Evaluation of the Monte Carlo Localization algorithm

-- May 2008 --

Author :

Andrei Stanculescu

Project Supervisor :

Michal Sojka

Abstract

A key aspect of a mobile robot is the ability to locate itself with high accuracy in the environment in which it operates. This paper presents the Monte Carlo localization algorithm and an implementation of it using Simulink S-Functions. Also, it includes a brief description of Simulink and an overview of the Simulink S-Functions.

1. Introduction

1.1 The Localization Problem

Localization means estimating the position of a mobile robot on a known or predicted map. It is one of the major challenges in mobile robotics.

Localization is necessary in robot applications because the robot uses its present pose (position and orientation) to decide on the next actions. Localization is difficult because of the noise and inaccuracies in sensors, and their limited range. In this context, localization techniques using particle filters have been developed.

Throughout the last decade, sensor-based localization has been recognized as a key problem in mobile robotics. Localization is a version of on-line temporal state estimation, where a mobile robot seeks to estimate its position in a global coordinate frame. The localization problem comes in two flavors: *global localization* and *position tracking*. The second is by far the most-studied problem; here a robot knows its initial position and “only” has to accommodate small errors in its odometry as it moves. The global localization problem involves a robot which is not told its initial position; hence, it has to solve a much more difficult localization problem, that of estimating its position from scratch (this is sometimes referred to as the *kidnapped robot problem*. The ability to localize itself—both locally and globally—played an important role in a collection of recent mobile robot applications [1].

1.2 The Monte Carlo Localization method

The **M**onte **C**arlo **L**ocalization (in short: MCL) methods were introduced in the Seventies, and recently rediscovered independently in the target-tracking, statistical and computer vision literature, and they have also been applied in dynamic probabilistic networks.

The MCL algorithm is applicable to both local and global localization problems. It is one of the most popular localization algorithms in robotics, because of its easy implementation and performance in a wide range of localization problems.

The key idea of MCL is to represent the belief by a set of samples (also called: *particles*), drawn according to the posterior distribution over robot poses. The MCL approach to representing the uncertainty is that instead of describing the probability density function itself, we represent it by maintaining a set of *samples* that are randomly drawn from it [2].

The MCL method is a particle filter that uses fast sampling techniques to represent the robot's belief. When the robot moves or senses, importance re-sampling is applied to estimate the posterior distribution.

The most important aspect in robot localization in the estimation of the state of the robot at the current time. The state vector contains the position and orientation of the robot. The probability density function is taken to represent all the knowledge possessed about the state, and from the current position can be estimated. To localize the robot, the density must be recursively computed. This is done in two phases: the prediction phase and the update phase.

- the prediction phase : In the first phase we start from the set of particles computed in the previous iteration, and apply the motion model to each particle by sampling from the density. In doing so a new set is obtained that approximates a random sample from the predictive density.In this phase we have not yet incorporated any sensor measurement.

- the update phase: In the second phase we take into account the measurements, so that we can correct the predicted position.

By using a sampling-based representation, MCL has several key advantages over earlier work in the field [1]:

1. In contrast to existing Kalman filtering based techniques, it is able to represent multi-modal distributions and thus can *globally* localize a robot.

2. It drastically reduces the amount of memory required compared to grid-based Markov localization and can integrate measurements at a considerably higher frequency.

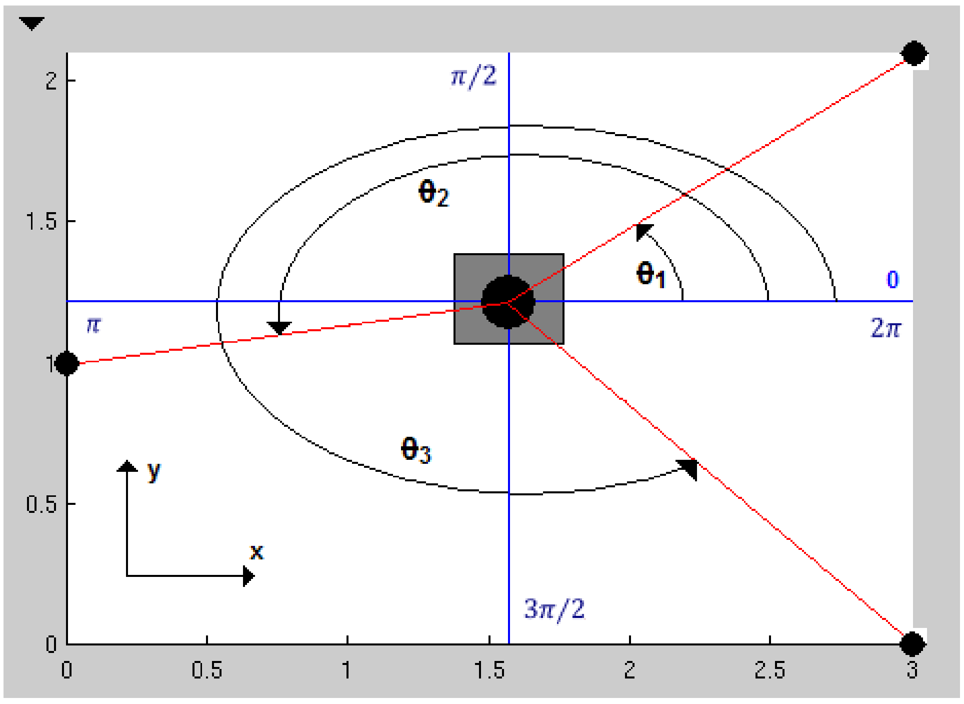
3. It is more *accurate* than Markov localization with a fixed cell size, as the state represented in the samples is not discretized.

4. It is much easier to implement.

1.3 Localization of our robot

The localization of the robot is realized by using one laser beacon, placed on top of the robot and three passive reflectors. These reflectors are covered with tape and are situated in defined positions, around the playground. The position of the robot is estimated using the measured angles between the laser reflections.

The following picture shows the positions of the reflectors and how the angles are measured :



2. MCL Implementation Using Simulink S-Functions

2.1 Introduction to Simulink

Simulink, developed by MathWorks, is an environment for multidomain simulation and Model-Based Design for dynamic and embedded systems. It provides an interactive graphical environment and a customizable set of block libraries that let you design, simulate, implement, and test a variety of time-varying systems, including communications, controls, signal processing, video processing, and image processing.

Add-on products extend Simulink software to multiple modeling domains, as well as provide tools for design, implementation, and verification and validation tasks.

Simulink is integrated with MATLAB, providing immediate access to an extensive range of tools that let you develop algorithms, analyze and visualize simulations, create batch processing scripts, customize the modeling environment, and define signal, parameter, and test data.

