Position Recovery from Accelerometric Sensors
Algorithms analysis and implementation issues

Antonio Filieri, Rossella Melchiotti

Dipartimento di Elettronica e Informazione, Politecnico di Milano, Milan, IT
Email: {antonio.filieri, rossella.melchiotti}@mail.polimi.it;

Abstract

The goal of this project is to conceive a system for estimating the position of an agent using accelerometers. We explore the feasibility of this kind of system paying particular attention to the evaluation of the accuracy obtainable.
We analyze different solutions based on numerical integration techniques and Kalman filtering. Some complexity and functional issues are discussed along the report. A C implementation of one of the algorithm is proposed for an Atmel AT90CAN128 microcontroller with a LIS3L02AL MEMS sensor.

Results:

- Extended error sources identification and profiling of noise properties.
- Error propagation through integral operators evaluated via simulation.
- Kalman filtering was applied to reduce accumulation of position uncertainty.

Contents

1 Introduction

Positioning based on inertial sensors is a spin-off of robotics navigation field that is getting an increasing importance nowadays. The new interest is due to the introduction of low-cost MEMS devices.
Those small chips are today integrated in a lot of mobile devices, like phones and PDA. MEMS are small, and cheap and their application field include GPS-recovery, pedestrian positioning, drift compensation, micro movements detection.
MEMSs provide acceleration information that can be used to reconstruct position movements of an agent. Inertial sensors have well known limitations in term of accuracy, because even a minimal error in sensed accelerations may lead to a significant displacement in position because of the double integration. Those, INS were typically used in pair with other sensors (e.g. GPS, optical, gyroscopes) in order to improve the global accuracy.
But in many situations there's no need for so much accuracy: approximative localization of pedestrians inside a building or a tunnel, location-based service providing, short time GPS-recovery, naval positioning are just some common examples of such situations.
The goal of this project is to conceive a system for estimating the position of an agent in a 3D space using accelerometers. Our work is mainly concerned with error sources identification and modeling and integration algorithms for accuracy analysis.

This report is organized as follows: in the next section we provide two different formalizations of the problem; then in section 3 we present an analysis of the main error sources. In section 4 we show some possible solutions and then in section 5 simulations give an insight on methods accuracies. In section 6 we discuss some basic implementation issues and in section 7 there are some final consideration.
In appendices some information concerned the position reference system and the gyroscopes used for orientation are presented; code used for simulation and implementation is in appendices too.

2 Problem analysis and modeling

The goal of the system is to recover the position of an agent sensed by accelerometers in a 3D cartesian space. Restricting the scope to a single axis, the procedure of position extraction with respect to time, can be see as a double integration of acceleration function over time:

$$\int_0^{\bar{t}} \int_0^{\bar{t}} a(t)\ dt\ dt$$

where $a(t)$ is the acceleration.
In order to estimate position using a digital device it is then necessary to use numerical integration. In the next paragraph we will discuss the basic principles of numerical integration; a deeper insight on the trapezoidal method (which is one of the most used algorithms for numerical integration) will be presented later and simulated in a case study.
After a short introduction to numerical integration, an alternative formulation of our problem is proposed. The problem is conceptualized as a linear system which is a suitable representation for using powerful processing techniques coming from control theory. Among them we will present Kalman Filter in section

4.3. The linear system form may be conveniently used to explore the application of automation and control theory to improve position recovery performances.

2.1 Numerical integration

The purpose of numerical integration is the approximate computation of an integral using numerical techniques. Before presenting some useful techniques that can be deeper explored and adapted, we review two basic theoretic ideas behind the approximation of integral functions by discrete samples.

Riemann sum . Let f be defined on the closed interval $[a, b]$, and let Δ be an arbitrary partition of $[a, b]$ such as: $a = x_0 < x_1 < ... < x_{n-1} < x_n = b$, where Δx_i is the length of the ith subinterval.
Taken a sequence of points $c_i \in \Delta x_i$, the sum:

$$\sum_{i=0}^{n} f(c_i)\Delta x_i \ , \ x_{i-1} \leq c_i \leq x_i$$

is called the Riemann sum of the function f for the partition Δ on the interval $[a, b]$. For the partition Δ, the length of the longest subinterval is called the norm of the partition, and it is represented by $\|\Delta\|$.
The (definite) integral operator can be defined, starting from the Riemann sum, by the following limit:

$$\lim_{\|\Delta\|\to 0} \sum_{i=0}^{n} f(c_i)\Delta x_i \ = I$$

If this limit exists the function is said to be integrable, and the Riemann sum of f on $[a, b]$ approaches the number I.
Riemann formulation has a lot of implications outside the scope of our work. It is anyways immediate to identify the strong support the Riemann sum provides in our context: each sample can be seen as a reading taken in the time point c_i belonging to a non-overlapping sub-interval of the observation time.
The basic approximation of the interval function is straightforward by considering samples as $f(c_i)$ values and sampling period as Δx. It also protects us against small delays in sampling procedure and ADC, because no assumptions are made on the position of c_i inside Δx_i.
This sentence is a rough definition of the quadrature method that consists in approximating the curve as a sequence of rectangles with base Δx and heights $f(c_i)$.
Before analyzing the most common numerical integration methods, it is important to notice that as Δx gets larger, the choice of $f(c_i)$ acquires importance.

Mean value theorem The last theoretic note concerns the possibility to find a rectangle whose area is exactly equal to the area of the region under the curve. This rectangle exists under certain circumstances, as stated by the mean value theorem for integrals, that is, if the function f is continuous on the closed interval $[a, b]$, then there exists a number c in $[a, b]$ such that:

$$f(c) = \frac{1}{x_{i+1} - x_i} \int_{x_i}^{x_{i+1}} f(t)dt$$

$f(c)$ is the height of the rectangle with base Δx_i having area exactly equal to the integral of the function f on the same interval.

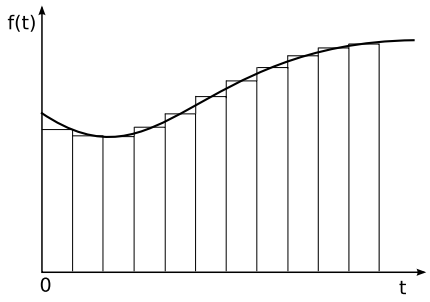


Figure 1: Function area approximation via rectangles.

We introduced the two theoretical notes in order to provide a fast validation of the procedure and to facilitate a deeper reasoning about different integration techniques.
Real-life methods use further approximation procedures in order to reduce the integration error. There are two main classes of numerical integrations, with different application cases.
The most straightforward procedures use the Newton-Cotes formulas ([6]) which approximate a function sampled periodically by polynomials of various degrees (the basic quadrature that can be inferred from Riemann sum definition is the approximation of the function by a 0 degree poly). In order to get an n-degree polynomial approximation, n+1 samples are required. The two point approximation (first order poly, or line) is called trapezoidal rule; three points are used by Simpson's rule and a 5-points formula is used by Boole's rule.
There exists also a generalization of the trapezoidal rule, known as Romberg integration, which can yield accurate results for fewer function evaluations.
The second large class of methods are based on the so called Gaussian quadrature ([8]). Even if they produce the most accurate approximations, they can apply only if the analytical form of the function f is known. Unfortunately, this is not the case.

2.2 Linear system

We are describing the movement of an agent as a stateful process. In a stateful process the response of the system is related to both the user's input and the system history, which means that the same input in different states may lead to different responses. Under the markovian assumption, the entire history of the system is condensed for each time point in its state vector. Thus, to compute the next state, user input and current state suffice.
Transformation between process states is represented by the following linear stochastic difference equations:

$$\begin{aligned} &\text{State equation:} && x_{k+1} = Ax_k + Bu_k + w_k \\ &\text{Output equation:} && y_k = Hx_k + z_k \end{aligned}$$

In the state equation the state vector x evolves during one time step by premultiplying by the state transition matrix A. u is an input vector which can eventually be zero. If non-zero, it affects the state

evolution linearly as described by the input matrix B. In the state equation appears also a gaussian process noise w, which is related to the ability to estimate the system state.
The output equation describes the observable part of the system. The observation vector y is a linear function of the state vector, and this linear relationship is represented by premultiplication by the observation matrix H. Also in the case of observation a non-null gaussian measurement noise v could be added.
The entire system is based on the markov assumption that the state x_{k+1} only depends on its previous state x_k.
The state of a moving agent can be described by at least its position and its velocity.
Consider the sampling period Δ as the length of each time step. For a short enough time step, the acceleration can be approximated as a constant a. Thus, integrating we get the velocity and the position laws as follows:

$$v_{k+1} = v_k + \Delta\ a_k$$

$$p_{k+1} = p_k + \Delta\ v_k + \frac{1}{2} a_k \Delta^2$$

We choose to include in our system state position, velocity and acceleration on the three axis.
Our observation matrix is the identity matrix and the observation vector has the same length of the state vector but non-null values occur only for the accelerations read from the sensor. The matrix A is easily constructed from position, velocity and acceleration equations (see section 4.3). Since our system has no user input both B and u are null.
We want to recover the agent position at any time point k and from this representation we want to get an estimation of the state x.
Before showing possible solutions that apply both numerical integration and Kalman filtering, we propose an analysis of error sources involved in acceleration sensing.

3 Error sources

In the proposed architecture there are many sources of errors that can reduce the accuracy of the results. Some errors are related to environment conditions (like temperature, magnetic fields, etc) others to sampling and processing. The former can be reduced by adding more sensors and integrating their information as in [20] and [16].

3.1 MEMS error

An accelerometer is a device used to measure non-gravitational accelerations in terms of g-force (A beginner's guide to accelerometers s.d.). For this particular application we used the LIS3L02AL MEMS (Micro-Electro-Mechanical System).
We have multiple errors related to these devices. First of all from the datasheet it is clear that sensitivity is not constant but varies between Vdd/5 – 10% and Vdd/5+ 10%. Sensitivity also varies with temperature by $0.01\%/^{\circ}$C from the reference temperature of + 25°C. Even zero-g level is not constant but varies

between Vdd/2 -6% and Vdd/2 + 6%. Moreover zero-g level changes by $\pm$0.5mg/$^\circ$C from the reference temperature of $+25^\circ$C.
The signal from the accelerometers is not noise free but has a noise density of 50 μm/sqrt(Hz).
There are also non-linearity errors varying from $\pm$ 0.3% to $\pm$1.5% for x and y axis and from $\pm$0.5% to $\pm$1.5% for z axis (MEMS inertial sensor: 3-axis - +/-2g ultracompact linear accelerometer s.d.).
This strongly affects the quality of our acceleration measures. For example an error of -6% on zero-g level can transform a zero acceleration on x into an acceleration of -1.47 m/s2 which can lead to huge errors in position estimation after few seconds.
As figure 2 shows, zero-g level and sensitivity can vary greatly from device to device.
The best way to solve the problem of sensitivity and zero-g level varying from device to device is to calibrate our accelerometer when the application starts in order to get the correct values of these parameters. It is much more difficult to correct for the variations due to changes in temperature unless we provide the system with a way to measure temperature feedback or with an efficient thermal isolation (so that the temperature remains constant). Considering a mobile deployment, thermal isolation is quite unfeasible.

