network`. Note, that by default each connection is directed. For each call `nwk.addEdge(from,to)`, underneath outgoing connection is added to the node `from` and an incoming connection is added to the node `to`:

```
nwk.addEdge(xin, A);
nwk.addEdge(A, add_xin_u);
nwk.addEdge(u, add_xin_u);
nwk.addEdge(add_xin_u, equ);
nwk.addEdge(equ, H);
nwk.addEdge(equ, xout);
nwk.addEdge(H, add_w_y);
nwk.addEdge(w, add_w_y);
nwk.addEdge(add_w_y, y);
```

Some nodes, such as `ffg.EvidenceNode` and `ffg.EqualityNode` do not distinguish between incoming and outgoing connections, so, in this particular case `nwk.addEdge(e, xout)` and `nwk.addEdge(xout, e)` are equivalent.

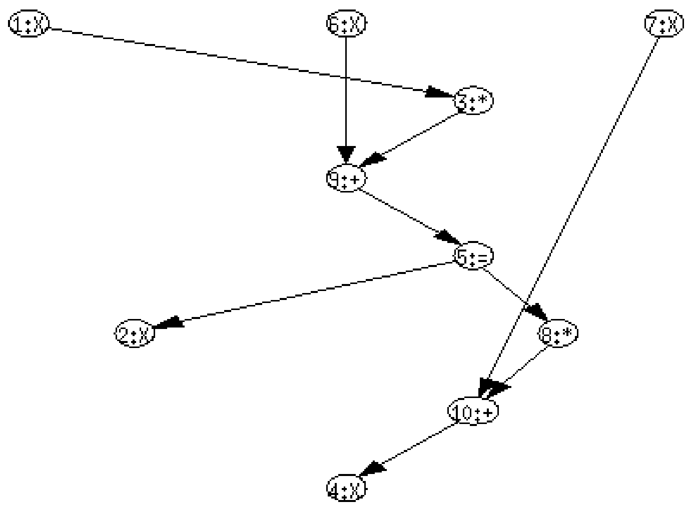


Figure 7: Network visualization via `function draw(this)` of `ffg.Network`.

This finishes the construction of our factor graph. To have a look at it, we can use the method `function draw(this)` of `ffg.Network`. Which should produce a figure identical to Figure 7, where each node is labeled with its id and short type label.

```
nwk.draw();
```

Now, we need to set the message passing schedule for the sum-propagation algorithm. This can be done by a method `function setSchedule(this, ... schedule)` of `ffg.Network`. `schedule` is just a cell array of pairs of nodes. For our example, setting the schedule would look as follows:

```
nwk.setSchedule({xin, A; ...
                 A, add_xin_u; ...
                 u, add_xin_u; ...
                 add_xin_u, equ; ...
                 w, add_w_y;...
                 y, add_w_y;...
                 add_w_y, H;...
                 H, equ;...
                 equ, xout});
```

Currently, there is also another way of running sum-propagation algorithm, which is by using

```
function propagate(this, message)
```

of the `ffg.EvidenceNode`, but it is applicable only to simplest networks, and might be removed in future.

Before the specified schedule can be applied, it is necessary to initialize the model by setting such parameters as process and observation noise variances, guesses or just dummy values for internal steps, so that the sum-propagation algorithm can actually form the necessary messages. In factor graphs, all such parameters are nothing but messages, that can be set from the "outside" by a method `function receive(this, message)` of `ffg.EvidenceNode`. Underneath, this method just stores the message inside the node and so that it can be sent it when the sum-propagation algorithm is executed.

Thus, in this particular example, we need to initialize the noise variances for `u, w` and make an initial guess about the internal state `xin`. To create a Gaussian message, there are two functions available:

- `function ffg.messages.gaussVariance(mean, variance, <from>)`
- `function ffg.messages.gaussPrecision(mean, precision, <from>)`

Each function creates a Gaussian message from the specified distribution parameters, which is represented by a Matlab structure with following fields:

- `type` - the type of the message, currently only `'VARIANCE'` and `'PRECISION'` are supported
- `from` (default = -1) - the identifier of the node that generated this message
- `mean` - mean of the distribution
- `var` or `precision` - variance or precision matrix of the distribution

For this example we will only use variance-based Gaussian messages:

```
% specifying some gaussian parameters
sd = 10.0;
% initialising X_{t-1}
xin.receive(ffg.messages.gaussVariance([0 0], sd^2 * eye(2,2)));
% process noise, setting it to zero
u.receive(ffg.messages.gaussVariance([0, 0], zeros(2,2)));
% observation noise
w.receive(ffg.messages.gaussVariance(0, sd^2));
```

To see the list of messages for a node, one can use a method `function ... messages(this)` of the node that returns a cell array of messages. It is also possible to retrieve a single message from a specific node by calling a method `function message(this, node)`. For instance for a node `w`, the call `w.messages{1}` will give the following result:

```
 ans =
    from: -1
    type: 'VARIANCE'
    mean: 0
     var: 100
```

Here, `from: -1` means that there is no sender of this message.