Key Features:

- Extensive and expandable libraries of predefined blocks
- Interactive graphical editor for assembling and managing intuitive block diagrams
- Ability to manage complex designs by segmenting models into hierarchies of design components
- Model Explorer to navigate, create, configure, and search all signals, parameters, properties, and generated code associated with your model
- Application programming interfaces (APIs) that let you connect with other simulation programs and incorporate hand-written code
- Embedded MATLAB™ Function blocks for bringing MATLAB algorithms into Simulink and embedded system implementations
- Simulation modes (Normal, Accelerator, and Rapid Accelerator) for running simulations interpretively or at compiled C-code speeds using fixed- or variable-step solvers
- Graphical debugger and profiler to examine simulation results and then diagnose performance and unexpected behavior in your design

- Full access to MATLAB for analyzing and visualizing results, customizing the modeling environment, and defining signal, parameter, and test data
- Model analysis and diagnostics tools to ensure model consistency and identify modeling errors

Creating and Working with Models:

With Simulink, you can quickly create, model, and maintain a detailed block diagram of your system using a comprehensive set of predefined blocks. Simulink provides tools for hierarchical modeling, data management, and subsystem customization, making it easy to create concise, accurate representations, regardless of your system's complexity.

Selecting and Customizing Blocks:

Simulink software includes an extensive library of functions commonly used in modeling a system. These include:

- Continuous and discrete dynamics blocks, such as Integration and Unit Delay
- Algorithmic blocks, such as Sum, Product, and Lookup Table
- Structural blocks, such as Mux, Switch, and Bus Selector

You can customize these built-in blocks or create new ones directly in Simulink and place them into your own libraries.

Additional blocksets (available separately) extend Simulink with specific functionality for aerospace, communications, radio frequency, signal processing, video and image processing, and other applications. [3]

2.2 Overview of S-functions

An *S-function* [4] is a computer language description of a Simulink block. S-functions can be written in MATLAB, C, C++, Ada, or Fortran. C, C++, Ada, and Fortran S-functions are compiled as MEX-files using the mex utility. As with other MEX-files, they are dynamically linked into MATLAB when needed. S-functions use a special calling syntax that enables you to interact with Simulink's equation solvers. This interaction is very similar to the interaction that takes place between the solvers and built-in Simulink blocks.

The form of an S-function is very general and can accommodate continuous, discrete, and hybrid systems. S-functions allow you to add your own blocks to Simulink models. You can create your blocks in MATLAB®, C, C++, Fortran, or Ada. By following a set of simple rules, you can implement your algorithms in an S-function. After you write your S-function and place its name in an S-Function block (available in the Functions & Tables block library), you can customize the user interface by using masking.

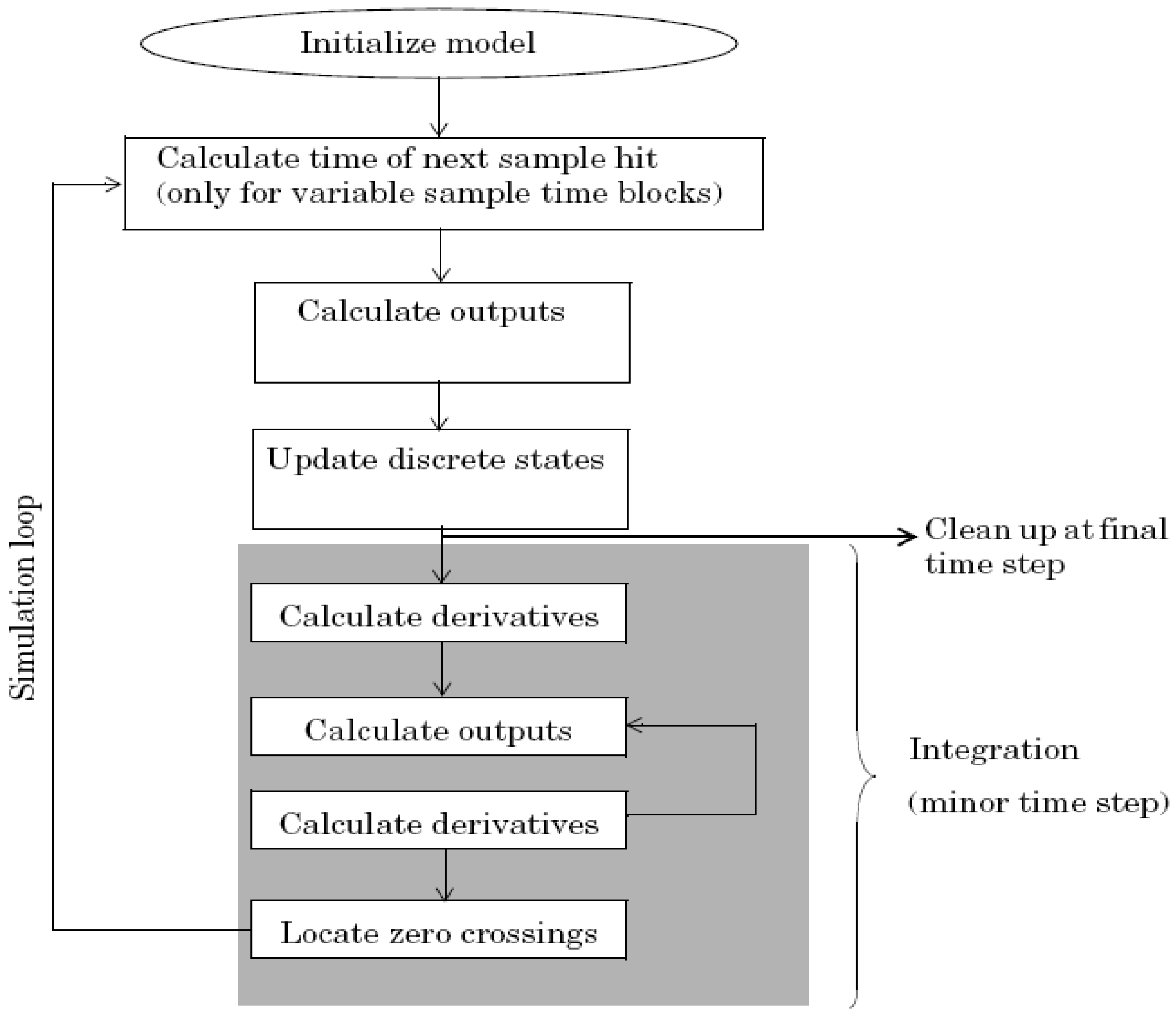
The most common use of S-functions is to create custom Simulink blocks. You can use S-functions for a variety of applications, including :

- •Adding new general purpose blocks to Simulink
- •Adding blocks that represent hardware device drivers
- •Incorporating existing C code into a simulation
- •Describing a system as a set of mathematical equations
- •Using graphical animations (see the inverted pendulum demo, penddemo)

An advantage of using S-functions is that you can build a general purpose block that you can use many times in a model, varying parameters with each instance of the block.