3.1.1 Tilt orientation error

Another significant error which affects this kind of devices is the tilt in orientation. At 0g this error becomes really significant. A 1° tilt in the 0g position creates an output error equivalent to a10° tilt in the +1g or -1g positions.
This can be clearly seen from the following figure 3.
We can then infer that at 0g orientation, change in 1° tilt causes 59 x (0.9998/1.7e-2) bigger changes in sensor output versus -1g or +1g orientation (Instruments s.d.).
From the graph in figure 4 shows how the effect of tilt can greatly impact the output of the accelerometer and consequently all the position estimations based on this datum.

3.2 Errors related to sampling frequency

In order to avoid distortion in the reconstruction of the analogical signal sampled it is necessary to respect the conditions of the Nyquist - Shannon theorem which states that if the maximum band of a signal is B then the sampling frequency must be at least equal to 2B (Unser 2000).
The bottleneck in the sampling frequency of our device is related to the microcontroller which must perform a AD conversion before taking a new sample.
Conversion from analogical to digital takes 13 clock cycles on the microcontroller AT90CAN128. Our algorithm takes a sample from x acceleration, a sample from y, a sample of z and starts again with x. The distance between two subsequent samples on each channel is then equal to 13*3 clock cycles. Since our device works at a frequency of 4MHz we can implement a maximum sampling period of 13*3/(4MHz)=9.75μs (102KHz).
In reality this sampling rate is too optimistic since it doesn't take in account all the cycles spent for calling the routines, setting the variables, computing the sums, etc.
Simulating the algorithm using AVR (one of the most used IDE for Atmel microcontrollers) we find that the number of cycles between two subsequent readings of the same variable is 3370 which corresponds to a sampling period of 842.5μs (1.19KHz).

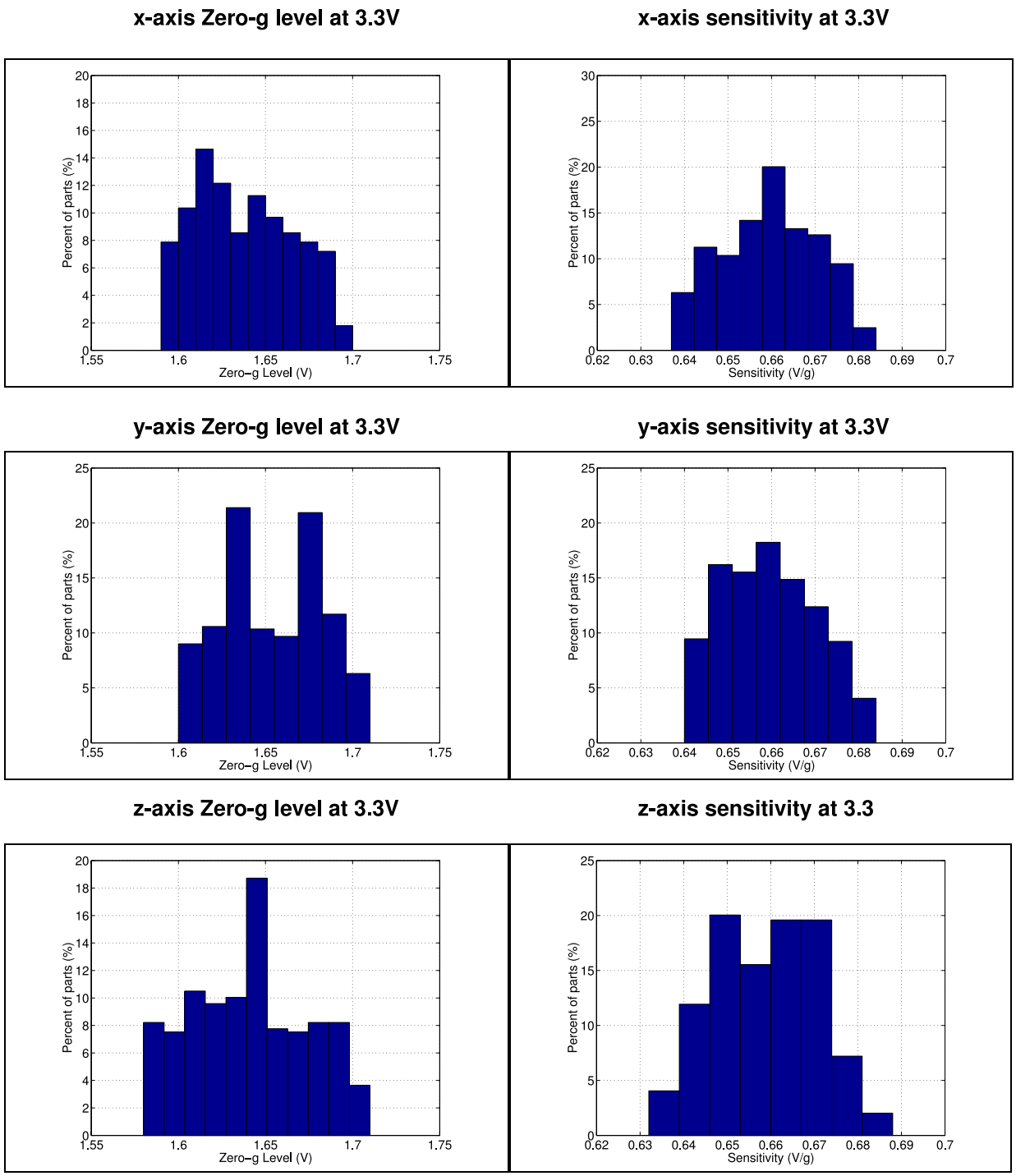


Figure 2: LIS3L02AL mechanical characteristics at 25°C

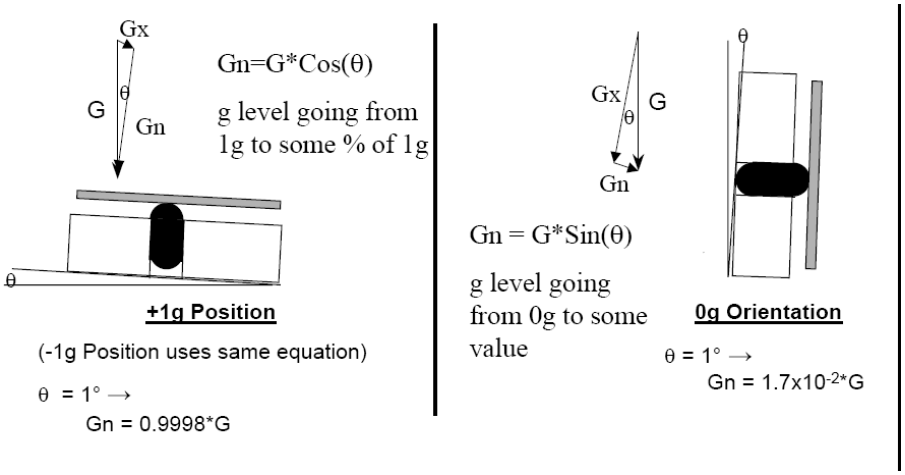


Figure 3: Tilt orientation error

The three Sample and Holders of the accelerometer, corresponding to the three output accelerations channels, operate at a sampling frequency of 66kHz. Since we can at most sampling the analogical input at a frequency of 1.19KHz the maximum frequency of the acceleration input we can accept is of fsampling/2=1.19KHz/2=595Hz (Unser 2000).
If our acceleration has a frequency higher than 595Hz the original signal will be distorted by the sampling procedure. It is then necessary to filter the input signal to be sure that the Nyquist theorem is respected. This can be easily done using a RC filter.

3.3 Errors related to the AD conversion

The Analogical-Digital conversion introduces multiple errors. An n-bit single-ended ADC converts a voltage linearly between GND and VREF in 2n steps (LSBs). Since the AD converter integrated in our microcontroller (AT90CAN128) uses 10 bits we can convert our input (the analogical acceleration) to a value between 0 and 1023 ($2^{10} - 1$).
We have then a quantization error of $\pm$ 0.5 LSB, due to the fact that a range of input voltages (1 LSB wide) will code to the same value. In our particular case 1LSB corresponds to 0.0025V (2.56V / 1024 where 2.56V is the reference voltage). Another typical error typical of this kind of devices is the offset error which is the deviation of the first transition (from 0x000 to 0x001) from the ideal transition (which should be at $+$0.5 LSB).
In order to obtain the wanted precision it is necessary to correct this error in the application.
Another significant error, that requires a correction in the application, is the gain error. The gain error is defined as the deviation of the last transition (from 0x3FE to 0x3FF) compared to the ideal transition (which should be at 1.5 LSB below the maximum).

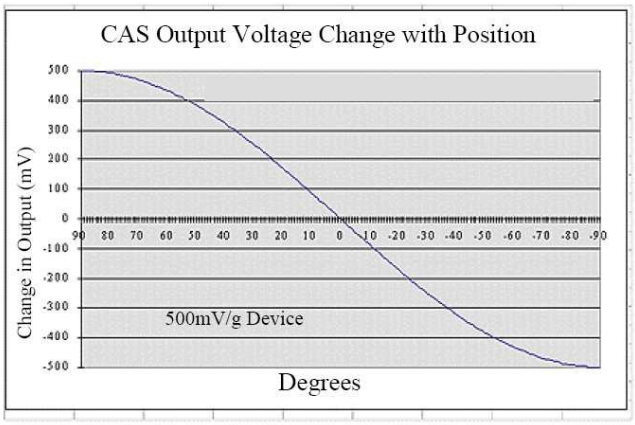


Figure 4: Output effect of the tilt error

3.3.1 Non-linearity

Relevant errors can be due also to non-linearity in the converter. The integral non-linearity is by definition the maximum deviation of an actual transition compared to an ideal transition for any code.
A similar error is caused by the differential non-linearity which is the maximum deviation of the interval between two adjacent transitions from the 1LSB ideal code width.
While it is possible to correct for offset error, gain error and non-linearity, the quantization error can only be reduced increasing the resolution of the converter (8-bit Microcontroller with 128K Bytes of ISP Flash and CAN controller s.d.) . In our particular case, the input analogical signal varies between 0 and 3.3V. In order to convert it using the reference voltage of the microcontroller (2.65V) the input must be scaled to the reference voltage. The quantization error is going to be 2.65V/(1024)=0.0025V which corresponds to a 0.00322V error with respect to the MEMS voltage reference (LIS3LO2AL, voltage reference=3.3V) and to an error in acceleration of 0.00488g (0.048 m/s2).

3.4 Errors related to the resolution of the microcontroller

In our microcontroller (AT90CAN128) double precision numbers are represented by a 32-bit variable. In this particular case 1 bit is used for the sign, 8 bits are used for the exponent and 23 bits for the mantissa. Since the number is represented with finite precision we have necessarily some inaccuracies when performing operations with floats. These inaccuracies are negligible with respect to the errors introduced by the other components of the device.

4 Proposed solutions

In this section we propose three different solutions based on the previous analysis.
The first one will consider numerical integration by directly applying the trapezoidal rule. Then, a simple low complexity numerical integration algorithm is analyzed. Finally a Kalman filter solution is explored.