After all these preparations, we can do the actual Kalman filtering. In this example, we use the current iteration number with the added noise as input data, and run sum-propagation for 200 times (`N_ITERATIONS = 200`). On every step first of all `y` receives the new observation. Then we run a single run of message-passing algorithm according to the schedule that was set previously, and using `function makeStep(this)` of `ffg.Network`. Then, the resulting estimate on the current time step is passed as an initial estimate for the next step: `xin.receive()`. Method `function evidence(this)` of `ffg.EvidenceNode` is used to retrieve the current message that the node received from its only connection (this also can be done via `function ... message(this, sender)`). The complete code will then look as follows:

```
% place to store results for each
result = zeros(N_ITERATIONS, 2);
samples = zeros(N_ITERATIONS, 1);
for i = 1:N_ITERATIONS
    samples(i) = i+randn()*sd;
    % receiving subsequent observation
    y.receive(ffg.messages.gaussVariance(samples(i), 0));
    nwk.makeStep();
    xin.receive(xout.evidence());
    result(i,:) = [xout.evidence().mean(1), xout.evidence().var(1)];
end
```

Finally, we can plot the filtering results with the following code:

```
subplot(2,1,1);
hold on;
plot(1:N_ITERATIONS,result(:,1), 'Color', 'b');
plot(1:N_ITERATIONS,samples(:,1), 'Color', 'r');
hold off;
title('X value');
legend('Filtered', 'Observation', 'Location', 'SouthEast');
subplot(2,1,2);
plot(1:N_ITERATIONS,result(:,2));
title('variance');
```

which should generate plots similar to those in the Figure 8.

One can see, that it takes the filter (blue plot) around 40 iterations to converge to the true values, with the variance estimation dropping rapidly, but slightly too optimistic. Even large observation errors (e.g. a value fluctuation around 100-120 iteration number) do not make the model deviate too much from the real values.

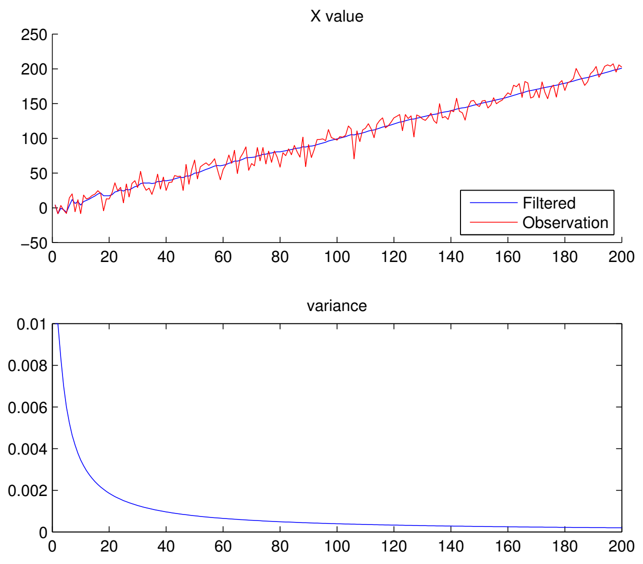


Figure 8: Output of Kalman filtering.

2.4 Extending the toolbox

The only documented way of extending the toolbox by means of Matlab is to use `ffg.CustomNode`. This class has a method

```
\mcode{function setFunction(this, functionName)},
```

that takes as a parameter the name of the function that implements message update rules.

The function should have a specific signature:

```
function out_msg = customnode_function_gauss(from, to, msgs)
```

where:

- `from` - identifier of the node
- `to` - identifier of the node where the message should (or should not be) sent
- `msgs` - an array of structs with following fields:
  - `type` - the type of the message: `'VARIANCE'` or `'PRECISION'`
  - `mean` - mean vector
  - `var` - variance | precision matrix, depending on `type`
  - `connection` - the type of connection `'INCOMING'` for incoming connections, `'OUTGOING'` for outgoing messages, and arbitrary for user-defined types of connections
  - `from` - the sender of the message
- `out_msg` is the resulting message to the `to` node