Execution of a Simulink model proceeds in stages. First comes the initialization phase. In this phase, Simulink incorporates library blocks into the model, propagates widths, data types, and sample times, evaluates block parameters, determines block execution order, and allocates memory. Then Simulink enters a *simulation loop*, where each pass through the loop is referred to as a *simulation step*. During each simulation step, Simulink executes each of the model's blocks in the order determined during initialization. For each block, Simulink invokes functions that compute the block's states, derivatives, and outputs for the current sample time. This continues until the simulation is complete.

Stages of a simulation:



S-functions callback methods :

An S-function comprises a set of *S-function callback methods* that perform tasks required at each simulation stage. During simulation of a model, at each simulation stage, Simulink calls the appropriate methods for each S-function block in the model. Tasks performed by S-function methods include :

•Initialization — Prior to the first simulation loop, Simulink initializes the S-function. During this stage, Simulink

- Initializes the SimStruct, a simulation structure that contains information about the S-function
- Sets the number and dimensions of input and output ports

- Sets the block sample times

- Allocates storage areas and the sizes array

•Calculation of next sample hit — If you've created a variable sample time block, this stage calculates the time of the next sample hit; that is, it calculates the next step size.

•Calculation of outputs in the major time step — After this call is complete, all the output ports of the blocks are valid for the current time step.

•Update of discrete states in the major time step — In this call, all blocks should perform once-per-time-step activities such as updating discrete states for next time around the simulation loop.

•Integration — This applies to models with continuous states and/or nonsampled zero crossings. If your S-function has continuous states, Simulink calls the output and derivative portions of your S-function at minor time steps. This is so Simulink can compute the states for your S-function. If your S-function (C MEX only) has nonsampled zero crossings, Simulink calls the output and zero-crossings portions of your S-function at minor time steps so that it can locate the zero crossings.

2.3 The MCL S-Function

To implement the MCL S-Function, I have used some libraries and MCL implementation files, previously developed by other project team members for the Eurobot project.

For S-functions to operate properly, each source module of the S-function that accesses the SimStruct must contain the following sequence of defines and include :

```
#define S_FUNCTION_LEVEL 2
#define S_FUNCTION_NAME  sf_mcl

#include "simstruc.h"
```

These statements give us access to the SimStruct data structure that contains pointers to the data used by the simulation. The included code also defines the macros used to store and retrieve data in the SimStruct. In addition, the code specifies that we are using the level 2 format of S-functions.

The "robomath.h" and "mcl.h" are two Eurobot project headers: "robomath.h" contains useful mathematic functions and "mcl.h" contains different structures and function prototypes required by the MCL algorithm.

```
#include "math.h"
#include "stdbool.h"
#include "robomath.h"
#include "mcl.h"
```

The following definitions are used to provide human understandable names for the inputs and outputs :
INPUT_MOVE, INPUT_MEASURE are labels of the input ports of the MCL S-Function.
OUTPUT_EST_POS, OUTPUT_BITMAP are labels of the output ports.

```
#define INPUT_MOVE 0
#define INPUT_MEASURE 1
#define OUTPUT_EST_POS 0
#define OUTPUT_BITMAP 1
```

BITMAP_WIDTH and BITMAP_HEIGHT define the dimensions of the particle bitmap that estimates the robot’s position.

```
#define BITMAP_WIDTH  300
#define BITMAP_HEIGHT 210
```

Next, we define the human understandable names of the parameters :

```
#define PARAM_PART_COUNT_IDX 0
#define PARAM_NOISE_XY_IDX 1
#define PARAM_NOISE_ANGLE_IDX 2
#define PARAM_AEVAL_SIGMA_IDX 3
```

The following statements define macros to access the values of the parameters and the names for the sample times :

```
#define PART_COUNT(S)   (*mxGetPr(ssGetSFcnParam(S,PARAM_PART_COUNT_IDX)))
#define NOISE_XY(S) (*mxGetPr(ssGetSFcnParam(S,PARAM_NOISE_XY_IDX)))
#define NOISE_ANGLE(S)  (*mxGetPr(ssGetSFcnParam(S,PARAM_NOISE_ANGLE_IDX)))
#define AEVAL_SIGMA(S)  (*mxGetPr(ssGetSFcnParam(S,PARAM_AEVAL_SIGMA_IDX)))

#define SAMPLETIME_MOVE 0
#define SAMPLETIME_MEASURE 1
```

The “state” structure represents the current state of the robot (position and orientation):

```
struct state {
    double x, y, ang; /* [m, m, rad] */
};
```

Every user-written S-function must implement a set of methods, called *callback methods* or simply *callbacks*, that Simulink invokes when simulating a model that contains the S-function. Some callback methods are optional. Simulink invokes an optional callback only if the S-function defines the callback.

2.3.1. The **mdlInitializeSizes** callback method specifies the number of inputs, outputs, states, parameters, and other characteristics of the S-function.

```
/*====================*
 * S-function methods *
 *====================*/

/* Function: mdlInitializeSizes ===============================================
 * Abstract:
 *    The sizes information is used by Simulink to determine the S-function
 *    block's characteristics (number of inputs, outputs, states, etc.).
 */
static void mdlInitializeSizes(SimStruct *S)
{
    ssSetNumSFcnParams(S, 4);  /* Number of expected parameters */

#if defined(MATLAB_MEX_FILE)
    if (ssGetNumSFcnParams(S) == ssGetSFcnParamsCount(S)) {
        mdlCheckParameters(S);
        if (ssGetErrorStatus(S) != NULL) {
            return;
        }
    } else {
        return; /* Parameter mismatch will be reported by Simulink */
    }
#endif

    ssSetNumContStates(S, 0);
    ssSetNumDiscStates(S, 1);

    if (!ssSetNumInputPorts(S, 2)) return;

    ssSetInputPortWidth(S, INPUT_MOVE, 3);
    ssSetInputPortDirectFeedThrough(S, INPUT_MOVE, 1);

    ssSetInputPortWidth(S, INPUT_MEASURE, 3);
    ssSetInputPortDirectFeedThrough(S, INPUT_MEASURE, 1);

    if (!ssSetNumOutputPorts(S, 2)) return;
    ssSetOutputPortWidth(S, OUTPUT_EST_POS, 3);
    ssSetOutputPortMatrixDimensions(S, OUTPUT_BITMAP, BITMAP_HEIGHT,
BITMAP_WIDTH);

    ssSetNumSampleTimes(S, PORT_BASED_SAMPLE_TIMES);
    ssSetNumRWork(S, 0);
    ssSetNumIWork(S, 0);
    ssSetNumPWork(S, 1); /* reserve element in the pointers vector */
    ssSetNumModes(S, 0);
    ssSetNumNonsampledZCs(S, 0);
    ssSetOptions(S, 0);


    ssSetInputPortSampleTime(S, INPUT_MOVE, INHERITED_SAMPLE_TIME);
    ssSetInputPortSampleTime(S, INPUT_MEASURE, INHERITED_SAMPLE_TIME);

}
```

The **ssSetNumSFcnParams** function sets the number of parameters that an S-function expects. In this case, the number of expected parameters is 4 : the number of particles, the standard deviation of the movement distance noise, the standard deviation of the movement angle noise and the value of the measure dispersion.