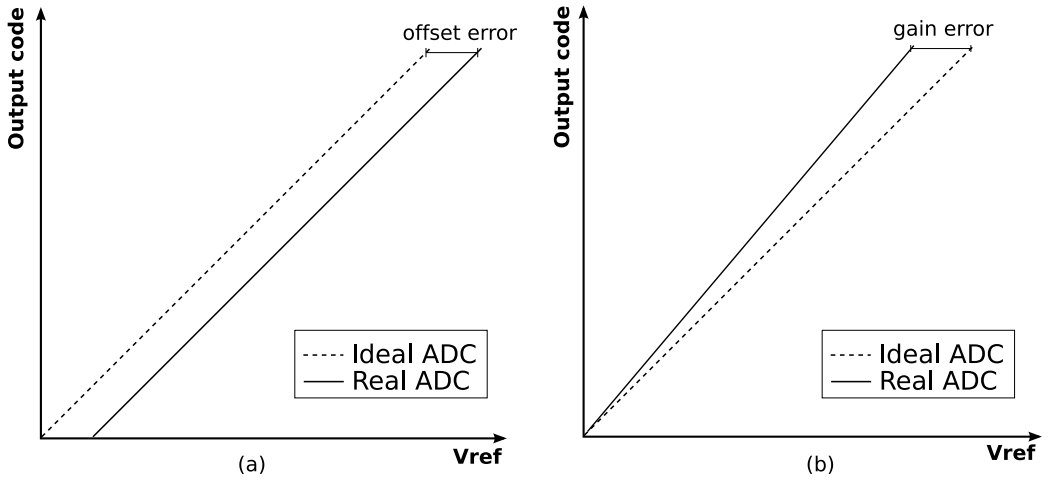


Figure 5: Errors due to ADC: (a) offset error, (b) gain error.

4.1 Numerical integration

The method considered for numerical integration is the trapezoid one. There isn't an integration technique which performs better than any other in any situation but every algorithm performs better than the others in some contexts and worse in others.

The trapezoid rule is an interesting example because of its flexibility. The principle of trapezoidal integration is simple, if $[a, b]$ is the interval over which we want to integrate our curve and n is the number of segments we want to consider then:

$$\begin{aligned} h &= \tfrac{b-a}{n} \\ I &= \textstyle\int_a^b f(t)\ dt \\ &= \textstyle\int_a^{a+h} f(t)\ dt + \int_{a+h}^{a+2h} f(t)\ dt + \ \ldots\ \int_{a+(n-2)h}^{a+(n-1)h} f(t)\ dt + \int_{a+(n-1)h}^{b} f(t)\ dt \end{aligned}$$

Approximating each of the integrals with a trapezoid we can compute the integral of the function $f(t)$ between a and b using the following formula:

$$\int_a^b f(t)\ dt \ \approx\ \frac{b-a}{n}\left(\frac{f(a)+f(b)}{2} + \sum_{k=1}^{n-1} f\left(a + k\frac{b-a}{n}\right)\right)$$

An application of this algorithm can be seen in the following figure 7.

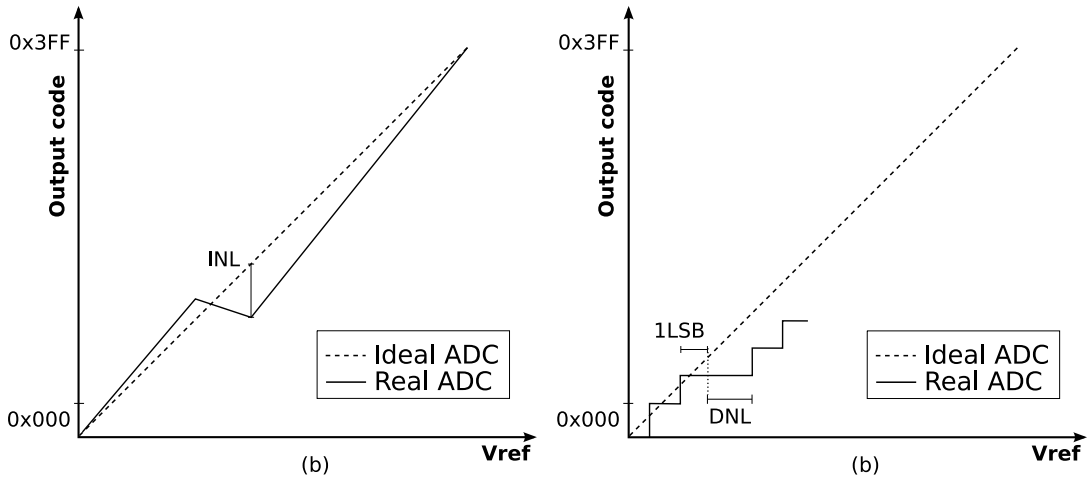


Figure 6: Errors due to ADC: (a) integral non-linearity, (b) differential non-linearity.

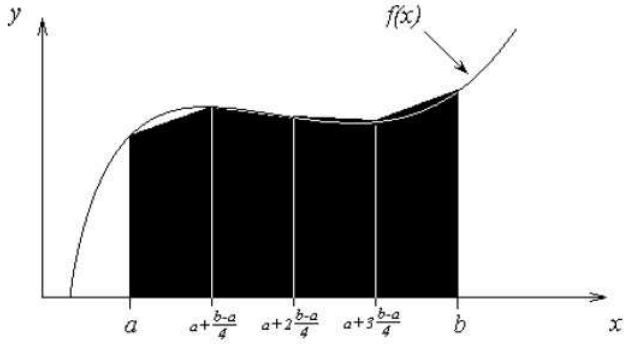


Figure 7: Trapezoid integration example.

If the function we are integrating is bounded then the maximum error we can obtain is given by the following formula:

$$|e_T| \leq \frac{M(b-a)^3}{12n^2}$$

Where (b-a) is the integration interval, M is the max of the function we are integrating and n is the number of segments we are considering (Numerical Integration s.d.).
If we consider an integration interval in the order of 20s (length of a small tunnel) and we know that our acceleration function can at maximum be equal to 2g (19.62 m/s) and we consider taking a sample every second we get a maximum error on the speed in the order of 32.7 m/s. If we consider a sample every 1ms we get a maximum error on the speed in the order of 0.0000327 m/s.

Since we need to integrate twice errors gets even greater. It is evident from this estimation that the integration step must be as little as possible in order to minimize error.

4.2 Trapezoid-like solution

A simple and efficient trapezoid-like approximation is proposed in [18]. The idea is to reduce errors of integration with a first order approximation.
Essentially, instead of using a fast but inaccurate rectangular approximation, Seifert and Camacho propose to consider for each time step two small areas:

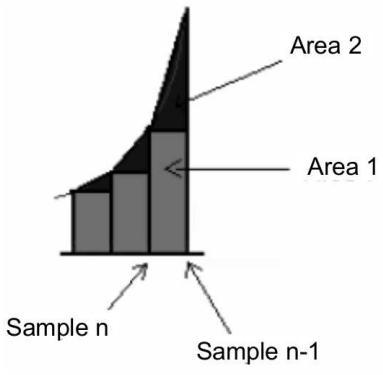


Figure 8: Two areas approximation.

To compute the area of sample k, the first area to be considered is a rectangle with base Δ and height $f(k-1)$ while the second is a triangle which area is computed as $\frac{f(k)-f(k-1)}{2}$. Essentially it is a 2-point interpolation, that reduce the error on the base of the trapezoid principle.
The area is then:

$$I_k = f(k)dt \ + \ \frac{f(k) - f(k-1)}{2}dt$$

This formula is quite fast and shares the properties already stated for trapezoidal rule, thus we choose it as the numerical integration candidate solution to be evaluated through simulation.

4.3 Kalman filter

The Kalman filter is an algorithm able to handle the separation of probabilistic noise from the state estimation of the system. It is an adaptive discrete recursive estimation algorithm [12].
Provided the structure of the linear system (state and output equations) is known, the idea is to define an estimator that gives a good estimation of the state. Notice that in the model proposed in section 2.2 the time becomes intrinsic to the model, there is not user input too, and the relation between observation vector (accelerations) and state vector is linear. Hence the entire system is linear.
Kalman theory comes from the probabilistic modeling field, and allows to build statistical estimators.
The first requirement for the estimator is that its expected value is equal to the expected value of the true state. This requirement provides an estimator not biased toward any improver value.

The second requirement the estimator must satisfy is to minimize the error variance.
Under certain conditions, the Kalman filter satisfies both these requirements. Both constraints are related to error characteristics:

- The average value of w and z must be 0
- No correlation must exists between w and z

Essentially, at any time point w and z are independent random variables. We assume then $w \sim N(0, S_w)$ and $z \sim N(0, S_z)$, meaning that w and z are gaussian noise with covariance, respectively, S_w and S_z. Notice that we are in presence of multivariate gaussian distributions for d variables, that are specified by a d-elements vector μ and a dxd covariance matrix. In our case the size of the vector is 9, as the size of the state vector and μ is a 9-elements null vector. We will show a simple way to compute the covariance matrix ([20]) later.
We can now briefly expose the Kalman filter equations for the proposed system:

$$K_k = AP_kH^T(HP_kH^T + S_z)^{-1}$$

$$\hat{x}_{x+1} = A\hat{x_k} + K_k(y_{k+1} - H\hat{x}_k)$$

$$P_{k+1} = AP_kA^T + S_w - AP_kH^TS_z^{-1}HP_kA^T$$

In the equations two new parameters appear: K is called the Kalman gain and P is the estimation error covariance.
Some first look observations:

- The state estimation is composed by two elements. The first one is the "natural" evolution of the state, as it appears in the process equations. The second term is called *correction term* and represents how the propagated state should be corrected because of the new measurement.
- Looking at the K definition, the measurement noise is placed at the denominator of the fraction, which implies that if the measurement error is large, S_z will be large and K will be small. Roughly speaking this means that the "credibility" of the measurement is reduced [19].

After this brief introduction on Kalman filtering, in the next paragraphs we will discuss how it is possible to apply this tool in the context of our project.

4.3.1 Initialization

Initial state The state of the system is a vector x initialized as follows:

$$\hat{x} = [p_{x,0}v_{x,0}a_{x,0}p_{y,0}v_{y,0}a_{y,0}p_{z,0}v_{z,0}a_{z,0}]$$

where $p_{i,0}$, $v_{i,0}$ and $a_{i,0}$ are respectively the initial values for position, velocity and acceleration on axis i. For our tests we assume all of them equal to 0. The initial state is just an offset that doesn't affect the evaluation of the goodness of the algorithms.

Transition matrix The state transition defined in the 2.2 is related to a single axis. To build A it is necessary to put the single axis linear system in matrix form:

$$A_i = \begin{bmatrix} 1 & \Delta t & \frac{1}{2}\Delta t^2 \\ 0 & 1 & \Delta t \\ 0 & 0 & 1 \end{bmatrix}$$

Now we can concisely define our state transition matrix:

$$A = \begin{bmatrix} A_x & O_3 & O_3 \\ O_3 & A_y & O_3 \\ O_3 & O_3 & A_z \end{bmatrix}$$

where O_3 is $3x3$ zero matrix.

Error covariance Even the error covariance matrix must be initialized. Its value is computed at every iteration and a wrong initial assumption just delays the convergence of the estimator, which in a long term prospective doesn't significantly affect the performance of the filter. Our initialization hypothesis is quite common (e.g. [20]) and coincides simply with a $9x9$ identity matrix.