To better understand what the parameters mean, let's consider a function that implements zero-sum constraint $x + y - z = 0$:

```
function out_msg = customnode_function_gauss(from, to, msgs)
% from — the sender of the message
% to — the destination of the message
% msgs — array of messages (struct)

    % creating the output message
    out_msg = struct('type', 'VARIANCE', 'mean',[], 'var',[]);
    isForward = 1;

    forward = −1;
    % determining which one is the message from 'z' (if there is ...
        one)
    for i = 1:length(msgs)
        if strcmp(msgs(i).connection, 'OUTGOING') && to ≠ ...
            msgs(i).from
            isForward = 0;
            forward = i;
            out_msg.mean = msgs(i).mean;
            out_msg.var = msgs(i).var;
            break;
        end
    end

    if isForward
    % if there was no message from 'z', this means
    % that the message should be sent to 'z',
    % with the mean that equals to sum of all other means
        sz = length(msgs(1).mean);
        out_msg.mean = zeros(1, sz);
        out_msg.var = zeros(sz, sz);
        for i = 1:size(msgs, 2)
            out_msg.mean = out_msg.mean + msgs(i).mean;
            out_msg.var = out_msg.var + msgs(i).var;
        end
    else
    % if there was a message from 'z', constructing
    % a message by subtracting all the means from it
        for i = 1:size(msgs, 2)
            if to ≠ msgs(i).from && forward ≠ i
                out_msg.mean = out_msg.mean − msgs(i).mean;
                out_msg.var = out_msg.var + msgs(i).var;
            end
        end
    end
end
```

The reason why we used array of structs with predefined fields is that in this case the codegeneration can be done for it, which can speed up the execution dramatically. Matlab's `codegen` command unfortunately does not support containers and cell arrays.

2.5 Connecting Matlab and C++

The major part of the toolbox is written in C++, the sources are available in `src/`. Mathworks provide a way to call compiled code written in C, C++ and Fortran using so-called MEX files, or MEX-functions, which should be compiled Matlab's MEX-compiler: `mex` command.

A MEX function is a gateway function with a specific signature. For C/C++, the structure of a MEX-file with such a function looks as follows:

```
#include "mex.h"
void mexFunction(int nlhs, mxArray *plhs[],
                 int nrhs, const mxArray *prhs[])
{}
```

The arguments of this function are:

- `nlhs` - expected number of output variables
- `plhs` - array of pointers to the expected output
- `nrhs` - expected number of input variables
- `prhs` - array of pointers to the expected inputs

The `ffg` toolbox's MEX function is declared and implemented in `src/mexfactorgraph.cpp`, and thus its name in Matlab environment will be: `mexfactorgraph`. Unfortunately, there is no decent way to connect C++ classes. `mexfactorgraph` always expects at least two arguments to be passed from Matlab:

1. `type_name` - the class that is calling the MEX function (i.e. `ffg.Network` or some descendant of `ffg.FactorNode`
2. `function_name` - the name of the class's method being called. There are two reserved names:
   - `'create'` - corresponding to constructor
   - `'delete'` - corresponding to destructor

When the `'create'` method is called, the C++ object of `type_name` is created, and the pointer to this object is returned and stored in Matlab environment as an integer number. This pointer is then passed to MEX-function every time a method of the object is called. So, creation of an object of type `'Network'` will look as follows in Matlab:

```
pointer = mexfactorgraph('Network', 'create');
```

Then, if one wants to call a method for this object (e.g. a destructor), he should call:

```
mexfactorgraph('Network', 'delete', pointer);
```

Almost every other method for every class in the `ffg` toolbox is underneath just calling a MEX-function with the required parameters. To avoid code duplication, we joined the common part of the object lifecycle - creation and deletion - into Matlab class `ffg.CppObject`. The same holds for factor nodes: all the common methods are already taken care of in the base class `ffg.FactorNode`.

The main deficiency of our approach is that, since all the parameters passed to the MEX-function are destroyed right after the function terminates, it is absolutely necessary to copy the data of `mxArray`-s to C-style arrays. This copying, of course, decreases the performance. Also, the calls to MEX-functions are not very-well optimized in Matlab, so, frequent calls can cause performance issues.

3 Application example

In this section we describe a Matlab application created to demonstrate use-cases of the FFG framework.

3.1 Object tracking

This application demonstrates how the framework can be used to solve the real-world problem - tracking an object on a video signal. The `vision` system toolbox was used for detection purposes. Full code of the example is available in `examples/object_tracking_video.m`. Here we will shortly discuss the parameters of the model that were used and the interpretation of results.