The following “if” statement checks whether the number of given parameters is the same as the number of expected parameters. If not, it returns an error message.

```
#if defined(MATLAB_MEX_FILE)
    if (ssGetNumSFcnParams(S) == ssGetSFcnParamsCount(S)) {
        mdlCheckParameters(S);
        if (ssGetErrorStatus(S) != NULL) {
            return;
        }
    } else {
        return; /* Parameter mismatch will be reported by Simulink */
    }
#endif
```

The **ssSetNumContStates** and **ssSetNumDiscStates** functions set the number of continuous and discrete states. The MCL block operates with discrete time so we have one discrete state.

The **ssSetNumInputPorts** is used to specify the number of input ports of the S-Function block**.** In our case, the number of input ports is 2, one for the robot movement and one for the measurements. It is invoked by :

```
if (!ssSetNumInputPorts(S, 2)) return;
```

The width of the each input port is set using the **ssSetInputPortWidth** function. For the MCL block, the width is 3 (for the INPUT_MOVE port there is a vector containing the x, y coordinates and the orientation angle, and for the INPUT_MEASURE, a vector containing the values of the 3 beacon angles).

```
ssSetInputPortWidth(S, INPUT_MOVE, 3);
ssSetInputPortDirectFeedThrough(S, INPUT_MOVE, 1);

ssSetInputPortWidth(S, INPUT_MEASURE, 3);
ssSetInputPortDirectFeedThrough(S, INPUT_MEASURE, 1);
```

Both input ports have direct feedthrough (set by the **ssSetInputPortDirectFeedThrough** function), which means that the output is controlled directly by the value of the input port.

For the output ports, the settings are approximately the same.
The number of output ports is 2:

```
if (!ssSetNumOutputPorts(S, 2)) return;
```

The first output port supplies the estimated position of the robot, thus outputing a vector with 3 elements : estimated x, y and angle.

The second port outputs a matrix and the **ssSetOutputPortMatrixDimensions** function specifies dimension information :

```
ssSetOutputPortMatrixDimensions(S, OUTPUT_BITMAP, BITMAP_HEIGHT, BITMAP_WIDTH);
```

Its arguments are : SimStruct representing an S-Function block, the port's name, the Row dimension of matrix signals emitted by port, Column dimension of matrix signals emitted by port.

The **ssSetNumSampleTimes** function is used to specify the number of sample times that an S-Function block has. For this MCL block, the sample times are port-based, meaning that each port can have a different sample time and the block processes a particular port only when a sample hit occurs for that port.

Work vectors are blocks of memory that an S-function asks the Simulink engine to allocate to each instance of the S-function in a model. If multiple instances of the S-function can occur in a model, the S-function must use work vectors instead of global or static memory to store instance-specific values of S-function variables. Otherwise, the S-function runs the risk of one instance overwriting data needed by another instance, causing a simulation to fail or produce incorrect results. The ability to keep track of multiple instances of an S-function is called reentrancy. You can create an S-function that is reentrant by using work vectors that the engine manages for each particular instance of the S-function.

Work vectors have several advantages:

- Provide instance-specific storage for block variables
- Support floating-point, integer, pointer, and general data types, pointers etc.
- Eliminate static and global variables
- Facilitate inlining the S-function during code generation
- Provide more control over how data appears in the generated code

The Pwork vectors are work vectors used for storing pointers. You are allowed only one Pwork vector.

The **ssSetNumPWork** function specifies the number of elements (size) to be allocated to the pointer work vector of the block.

```
ssSetNumPWork(S, 1);
```

For each input port, it must be specified the sample time period. For that, the **ssSetInputPortSampleTime** function is used :

```
ssSetInputPortSampleTime(S, INPUT_MOVE, INHERITED_SAMPLE_TIME);
ssSetInputPortSampleTime(S, INPUT_MEASURE, INHERITED_SAMPLE_TIME);
```

The sample time for both input ports is inherited, meaning that the MCL block has no sample time characteristics of its own.

2.3.2. The **mdlCheckParameters** callback checks the validity of an S-function's parameters. Its argument is S, SimStruct representing an S-Function block. It verifies new parameter settings whenever parameters change or are reevaluated during a simulation. Note: This implementation doesn't check for all errors.

```
#define MDL_CHECK_PARAMETERS
#if defined(MDL_CHECK_PARAMETERS)  && defined(MATLAB_MEX_FILE)
/*
 * Check to make sure that each parameter is 1-d and positive
 */
static void mdlCheckParameters(SimStruct *S)
{
    const mxArray *count = ssGetSFcnParam(S,PARAM_PART_COUNT_IDX);

    if (mxGetM(count) * mxGetN(count) != 1) {
        ssSetErrorStatus(S, "Parameter 1 must be scalar number");
        return;
    }
}
#endif
```

2.3.3. The **mdlProcessParameters** callback is an optional routine that the Simulink engine calls after mdlCheckParameters changes and verifies parameters. The processing is done at the top of the simulation loop when it is safe to process the changed parameters. This function is only valid for simulation. The purpose of this routine is to process newly changed parameters. The parameters can be changed during the runtime of the simulation.

```
#define MDL_PROCESS_PARAMETERS
#if defined(MDL_CHECK_PARAMETERS)  && defined(MATLAB_MEX_FILE)
static void mdlProcessParameters(SimStruct *S)
{
    struct mcl_model *mcl = ssGetPWork(S)[0];
    printf("%s\n", __FUNCTION__);
    /* Tunable parameters was changed during simulation */
    if (mcl) {
        printf("changing mcl\n");
        mcl->mov_dnoise = NOISE_XY(S);
        mcl->mov_anoise = NOISE_ANGLE(S);
        mcl->aeval_sigma = AEVAL_SIGMA(S);
    }
}
#endif
```

2.3.4. The **mdlSetInputPortSampleTime** callback is used to set the sample time of an input port that inherits its sample time from the port to which it is connected.

```
/* Function: mdlSetInputPortSampleTime =========================================
 * Abstract:
 *  When asked by Simulink, set the sample time of the enable or
 *  signal port. If we know both sample times, also set the output
 *  port sample times.
 */
static void mdlSetInputPortSampleTime(SimStruct *S,
                                      int_T     portIdx,
                                      real_T    sampleTime,
                                      real_T    offsetTime)
{
    ssSetInputPortSampleTime(S, portIdx, sampleTime);
    ssSetInputPortOffsetTime(S, portIdx, offsetTime);
}
```

The Simulink engine invokes this method with the sample time that the port inherits from the port to which it is connected.