Measurements matrix The measurements matrix has the only purpose to establish a relation between measurements and the state of the system. Intuitively, the only part of the state vector combinatorially (that is, without time delay) related to the measurements concerns accelerations. Thus:

$$H_1 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

and then:

$$H = \begin{bmatrix} H_1 & O_3 & O_3 \\ O_3 & H_1 & O_3 \\ O_3 & O_3 & H_1 \end{bmatrix}$$

Noise covariance Because of the integration, the accelerometer noise is propagated to velocity and position. Process noise covariance matrix Q_k has the purpose to allow the Kalman filter to estimate and reduce the effect of the noise propagated to the velocity and position. The Q_0 matrix initialization we choose is based on a proof in [20].

We define Q_i as the process noise covariance in axis i, and then build the entire Q matrix:

$$Q_i = \begin{bmatrix} \frac{1}{20}q_c\Delta t^5 & \frac{1}{8}q_c\Delta t^4 & \frac{1}{6}q_c\Delta t^3 \\ \frac{1}{8}q_c\Delta t^4 & \frac{1}{3}q_c\Delta t^3 & \frac{1}{2}q_c\Delta t^2 \\ \frac{1}{6}q_c\Delta t^3 & \frac{1}{2}q_c\Delta t^2 & q_c\Delta t \end{bmatrix}$$

$$Q = \begin{bmatrix} Q_x & O_3 & O_3 \\ O_3 & Q_y & O_3 \\ O_3 & O_3 & Q_z \end{bmatrix}$$

The definition is reduced to a single parameter q_c which is scalar and assumed equal to the square of the error variance.
The measurement noise covariance can be defined from measuring sensor noise during experiments. For our simulations we apply apriori the same definition of Q.
A deeper insight in how the measurement is updated can be found in [20], for a more theoretical exposition see [17].
As a last note, in a stationary state (the agent is not moving), variations on acceleration, velocity and position should be null. Because of the measurement noise this is not the case. Thus it is a good practice to use some heuristics in order to identify a no movement state (e.g. count the number of null or "about null" samples in a certain time interval, as in the proposed implementation).

5 Simulation

We simulate the results of our algorithm (in Matlab) considering errors on measurement as gaussian noises and using a trapezoid-like (see section 4.2) double integration in the first set of simulation and Kalman filtering in the second set.
We considered four different errors:

1. Error on Zero-g Level (in the datasheet is $\frac{Vdd}{2} \pm 6\%$)
2. Error on sensitivity ($\frac{Vdd}{5} \pm 10\%$)
3. Error on acceleration reading and conversion ($1LSB$)

The proposed errors are modeled as gaussian noise (without further knowledge all that errors are assume to be normally distributed) as follows (assuming Vdd=3.3V):

1. Gaussian noise with mean zero and variance $0.06 * Vdd/2 = 0.099$
2. Gaussian noise with mean zero and variance $0.1 * Vdd/5 = 0.066$
3. Gaussian noise with mean zero and variance $2.56V/1024 = 0.0025$

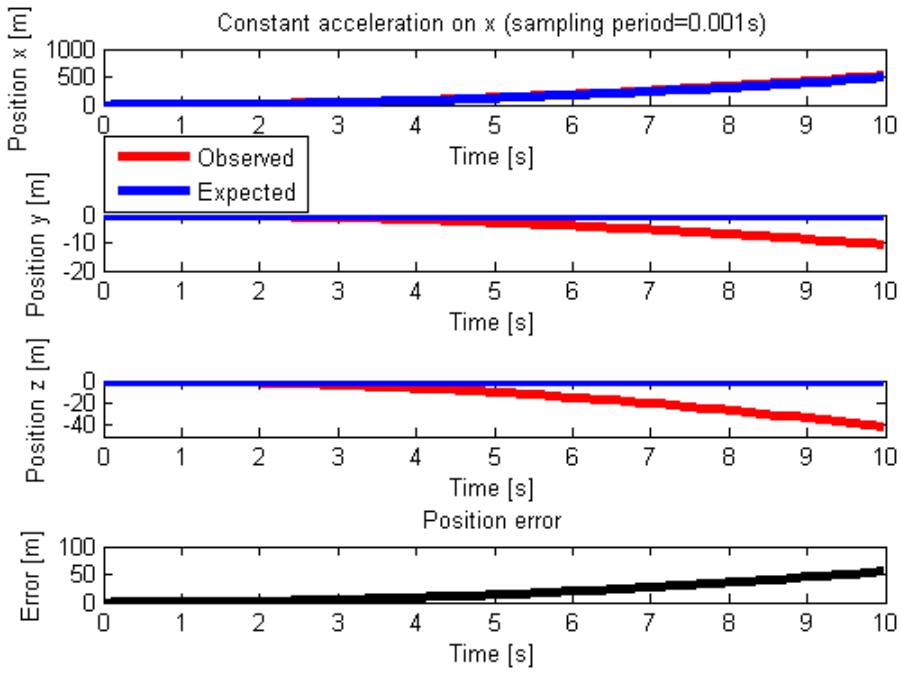
For each proposed solution, we considered the following scenarios:

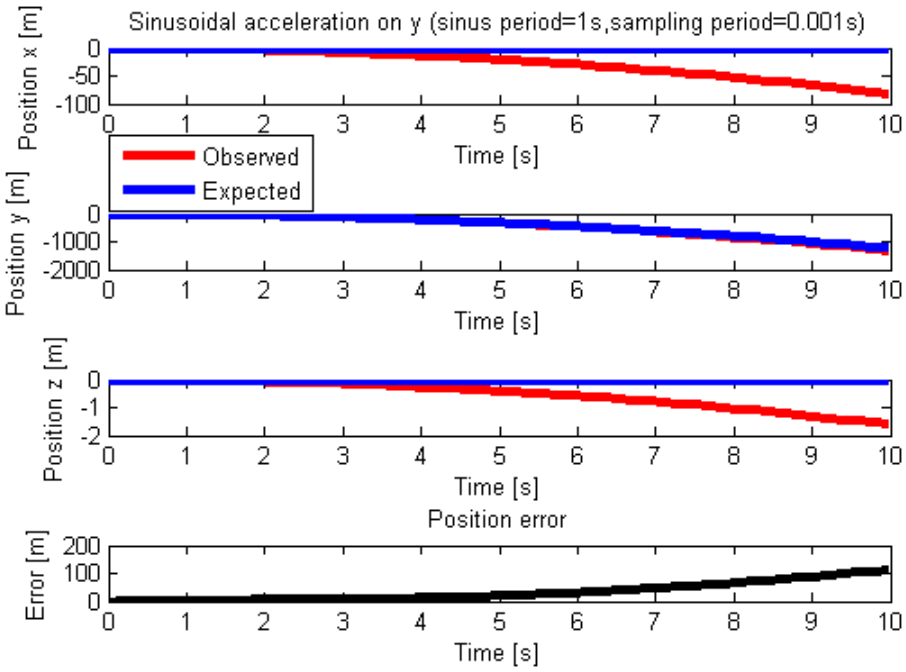
1. Constant acceleration on x
2. Sinusoidal acceleration on y (period=1s)
3. Sinusoidal acceleration on y (period=1s) and constant acceleration on z
4. Constant velocity on y and constant velocity on x

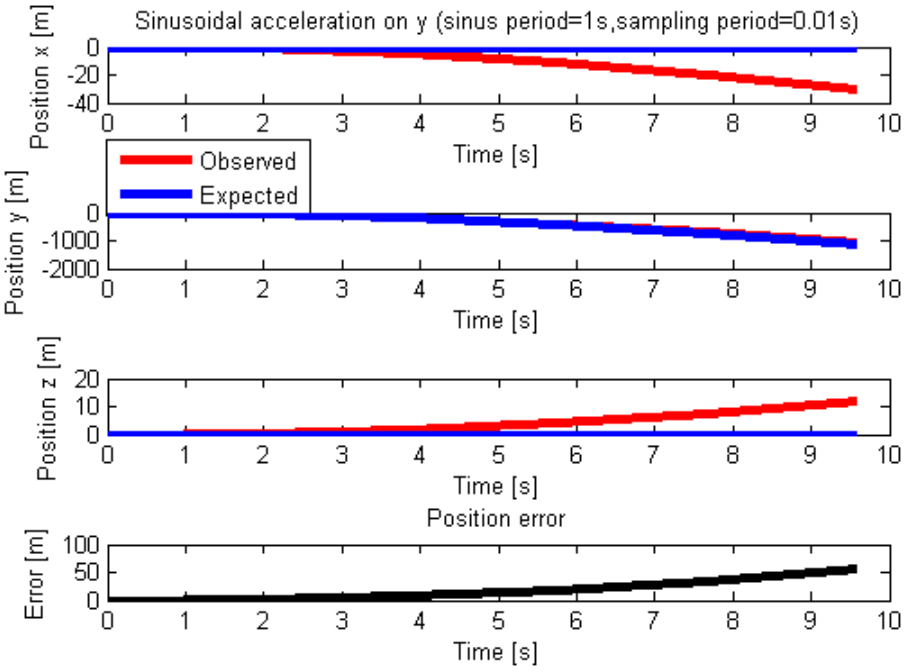
Here follows the results of the simulation with different sampling periods.