We also used a Kalman filter for this problem (Figure 3, with the following internal state model:

$$\mathbf{X}_t = \begin{bmatrix} x & v_x & y & v_y \end{bmatrix}^T \tag{7}$$

Where x and y are object's coordinates, v_x and v_y are velocities along the corresponding axes. Acceleration is assumed to be zero, i.e. the speed is constant.

Thus, the state transition model will look as follows:

$$\mathbf{A} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix} \tag{8}$$

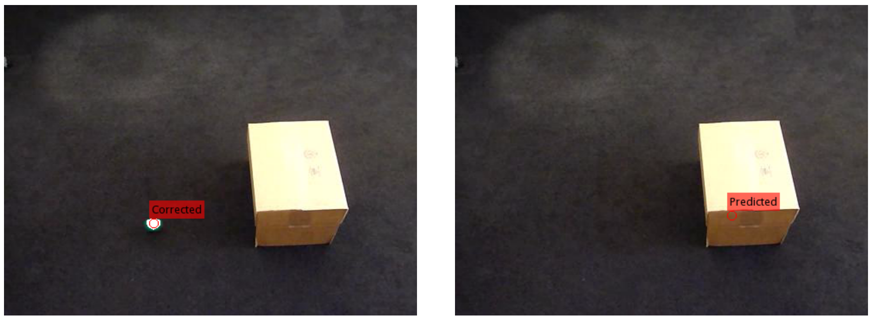
We are not interested in the realistic time and distance scales here, so the units of the coordinates are pixels, and the speed fictional.

We only can observe the coordinates of the object, by using `vision`'s detection methods. The observation model will then be:

$$\mathbf{H} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{9}$$

For convenience, we developed a class `KalmanFilter` that mimics the interface of `vision.KalmanFilter`. It has three methods that are of our interest.

- `function this = KalmanFilter()`
  - `InitialLocation` - the initial position of the object
  - `InitialEstimateError` - variances for the initial position and velocity
  - `MotionNoise` - process noise variances
  - `MeasurementNoise` - measurement variance
- `function [z_corr, x_corr, P_corr] = correct(this, measurement)`, where `z_corr` is the corrected coordinates, `x_corr` is the updated state mean, and `P_corr` is the state's variance matrix



(a) Corrected (b) Predicted

Figure 9: Object position

- `function z_pred = predict(this)`, where `z_pred` is the predicted position of the object

The constructor creates the network and initializes factor nodes with messages according to the arguments. The network structure is almost identical to the one described in Section 1.3, so we do not reproduce it here.

The `correct()` method is used to "train" the model by feeding it more evidence, and to retrieve corrected measurements (see Figure 9(a)) and the updated internal state. In our video sample the measurements are the coordinates that were detected by means of computer vision algorithms, which are a part of Matlab's `vision` toolbox.

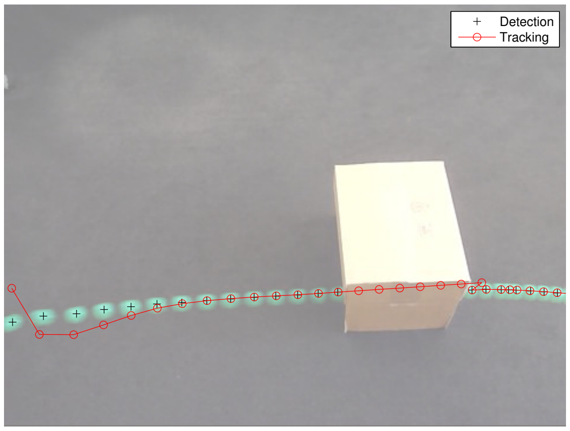
The `predict()` method is used to predict the position of the object when no detection is available. In the video sample that was used it is the moment when the ball goes through the box (see e.g. Figure 9(b)).

The latter two functions imply different message schedules, since when the prediction is made there is no observation available. So, in the constructor we create two schedules - `this.PredictSchedule` and `this.CorrectSchedule`, which are switched via `function setSchedule()` of `ffg.Network`, when the corresponding method is called:

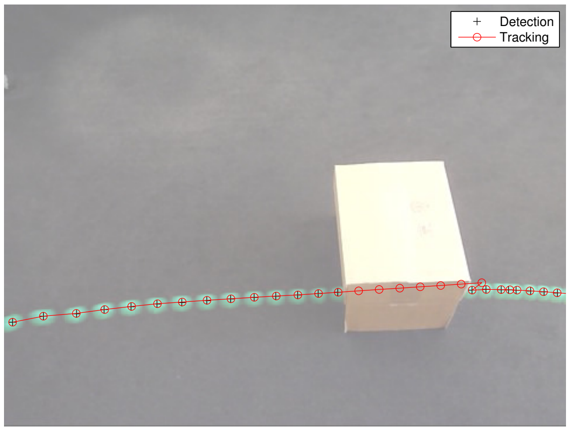
```
this.PredictSchedule = {this.xPrev, A;...
              A, add_X_U;...
              add_X_U, this.equ;...
              this.equ, this.xNext};

this.CorrectSchedule = {this.xPrev, A;...
            A, add_X_U;...
            add_X_U, this.equ;...
            this.equ, this.xNext;...
            this.Y, add_H_W;...
            add_H_W, this.H;...
            this.H, this.equ;...
            this.equ, this.xNext};
```