2.3.5. The **mdlSetOutputPortSampleTime** callback is used to set the sample time of an output port that inherits its sample time from the port to which it is connected.

```
/* Function: mdlSetOutputPortSampleTime ========================================
 * Abstract:
 *  When asked by Simulink, set the sample time of the specified output
 *  port. This occurs when back propagating sample times (see sfuntmpl_doc.c).
 *
 *  When called to set one sample time, we set them all.
 */
static void mdlSetOutputPortSampleTime(SimStruct *S,
                                      int_T     portIdx,
                                      real_T    sampleTime,
                                      real_T    offsetTime)
{
    ssSetOutputPortSampleTime(S, portIdx, sampleTime);
    ssSetOutputPortOffsetTime(S, portIdx, sampleTime);
} /* end mdlSetOutputPortSampleTime */
```

The Simulink engine calls this method with the sample time that the port inherits from the port to which it is connected.

2.3.6. The **mdlInitializeSampleTimes** specifies the sample rates at which this S-function operates.

```
/* Function: mdlInitializeSampleTimes =========================================
 * Abstract:
 *    This function is used to specify the sample time(s) for your
 *    S-function. You must register the same number of sample times as
 *    specified in ssSetNumSampleTimes.*/
static void mdlInitializeSampleTimes(SimStruct *S)

{
    #if 1   /* set to 1 to see port sample times */
    const char_T *bpath = ssGetPath(S);
    int_T        i;

    for (i = 0; i < 2; i++) {
        ssPrintf("%s input port %d sample time = [%g, %g]\n", bpath, i,
                 ssGetInputPortSampleTime(S,i),
                 ssGetInputPortOffsetTime(S,i));
    }

    for (i = 0; i < 1; i++) {
        ssPrintf("%s output port %d sample time = [%g, %g]\n", bpath, i,
                 ssGetOutputPortSampleTime(S,i),
                 ssGetOutputPortOffsetTime(S,i));
    }
#endif
}
```

In this callback, first we use the **ssGetPath** function to get the path of the MCL block. After that, using two "for" statements to set the sample rates for the two input ports and the first output port that estimates the position. The registered number of sample times is the same as the one specified in the **ssSetNumSampleTimes** function from the **mdlInitializeSizes** callback.

2.3.7. The **mdlStart** callback method is an optional method, used for the initialization of the state vectors of this S-function. It is invoked once, at the beginning of the simulation**.** It is the place in which activities that the S-function requires only once (allocating memory, setting up user data, initializing states) are initialized.

```
#define MDL_START  /* Change to #undef to remove function */
#if defined(MDL_START)
  /* Function: mdlStart =======================================================
   * Abstract:
   *    This function is called once at start of model execution. If you
   *    have states that should be initialized once, this is the place
   *    to do it.
   */
static void mdlStart(SimStruct *S)
{
    struct mcl_model *mcl;
    void **pwork;

    mcl = malloc(sizeof(struct mcl_model));
    memset(mcl, 0, sizeof(*mcl));
    pwork = ssGetPWork(S);
    pwork[0] = (void*)mcl;

        /* MCL initialization */
    mcl->beacon_cnt = 3;
    mcl->width = 3.0;   /* in m */
    mcl->height = 2.1;  /* in m */

    /* the noises */
    mcl->mov_dnoise = NOISE_XY(S);
    mcl->mov_anoise = NOISE_ANGLE(S);
    mcl->w_min = 0.25;
    mcl->w_max = 2.0;
    mcl->aeval_sigma = AEVAL_SIGMA(S);
    mcl->maxavdist = 0.150;

    /* amount of particles */
    mcl->count = PART_COUNT(S);

    mcl->parts = (struct mcl_particle *)
    malloc(sizeof(struct mcl_particle)*mcl->count);

    /* phases of the MCL */
    mcl->init = mcl_init;
    mcl->move = mcl_move;
    mcl->update = mcl_update2; /* angles evalution oriented update */
    mcl->normalize = mcl_normalize;
    mcl->sort = mcl_partsort;
    mcl->resample = mcl_resample;

    /* change flag */
    mcl->flag = MCL_RUN;

    /* generate particles with noises */
    mcl->init(mcl);
}
#endif /*  MDL_START */
```

For the MCL S-Function, we declare the *mcl pointer to the “mcl_model” structure and the **pwork pointer to the work vector. After that, we allocate memory space for the “mcl_model” structure and get the MCL block’s pointer work vector, using the **ssGetPWork(S)** function, and store the pointer to mcl_model in it.
Then the MCL model is initialized.
In the MCL initialization, the members of the mcl_model structure pointed to by the *mcl pointer are accessed and initialized.

First, it is set the number of beacons (3), and the size of the playground (3 m X 2.1 m).

```
mcl->beacon_cnt = 3;
mcl->width = 3.0;   /* in m */
mcl->height = 2.1;  /* in m */
```

After, the different noises that occur are initialized from block parameters:

- the movement noises (distance and angle):

```
mcl->mov_dnoise = NOISE_XY(S);
mcl->mov_anoise = NOISE_ANGLE(S)
```

- the minimum and maximum probability used for resampling :

```
mcl->w_min = 0.25;
mcl->w_max = 2.0;
```

- the parameter of likelihood function of measurements :

```
mcl->aeval_sigma = AEVAL_SIGMA(S);
```

The number of particles used for the simulation is then initialized and memory is allocated:

```
mcl->count = PART_COUNT(S);
mcl->parts = (struct mcl_particle *)
              malloc(sizeof(struct mcl_particle)*mcl->count);
```

Next, the phases of the MCL algorithm are initialized :

- the generation of the particles with noises:

```
mcl->init = mcl_init;
```

- the prediction phase (the particles are moved and some noise is added to simulate the real movement) :

```
mcl->move = mcl_move;
```

- the update phase (second method : evaluation of the angles between the robot's head and reflectors):

```
mcl->update = mcl_update2;
```

- the normalization :

```
mcl->normalize = mcl_normalize;
```

- the sorting of particles (first, the weight of each particle is evaluated and after that, the particles are sorted by their weight) :

```
      mcl->sort = mcl_partsort;
```

- the resampling (resamples the particles and eliminates the samples with too low or too high weights).

```
      mcl->resample = mcl_resample;
```

2.3.8. The **mdlOutputs** callback method is a required method that computes the signals emitted by the block. It is invoked by the Simulink engine at each simulation time step.
The method computes the S-function's outputs at the current time step and stores the results in the S-function's output signal arrays.
The tid (task ID) argument specifies the task (sample hit at some input) running when the mdlOutputs routine is invoked.

In the **mdlOutputs** callback two functions are used : **do_move** and **do_measure.**
The first one, **do_move**, is used to predict the position of each particle. During the simulation loop, the input signals for the INPUT_MOVE port are accessed, and an array of pointers (3 elements: x, y and angle) is returned for this port. We use the *iPtrsMove[i] to access each element.