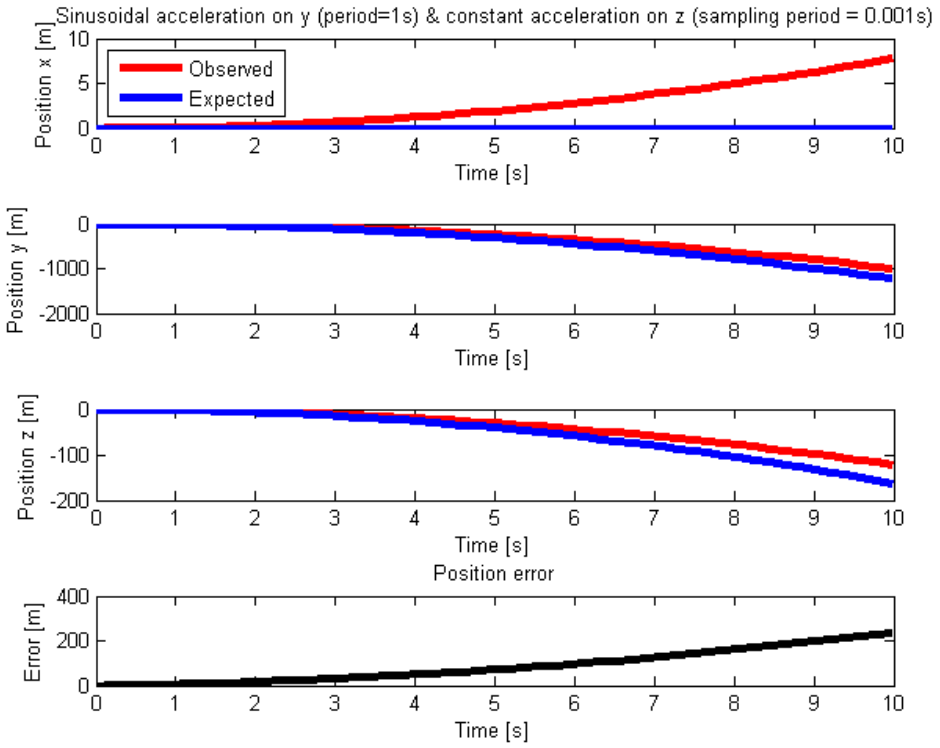
5.1 Numerical integration

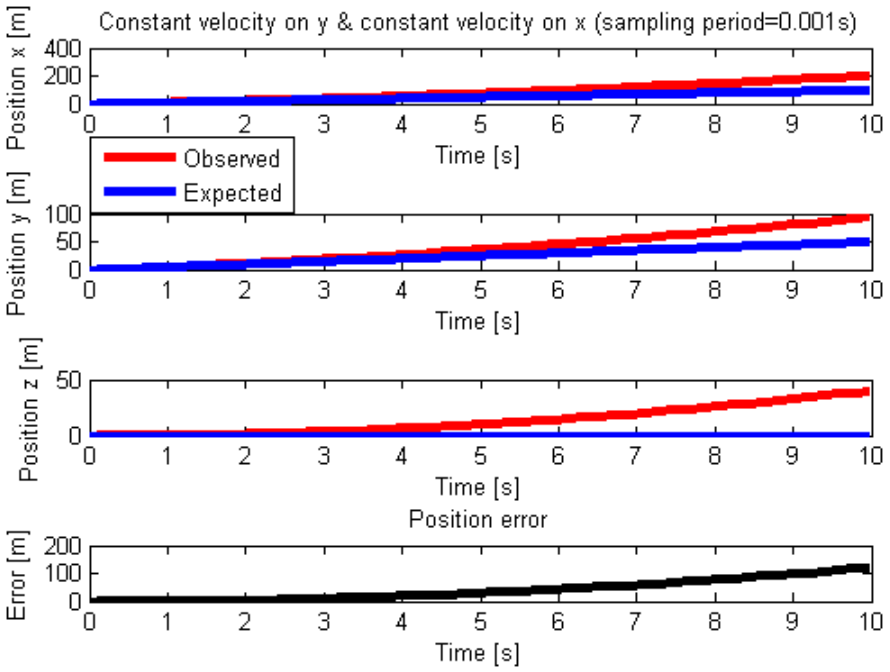
The Matlab code to simulate the double integration can be found in Appendix C.



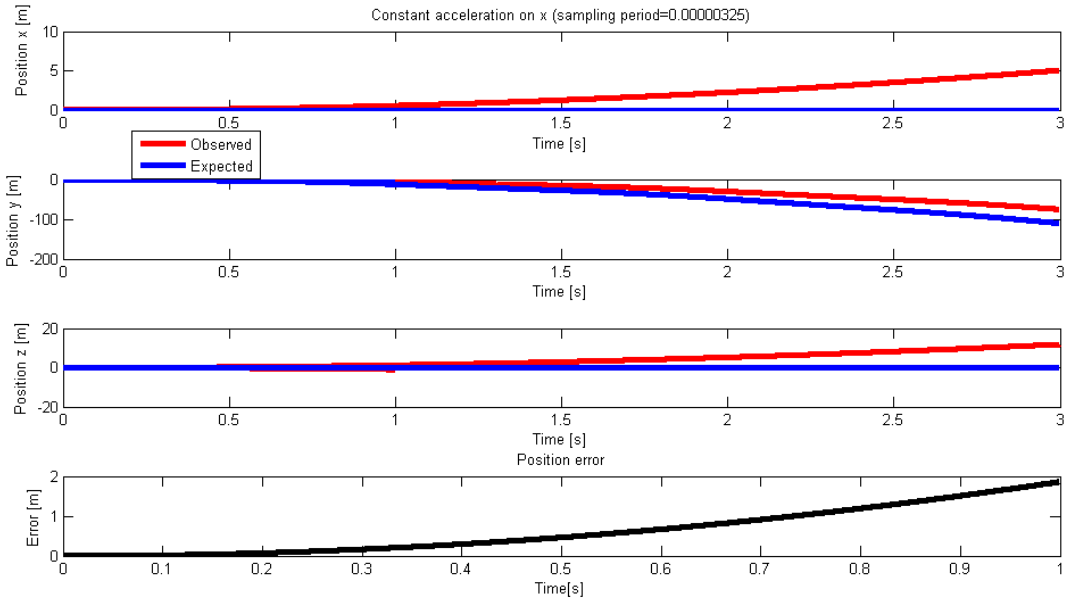

Sinusoidal acceleration on y (sinus period=1s,sampling period=0.001s)
Position x [m]
Time [s]
Observed
Expected
Position y [m]
Time [s]
Position z [m]
Time [s]
Position error
Error [m]
Time [s]

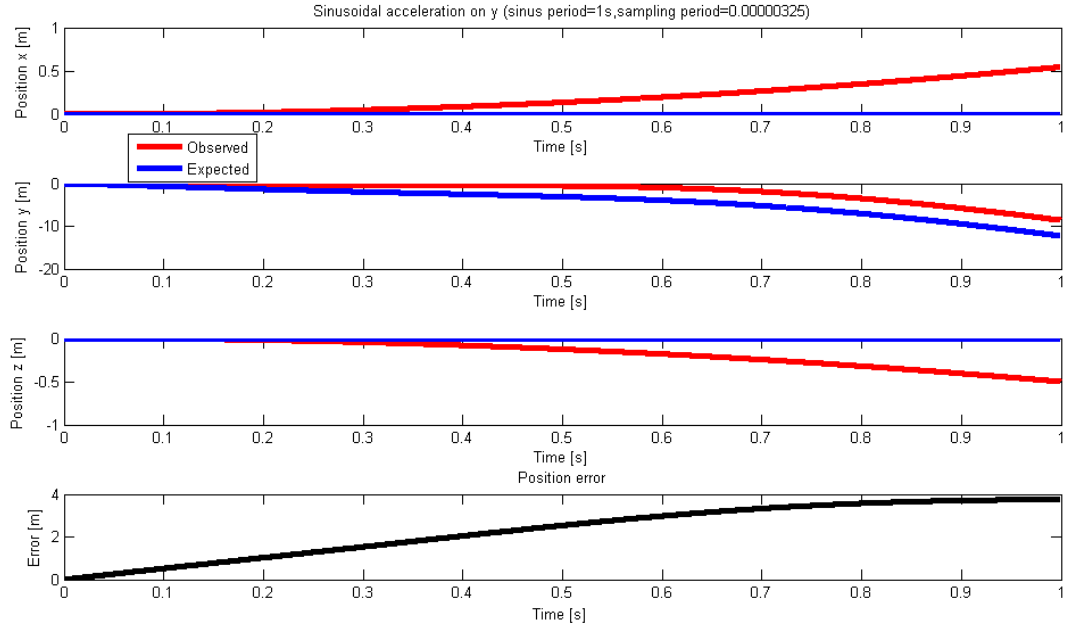

Sinusoidal acceleration on y (sinus period=1s,sampling period=0.01s)
Position x [m]
Time [s]
Observed
Expected
Position y [m]
Time [s]
Position z [m]
Time [s]
Position error
Error [m]
Time [s]

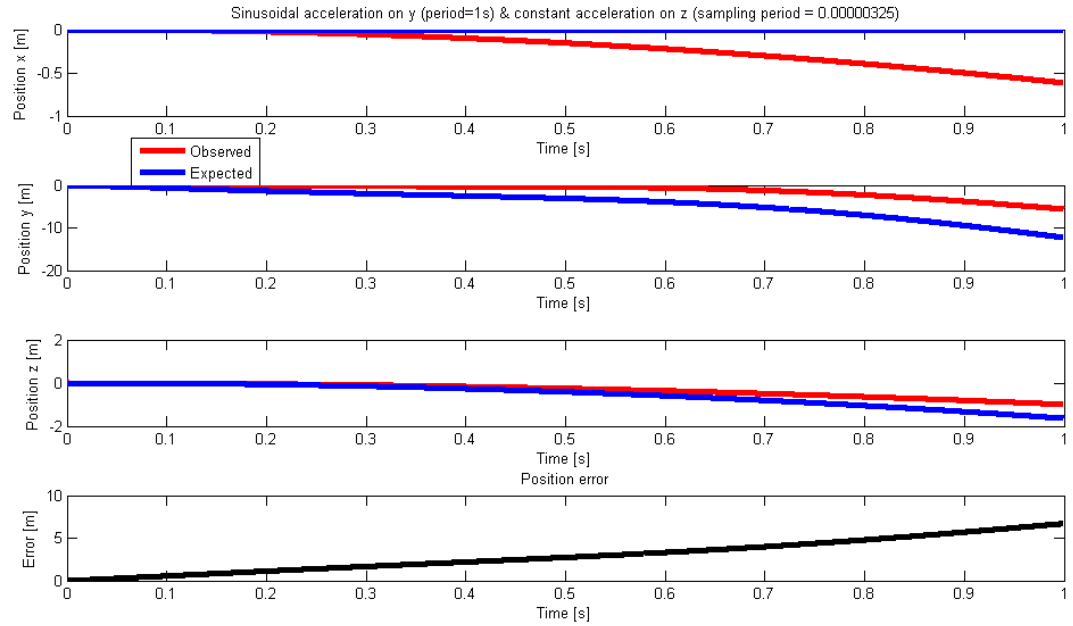

Sinusoidal acceleration on y (period=1s) & constant acceleration on z (sampling period = 0.001s)
Position x [m]
Observed
Expected
Time [s]
Position y [m]
Time [s]
Position z [m]
Time [s]
Position error
Error [m]
Time [s]

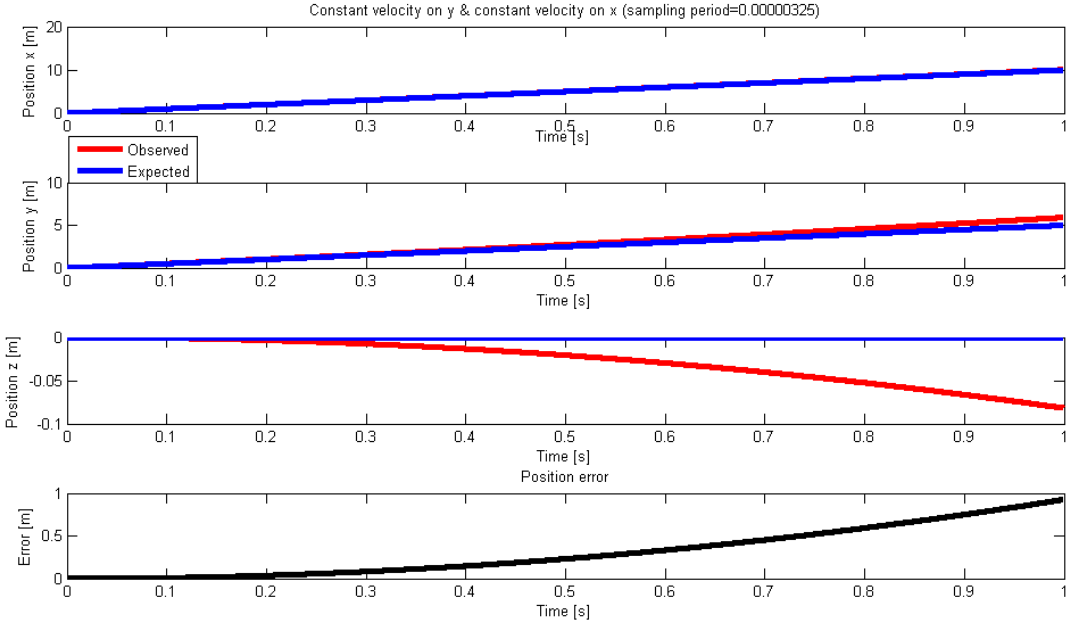

Constant velocity on y & constant velocity on x (sampling period=0.001s)
Position x [m]
Time [s]
Observed
Expected
Position y [m]
Time [s]
Position z [m]
Time [s]
Position error
Error [m]
Time [s]

If we consider the same scenarios using the maximum frequency possible for the microcontroller we get the results shown in the following images. Since a conversion takes approximatively 13 clock cycles and the frequency of the microcontroller is 4MHz, we can simulate our device using a sampling period of $3.25\mu s$.