In the Figure 10 we demonstrate how the object is being tracked by the model, in case the initial location is selected arbitrarily (Figure 10(a)), and if it is based on the initial detection (Figure 10(b)). The *detection* is a position of the object based on a computer vision algorithm from the `vision` toolbox. The algorithm is not important for our example, but what is important is that the detection is not completely precise. That is why filtering can be used to *track* the position of the object based both on the current detection and the data from previous detections. In both cases the model eventually converges to approximately the same internal state estimation.



(a) Manually set initial position



(b) Detected initial position

Figure 10: The detected and tracked trajectory of the object.

Conclusion

In this work, the generic Matlab/C++ toolbox for modeling and inference on factor graphs toolbox is presented. One of the important achievements, we believe, is that the toolbox can be used and extended, without relying on a generic-purpose language. Matlab provides a wide range of tools for matrix manipulation and signal processing (such as Fast Fourier transformations), which can be easily plugged into the developed toolbox, and which would not require developing an entire model from scratch. It also provides performance beyond pure Matlab implementations, since its core is C++-based and can be directly compiled into binary code. We have not presented formal results of performance comparison, however, even the generic benchmarks [4] show that Matlab is far slower than C/C++ in a majority of relevant cases.

We have demonstrated that the developed framework can be used to work with several signal processing models, such as Kalman filtering and autoregressive model. We also have shown that the toolbox can be applied to the computer vision problems which require near real-time performance. We believe that we have proven that basically any linear Gaussian model can be expressed using our toolbox.

Currently we are planning to use the toolbox to create an adaptive noise reduction system. The toolbox is now limited by Gaussian models only, however a wide range of noise reduction models go beyond that limitation. To solve this, we are planning to integrate Variational Bayes methods [3]. Another direction for improvement is introducing ways to compare model, such as, for instance, Bayesian evidence.

References

[1] D. Barber. *Bayesian reasoning and machine learning*. Cambridge University Press, 2012.

[2] J. Besag. Statistical analysis of non-lattice data. *The statistician*, pages 179–195, 1975.

[3] C.M. Bishop et al. *Pattern recognition and machine learning*, volume 4. springer New York, 2006.

[4] Justin Domke. Julia, Matlab and C. `http://justindomke.wordpress.com/2012/09/17/julia-matlab-and-c/`, 2012. [Online; accessed 23-January-2013].

[5] G.D. Forney Jr. Codes on graphs: normal realizations. *Information Theory, IEEE Transactions on*, 47(2):520–548, 2001.

[6] B.J. Frey. Factor graphs and algorithms. In *PROCEEDINGS OF THE ANNUAL ALLERTON CONFERENCE ON COMMUNICATION CONTROL AND COMPUTING*, volume 35, pages 666–680. UNIVERSITY OF ILLINOIS, 1997.

[7] S. Korl. *A factor graph approach to signal modelling, system identification and filtering*. Hartung-Gorre, 2005.

[8] S. Korl, H.A. Loeliger, and A.G. Lindgren. Ar model parameter estimation: from factor graphs to algorithms. In *Acoustics, Speech, and Signal Processing, 2004. Proceedings.(ICASSP'04). IEEE International Conference on*, volume 5, pages V–509. IEEE, 2004.

[9] Henry Leung. SumProductLab for Factor Graphs. `http://www.mathworks.com/matlabcentral/fileexchange/26607`, 2010. [Online; accessed 23-January-2013].

[10] H.A. Loeliger. An introduction to factor graphs. *Signal Processing Magazine, IEEE*, 21(1):28–41, 2004.

[11] H.A. Loeliger, J. Dauwels, J. Hu, S. Korl, L. Ping, and F.R. Kschischang. The factor graph approach to model-based signal processing. *Proceedings of the IEEE*, 95(6):1295–1322, 2007.

[12] J.M. Mooij. libdai: A free and open source c++ library for discrete approximate inference in graphical models. *The Journal of Machine Learning Research*, 11:2169–2173, 2010.