The second function, **do_measure**, is used to update the probability density of estimated position based on measurements. After accessing the input signals for the INPUT_MEASURE port, we use the *iPtrsMeasure[i] to access the elements of the returned array of pointers. If the value of each of the 3 angles is greater than zero, the MCL algorithm continues with the update phase, normalization of particles, sorting and resampling.

```
static do_move(struct mcl_model *mcl, InputRealPtrsType iPtrsMove)
{
    double dx, dy, dangle;
    dx = *iPtrsMove[0];
    dy = *iPtrsMove[1];
    dangle = *iPtrsMove[2];
    mcl->move(mcl, dx, dy, dangle);
}

static do_measure(struct mcl_model *mcl, InputRealPtrsType iPtrsMeasure)
{
    struct mcl_angles angles;
    angles.count = 3;
    int i;
    bool nonzero = false;
    for (i=0; i<3; i++) {
        angles.val[i] = *iPtrsMeasure[i];
        nonzero |= (angles.val[i] != 0);
    }
    if (nonzero) {
        mcl->update(mcl, &angles);
        mcl->normalize(mcl);
        mcl->sort(mcl);
        mcl->resample(mcl);
    }
}
```

The body of the mdlOutputs callback:

```
/* Function: mdlOutputs =======================================================
 * Abstract:
 *    In this function, you compute the outputs of your S-function
 *    block. Generally outputs are placed in the output vector, ssGetY(S).
 */
static void mdlOutputs(SimStruct *S, int_T tid)
{
    struct mcl_model *mcl = ssGetPWork(S)[0];
    struct mcl_particle *parts = (struct mcl_particle *)mcl->parts;

    double *pos = ssGetOutputPortRealSignal(S,OUTPUT_EST_POS);
    double *bitmap = ssGetOutputPortRealSignal(S,OUTPUT_BITMAP);
    InputRealPtrsType iPtrsMove = ssGetInputPortRealSignalPtrs(S, INPUT_MOVE);
    InputRealPtrsType iPtrsMeasure = ssGetInputPortRealSignalPtrs(S,
INPUT_MEASURE);

    int moveTid = ssGetInputPortSampleTimeIndex(S,INPUT_MOVE);
    int measureTid = ssGetInputPortSampleTimeIndex(S,INPUT_MEASURE);

    int i;

    if (ssIsSampleHit(S, moveTid, tid)) {
        do_move(mcl, iPtrsMove);
        }

    if (ssIsSampleHit(S, measureTid, tid)) {
        do_measure(mcl, iPtrsMeasure);
    }

    /* Update probability bitmap */
    for (i=0; i<BITMAP_WIDTH*BITMAP_HEIGHT; i++)
        bitmap[i] = 0;
    for (i=0; i<mcl->count; i++) {
        int x, y;
        double color;
        x = parts[i].x / mcl->width * BITMAP_WIDTH;
        y = parts[i].y / mcl->height * BITMAP_HEIGHT;
        y = BITMAP_HEIGHT - 1 - y;
        if (x>=0 && x<BITMAP_WIDTH && y>=0 && y<BITMAP_HEIGHT) {
            color = 64.0*parts[i].weight;
            color = fmax(color, 16);
            bitmap[x*BITMAP_HEIGHT+y] = color;
        }
    }
 /* Output */
    pos[0] = mcl->est_pos.x;
    pos[1] = mcl->est_pos.y;
    pos[2] = mcl->est_pos.angle;
}
```

First, we access the elements of the signals connected to the two output ports, and for that the **ssGetOutputPortRealSignal** function is used.
Next, the access of the signal elements connected to the two input ports, using the **ssGetInputPortRealSignalPtrs** function. This function returns a pointer to the elements of the signal connected to each port.

The “moveTid” and “measureTid” variables are used to store the sample time index of the two input ports. The sample time index is obtained by using the **ssGetInputPortSampleTimeIndex** function.

The two “if” statements verify whether the sample time is hit or not for each of the input ports. The **ssIsSampleHit** returns a boolean value. If this value is “true”, the simulation is executing in the task represented by task ID.

Further more, in this callback, it is implemented the update of the probability bitmap.
The position of each particle with a high associated likelihood is updated at each time step, thus a cloud of particles that represents the estimated position of the robot is formed.

2.3.9. The **mdlTerminate** callback method performs any actions required at the termination of the simulation, such as freeing memory.

```
/* Function: mdlTerminate =====================================================
 * Abstract:
 *    In this function, you should perform any actions that are necessary
 *    at the termination of a simulation.  For example, if memory was
 *    allocated in mdlStart, this is the place to free it. */
static void mdlTerminate(SimStruct *S)
{
    struct mcl_model *mcl = ssGetPWork(S)[0];
    free(mcl);
    ssGetPWork(S)[0] = NULL;
}
```

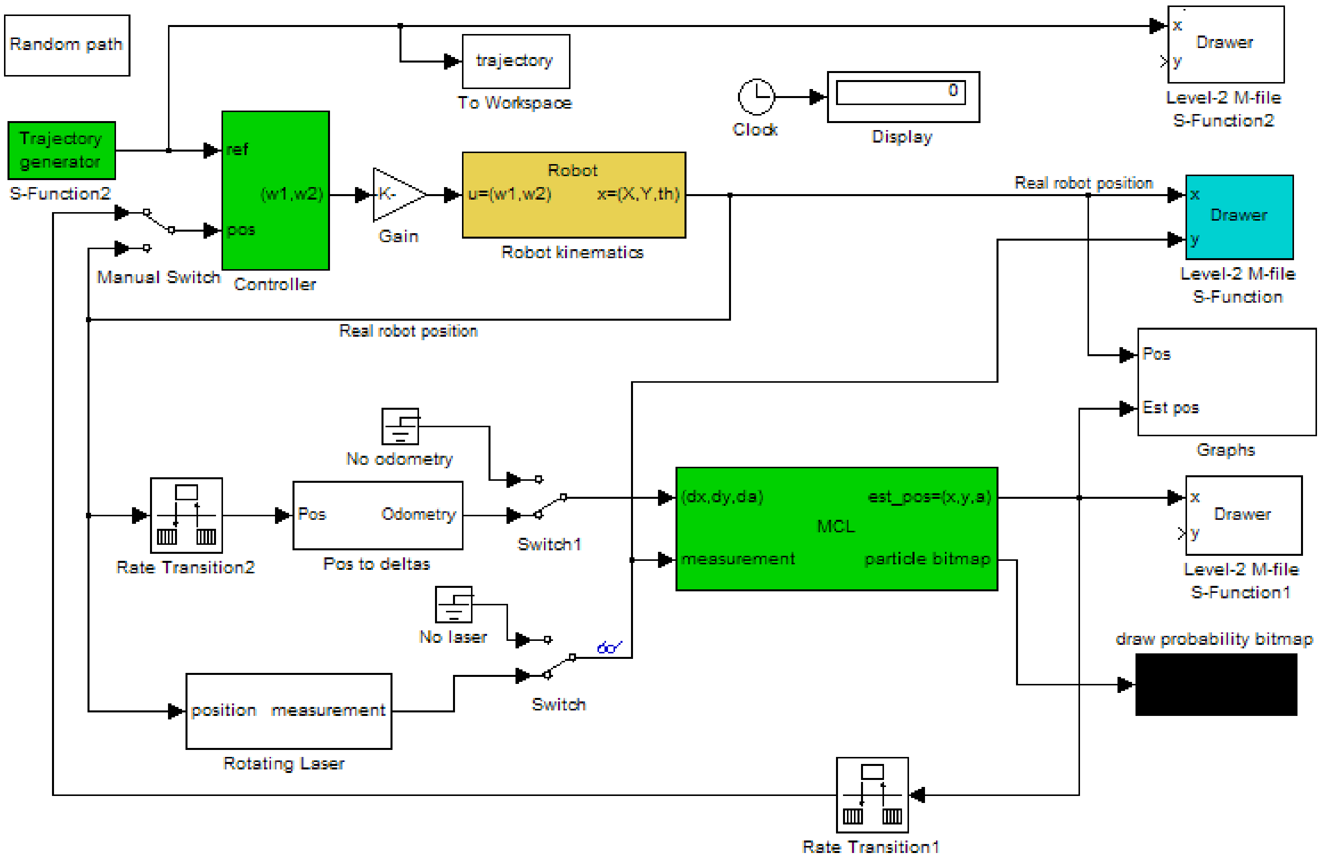
The S-Function code terminates with a required trailer :

```
#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
```