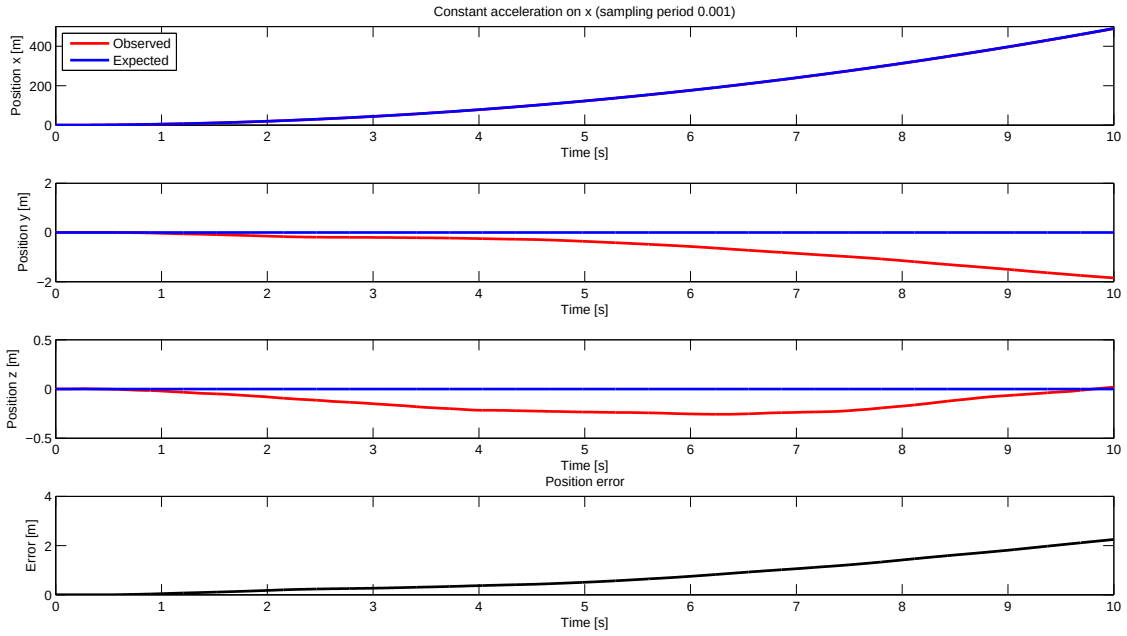


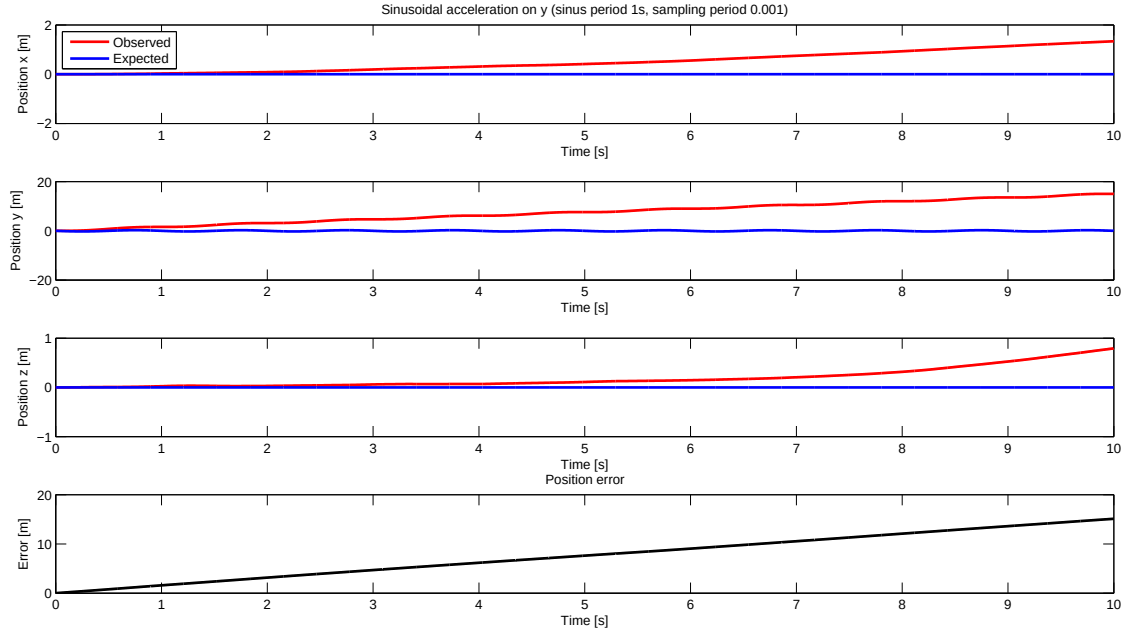


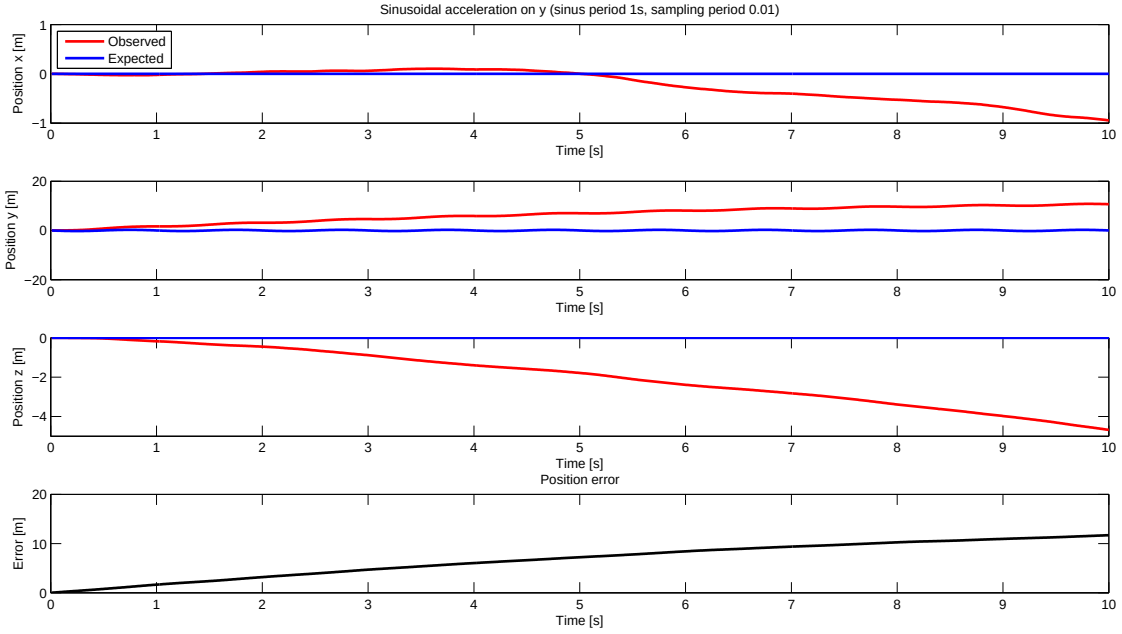


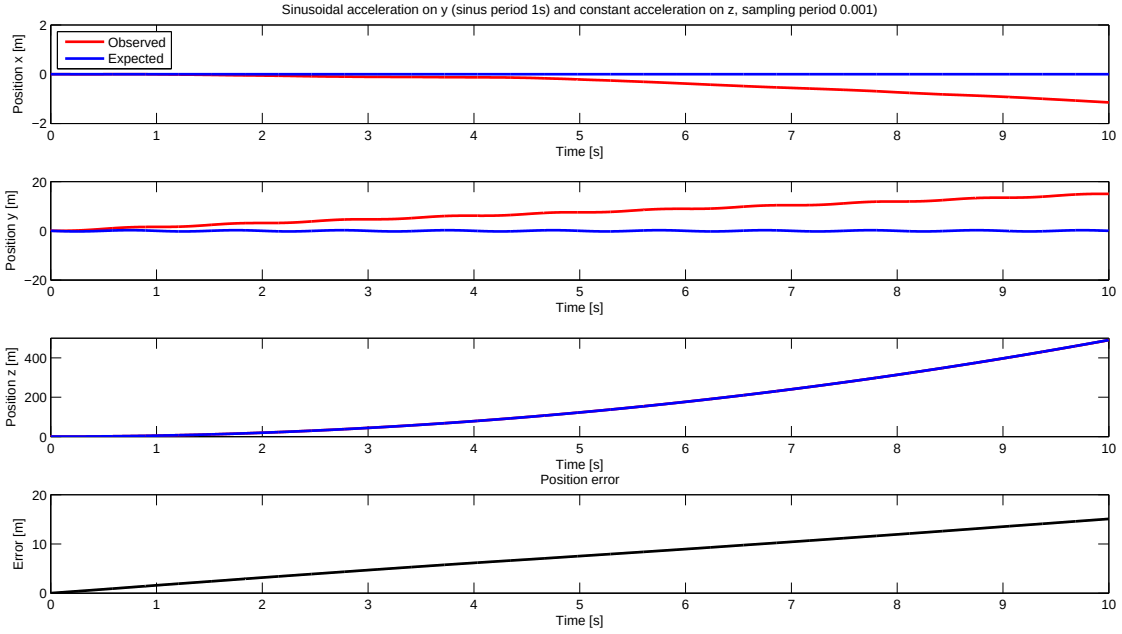
5.2 Kalman filter

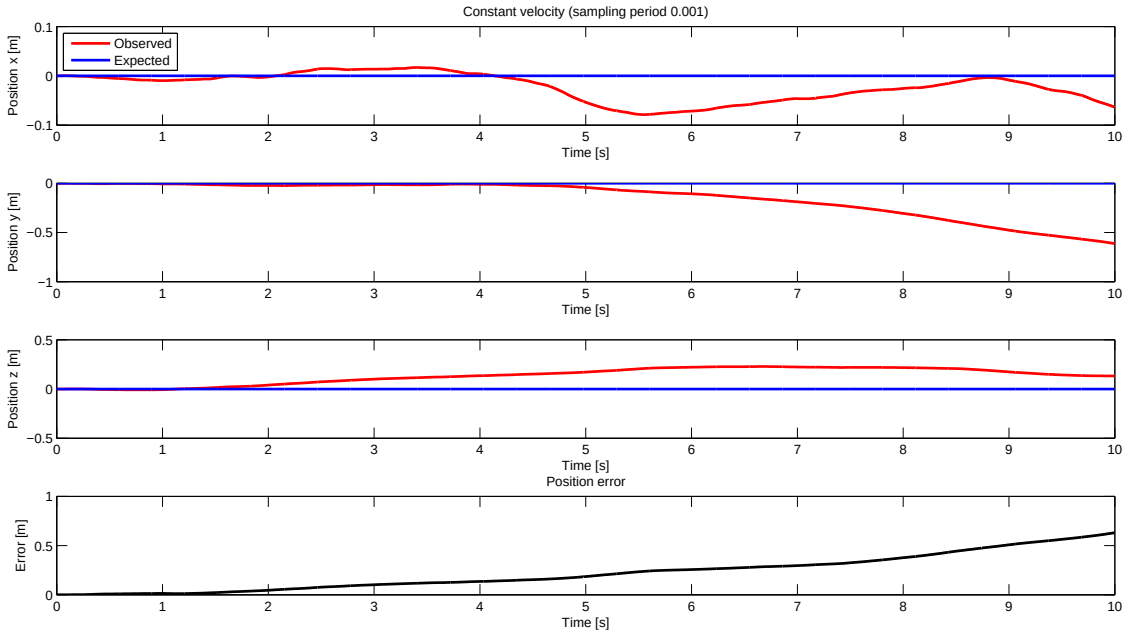
The Matlab code to simulate the double integration can be found in Appendix D.