3. Integration of the MCL block into the robot motion model

To integrate the MCL S-function into the Simulink model, previously developped by other team members, we use the S-Function block from the User Defined Functions library. This block provides access to S-Functions from the block diagram. It allows additional parameters to be passed directly to S-function. For the MCL block, these parameters are : the number of samples, the movement noise X,Y, the movement noise angle and the measure dispersion.

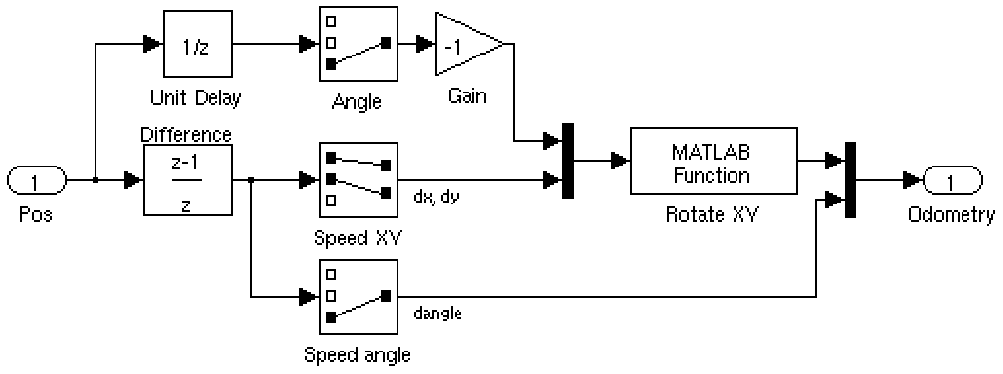
3.1 The model for the robot motion. The “Robot kinematics” supplies the coordinates of the real robot position: X, Y and Theta. These coordinates are used by the “Pos to deltas” block and “Rotating laser” block.



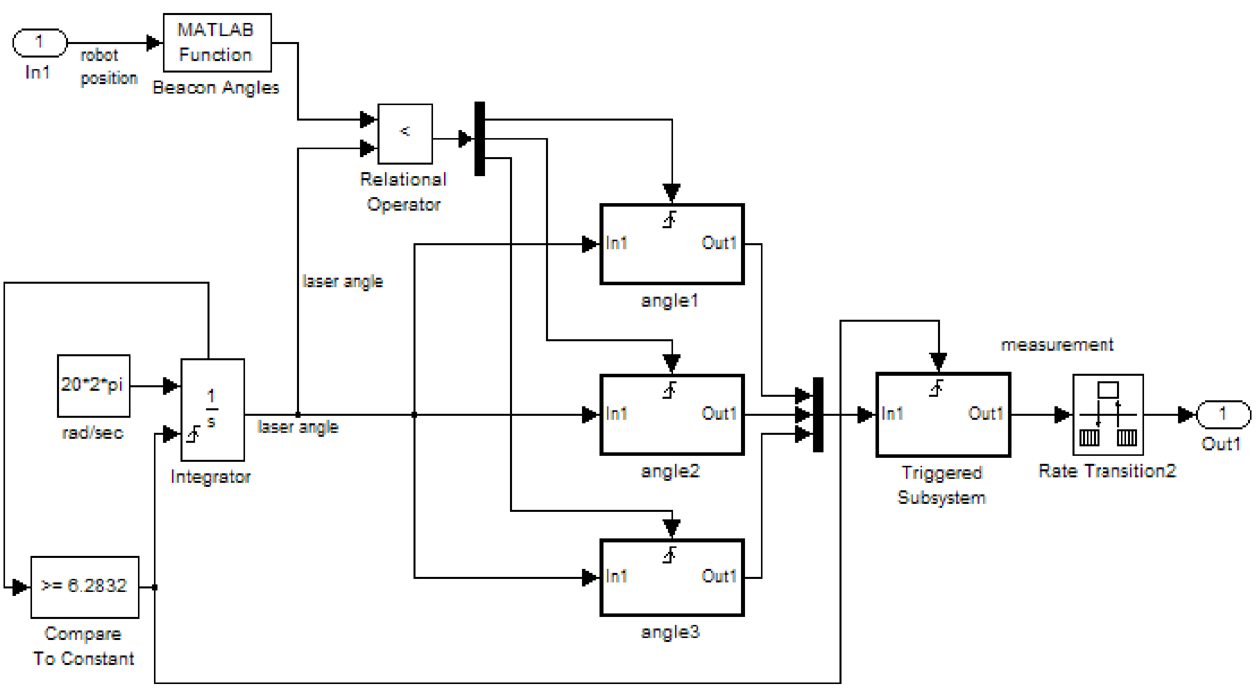
3.2 The “**Pos to deltas**” block :

Implemented in Simulink by using a subsystem, it is required to transform the coordinates from the global coordinate system to the robot’s coordinate system. The difference block reprezents the increment of movement. The value of the angle remains the same, so the values of x and y are the ones to be modified. For that, we need to rotate them using the “Rotate XY” function, which is the basic transformation matrix used for rotation :

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ +\sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}.$$