Constant acceleration on x (sampling period 0.001)
Observed
Expected
Position x [m]
Time [s]
Position y [m]
Time [s]
Position z [m]
Time [s]
Position error
Error [m]
Time [s]

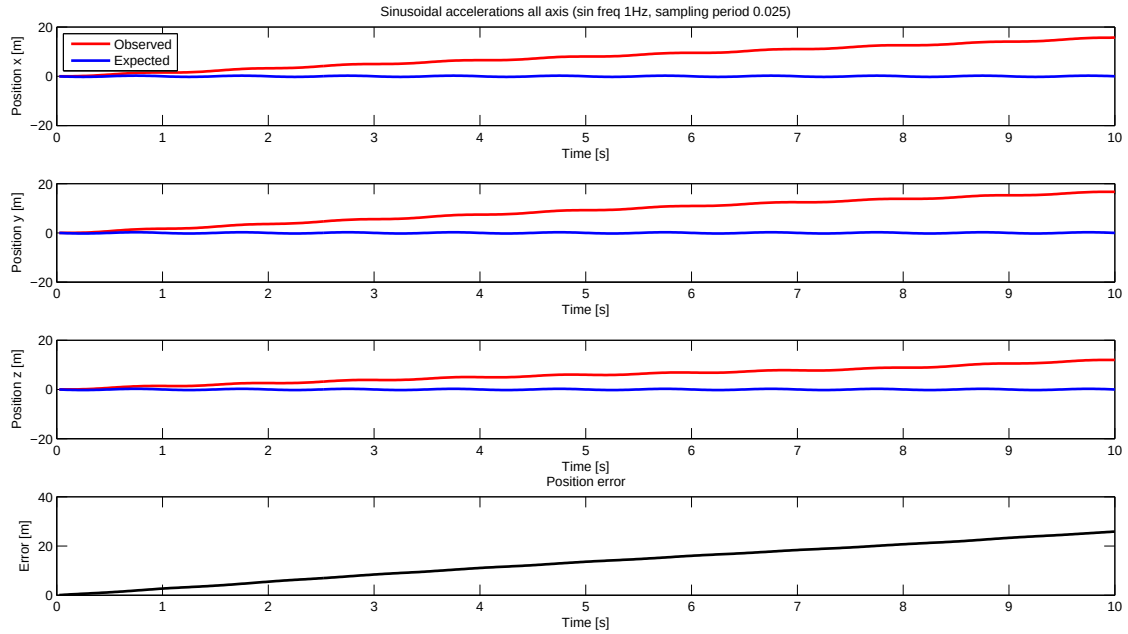

Sinusoidal acceleration on y (sinus period 1s, sampling period 0.001)
Observed
Expected
Position x [m]
Time [s]
Position y [m]
Time [s]
Position z [m]
Time [s]
Position error
Error [m]
Time [s]

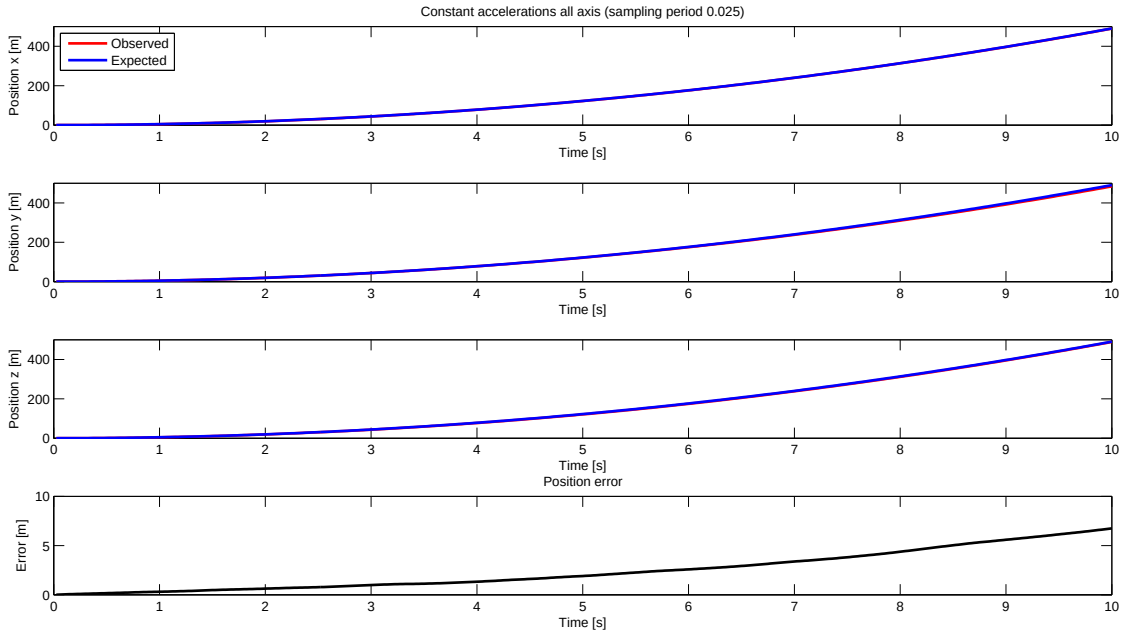

Sinusoidal acceleration on y (sinus period 1s, sampling period 0.01)
Observed
Expected
Position x [m]
Position y [m]
Position z [m]
Time [s]
Position error
Error [m]


Sinusoidal acceleration on y (sinus period 1s) and constant acceleration on z, sampling period 0.001)
Observed
Expected
Position x [m]
Position y [m]
Position z [m]
Time [s]
Position error
Error [m]

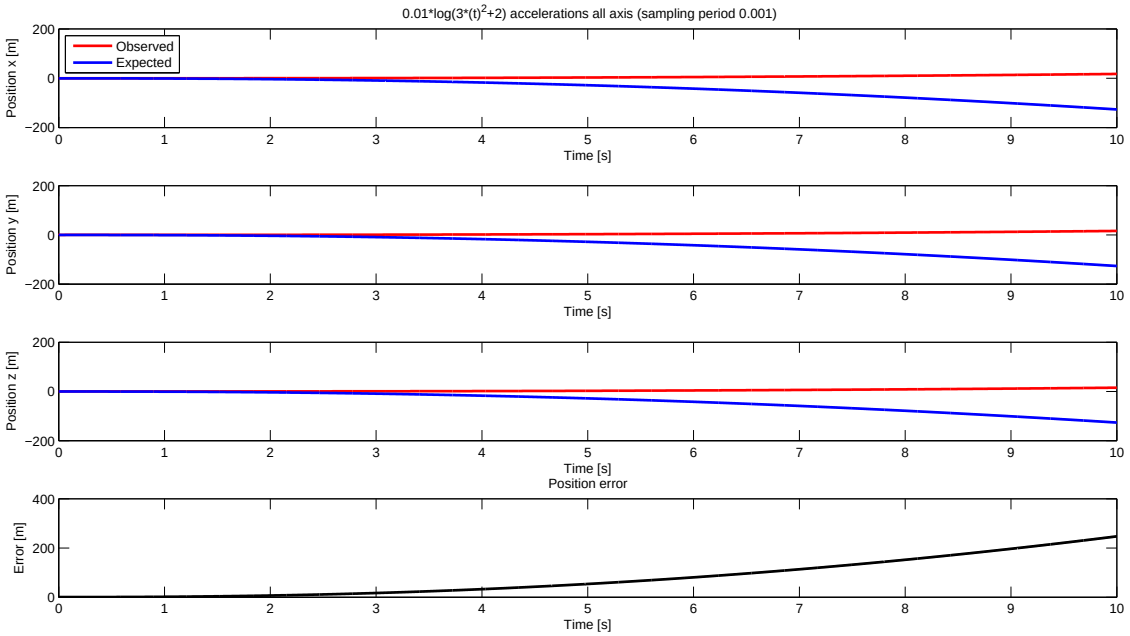


Low frequency sampling (40Hz):





Unusual function with unexpected result:



Error value after 10s position estimation via Kalman filter for sinusoidal functions on all axis (sin frequency 1Hz):

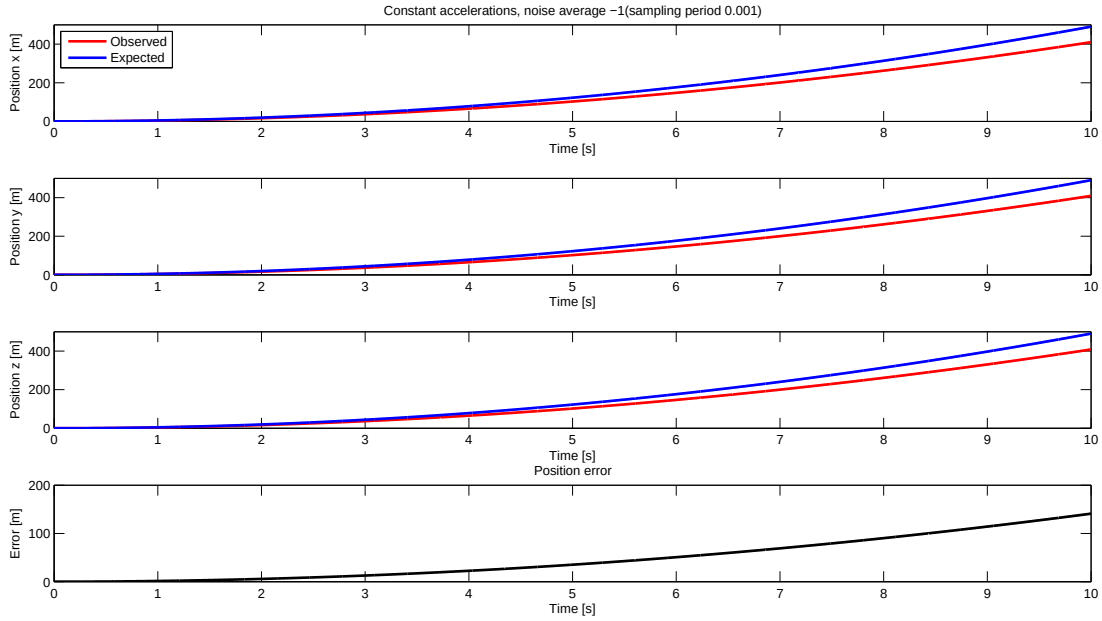
Sampling freq (Hz)	**Final error (m)**
1	98.346190480828426
5	44.878177548165787
10	20.629470928444160
25	26.547130754139552
50	21.870697959426270
100	29.363583202194931
500	25.371543999700716
1000	26.480478434364873