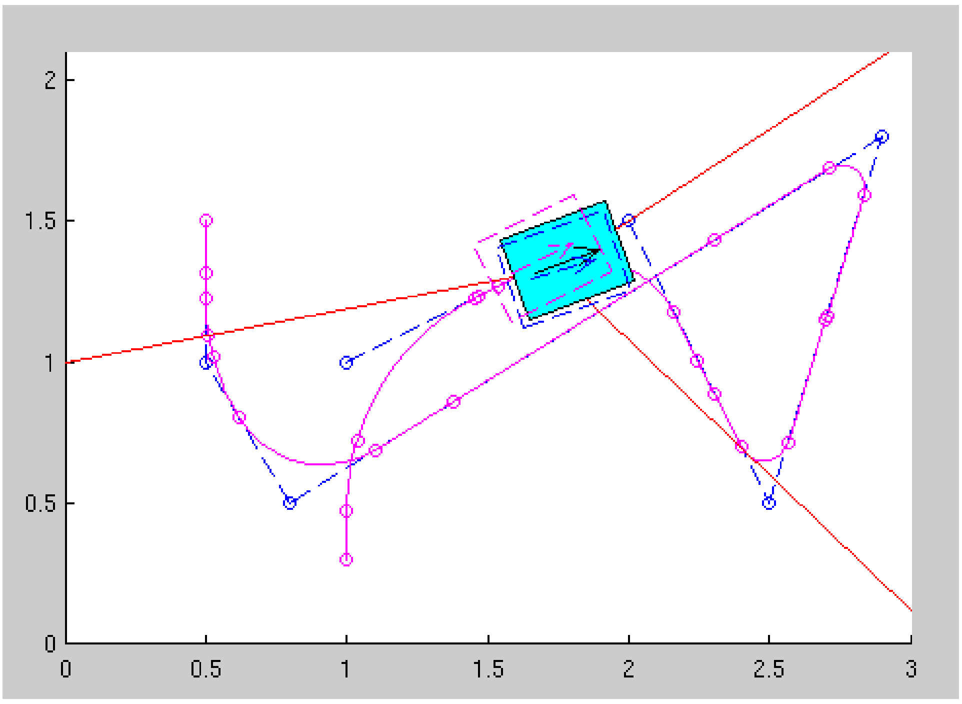
3.3 The “**Rotating laser**” block :



The angles between the laser reflection beams are not all measured at the same time, but one at a time because the laser beacon placed on the robot rotates at finite speed. So the

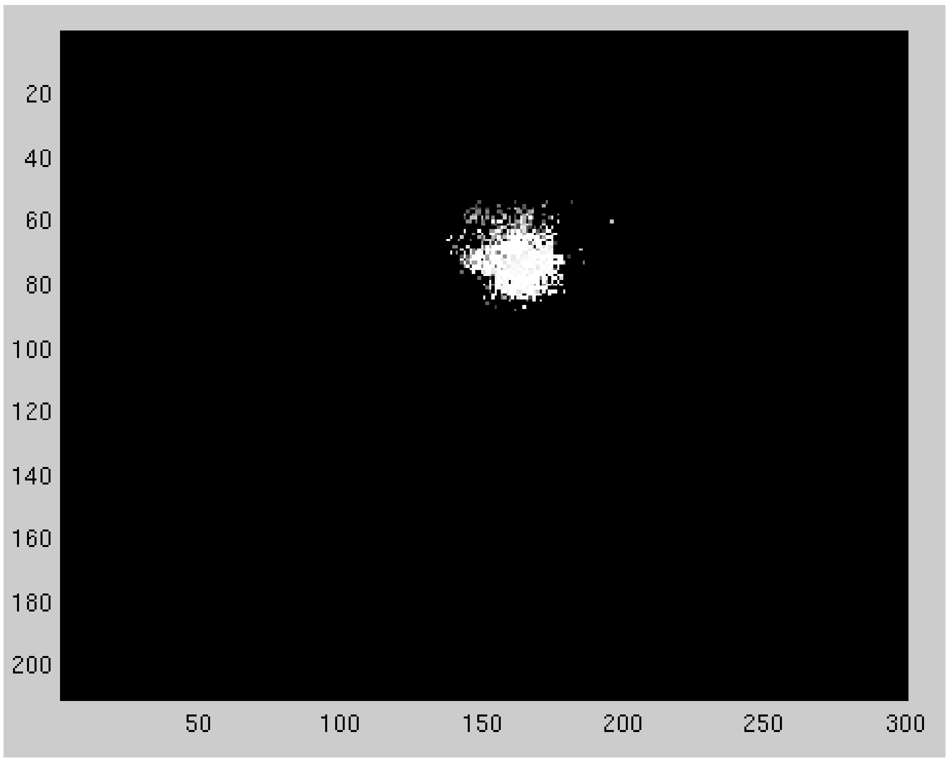
value of each of these angles is to be processed only after the beacon performs a complete rotation. This might result in some errors due to the the fact that the robot’s position might change before the measured data is sent to be processed.
The input for the “Beacon Angles” function is the real position of the robot. Using the coordinates of this position, the function calculates the three laser angles. The integrator block is used to model the rotational speed of the laser beacon.
Each measured angle is stored into a “memory” module. We use three “memory” modules, one for every angle. Each of these modules is actually a triggered subsystem. The external input that triggers the subsystem is the result of the comparison between the calculated angles and the angle of the laser. On every match, the stored values are sent to another triggered subsystem. On every complete rotation of the laser beacon, the stored data of this subsystem is outputed, to be used by the MCL block, to correct the robot’s position.

The next picture shows a frame from the MCL simulation:



The cyan box represents the real position of the robot. The robot moves on the trajectory (the magenta line). The dotted blue rectangle shows the estimated position of the robot, calculated via the MCL block.

The next picture reprezents the probability bitmap. It shows the cloud of particles that estimate the current position of the robot. You can observe the different shades of grey, corresponding to particles with higher or lower weight.



4. Conclusion

By using Monte Carlo type methods, we have combined the advantages of grid-based Markov localization with the efficiency and accuracy of Kalman filter based techniques.
As with grid-based methods, we are able to represent arbitrary probability densities over the robot's state space.
Therefore, the MCL method is able to deal with ambiguities and thus can *globally* localize a robot. By concentrating the computational resources (samples) on the relevant parts of the state space, this method can efficiently estimate the position of the robot.
Unfortunately, this model has some weaknesses. Given that the laser beacon measures one angle at a time and sends the three values only after a complete rotation, the processed data is old and might not correspond to the actual position of the robot.
An improvement would be increasing the rotation speed of the laser beacon, but that depends very much of the robot's hardware structure.

References:

[1] Dieter Fox, Wolfram Burgardy, Frank Dellaert, Sebastian Thrun, *“Monte Carlo Localization: Efficient Position Estimation for Mobile Robots”,* in *Proc. of the Sixteenth National Conference on Artificial Intelligence (AAAI'99)*, 1999.

[2] Sebastian Thrun, Dieter Fox, Wolfram Burgard, and Frank Dellaert, *“Robust Monte Carlo Localization for Mobile Robots”*, in *Artificial Intelligence*, 2001.

[3] *Simulation and Model-Based Design*: http://www.mathworks.com/products/simulink/

[4] *Simulink :Model-Based and System-Based design* : The MathWorks, Inc