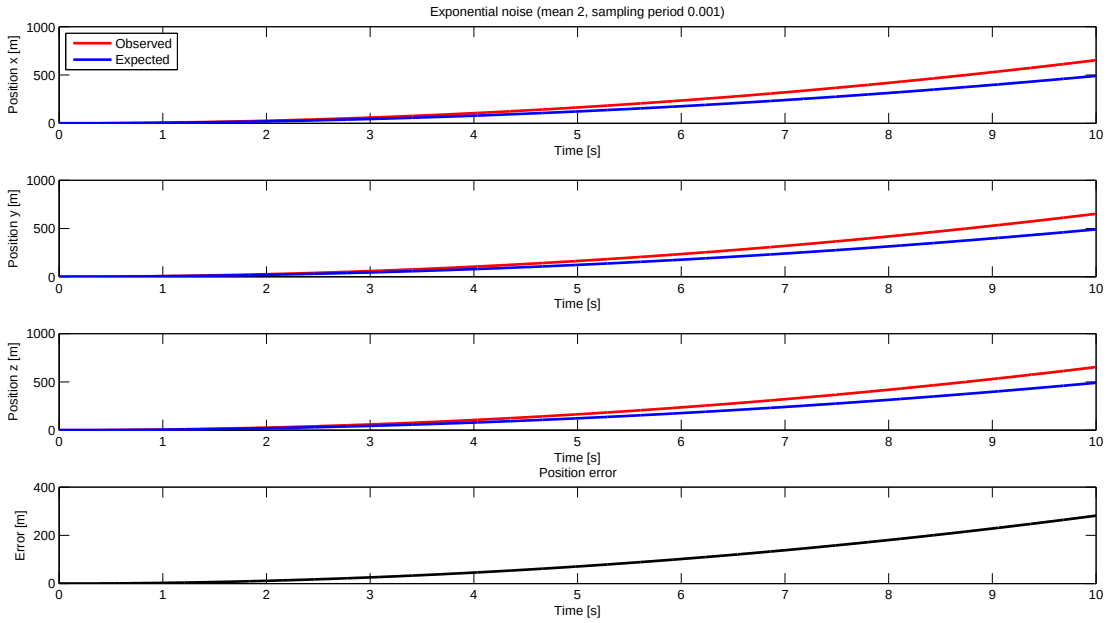
5.3 Considerations on results

Numerical integration procedures are very fast to execute. Algorithms have a quite low complexity and can be easily run on a microcontroller. On the other hand they are very limited in terms of accuracy. We don't exclude that on different inputs from accelerometers other integration rules can perform better, but this make the choice too much data-dependent. It's hard to think of an effective calibration procedure that solves the problem and that can also be user-friendly.
The errors we obtained in simulation, make the algorithm not suitable for pedestrian position recovery, but they may perform well on a car moving on a bounded space (e.g. traced over a street map).
A summary of pros and cons for numerical integration algorithm follows:

Pros	**Cons**
Low complexity	Low Accuracy
Well studied, optimized library	Hard to integrate different sensors
No special requirements on noise distribution	Error propagation at each iteration

On the other hand, Kalman filtering is quite complex for a microcontroller. There are fast implementations of this wide used algorithm. Just to give an idea, most of the matrices involved in the System description and in the Kalman filter itself are sparse. Many multiplications can be avoided just rolling out matrices products. The accuracy of the filter is quite high, in all of the tests. The Kalman filter is one of the most used and effective algorithms for sensors integration (see section 7). It uses noise information both to weigh the measured acceleration with respect to the expected one and to attenuate error propagation from one state to the next one.
Kalman filter has unluckily one strong limitation. It is very sensible to noise distribution. The two following simulations show a case where the average of the normal distribution is shifted and another case in which the noise distribution is exponential instead of gaussian.





Each measure is a source of gaussian noise (because of the reading errors), but it could be the case that other involved agents introduce non gaussian errors that must be pre-filtered so that they won't affect the filter performance. Other biases degrade the accuracy of Kalman filters, but they are easier to identify and to compensate (the average of a gaussian can be estimated with good accuracy).

Also for Kalman filters a short sketch of pros and cons follows:

Pros	**Cons**
High accuracy	Relatively high complexity
Easy sensors integration	Strict requirement on noise distribution
Noise propagation attenuation	Apply only on linear systems

6 Implementation issues

In this section we propose a draft of the implementation of our algorithm for the Atmel AT90CAN128 microcontroller [10]. The proposed implementation is adapted and extended on the base of [18]. It implements the trapezoid-like technique presented in section 4.2.
The source code is presented in appendix E. Here we briefly review some of the implemented functions.

Calibration routine The idea of the calibration routine is to capture a large amount of samples when the agent is not moving. This allows estimating the zero-g level of the accelerometers. Averaging these values can be useful in order to identify eventual bias in the sensors. Some MEMS sensors have dedicated pins for bias compensation directly on the output voltage. The larger the number of samples used for calibration is, the more precise the bias compensation can be. Even if LIS3L02AL has a gravity compensation function for the z-axis, calibration is still recommended for many reasons (wear, mechanical defects, particular environment conditions, ...).

Avering sample readings Again, the average is a good estimator of the expected value for a gaussian distribution, thus instead of a single sample, it is better to retrieve a series of samples and to use the average as the representative value for that time interval.

Almost null movements Sensors are typically sensible to noise in proximity of the zero value. We already analyzed the errors related to stationary states. It could be a good idea to define a minimum non-zero value such that all the elements $|x_i| < \delta$ are considered as $x_i = 0$.

Acquisition routines and reference conversion Reading samples from input ports and Analog-Digital conversion are accomplished in a standard way. A function to convert geodetic to cartesian coordinates is coded. This function can be useful to interface the system with a GPS device.

6.1 Development and testing environments

In order to test our application we used two different environments.
Low-level simulations on microcontroller code have been done using the AVR Simulator integrated in AVR Studio 4 (the suggested IDE for developing applications based on ATMEL devices).
This tool allows debugging the code and simulates the execution of the program line by line. At each step it is possible to see the status of registers, to check the values of the variables and to monitor the outputs in case I/O ports are used. A clock cycle is associated to each line so that it is possible to time the functions used.
This was particularly useful for estimating the maximum conversion frequency obtainable with our algorithm (which defines the maximum input frequency that our device can treat).
High-level simulation of the algorithms was done using Matlab. Errors between expected position and

observed one were computed using a symbolic integration for the expected values and a numerical one based on a trapezoidal method for the observed ones.

7 Conclusions and future work

Proposed solutions and simulations show an high sensitivity of results to the shape and frequency of the sensed acceleration curves. Sampling frequency has a basic lower bound coming from the Shannon theorem, but in our highest accuracy result (Kalman filter) it seems that, after a certain threshold, improvement in accuracy are not significant. This threshold seems to be related to both shape and frequency.

The Kalman filter performance revealed to be related on error profiling. A proper estimation of error variances makes the filter perform well. Kalman filter produced poor results in presence of non gaussian-distributed errors and in presence of biases. Thus if non gaussian error sources get identified a prefiltering of input may be needed. Also biased input should be corrected (e.g. by adding a compensation offset).

From previous considerations, we suggest to make an effort toward real-case data profiling in order to find a reasonable set of pedestrian (or car) acceleration curve shapes and error distributions. These information should drive next research about integration algorithms.

From our bibliographic researches, integration of different sensors achieves higher level of positioning accuracies and reliability. In most of the inertial sensing for position recovery MEMS are accompanied by other electro-mechanical sensors, like gyroscopes, magnetometers, GPS and optical sensors. The use of Kalman filters allows to integrate multi-sensor samples and to use one source to improve each other accuracy.

Another possibly useful procedure for real-life applications is the design of recalibration points. So far only initial calibration was considered (under stationary conditions). But it could be considered that at some point the system can be re-synchronized with a position feedback, coming, for example from a GPS or from the user (for a car another example may be "Telepass" signal on entry points of italian highway). Reliable feedbacks on position can be used to cut error propagation and to eventually refine the mathematical model.

The formalization of the problem as a linear system, opens the door to control and signal theories knowledge. In these field a lot of instruments are available and well studied to filter signals noise and to reconstruct curves from samples. This knowledge may be profitably exploited.

As an advanced topic, after a good profiling, the research of a special purpose estimator to replace the more general Kalman filter could produce very accurate estimations.

8 Appendix A: Reference systems

The main source of error in our system seems to be due to the conversion between different reference-systems. GPS coordinates are essentially spherical coordinates while our accelerometers measure acceleration in a Cartesian system (acc_x, acc_y, acc_z). In order to be able to estimate the evolution of the position when the GPS signal is weak or absent we need to convert all quantities in the same reference system. GPS coordinates needs to be first converted into Earth-Centered, Earth-Fixed system which is a Cartesian coordinate system used for GPS. The origin of the system (0,0,0) is placed at the center of the Earth. The z-axis is defined as being parallel to the earth rotational axes pointing towards north. The x-axis intersects the sphere of the earth at the 0° latitude, 0° longitude. This system of coordinates rotates with the earth around its z-axis which means that coordinates at the surface of the earth doesn't change with rotation (hence the name earth-fixed) ([25]).
This system of coordinates is generally used as an intermediate step when converting from GPS coordinates to the local North East Up system of coordinates (accelerometers gives measures in a local system of coordinates).
Conversion from GPS coordinates to EFEC is done using the formulas:

$$X = \left(\frac{a}{\chi} + h\right) cos(\phi)cos(\lambda)$$

$$Y = \left(\frac{a}{\chi} + h\right) cos(\phi)sin(\lambda)$$

$$Z = \left(\frac{a(1-e^2)}{\chi} + h\right) sin(\phi)$$

where $\chi = \sqrt{1 - e^2 sin^2(\phi)}$, a and e^2 are the semi-major axis and the square of the first numerical eccentricity of the ellipsoid respectively. Geodetic coordinates are expressed by the tuple (latitude ϕ, longitude λ, height h).
The standard used by GPS to determine the location of a point near the surface of the Earth is the world geodetic system 1984 (WGS84) which establish the values of the main parameters as:

$Semi - majoraxis$(a)	6378137.0m
$Semi - minoraxis$(b)	6356752.3142m
$FirstEccentricitySquared$(e^2)	6.69437999014x10^{-3}

Conversion errors

The main errors introduced by this kind of conversion are due to the inaccuracies in the estimation of the parameters. The accuracy (one sigma) of WGS84 coordinates in terms of geodetic latitude ϕ, geodetic longitude λ, and geodetic height h is:

Horizontal $\sigma\phi = \sigma\lambda = \pm 1m(1\sigma)$
Vertical $\sigma h = \pm 1...2m(1\sigma)$

The accuracy of $\pm 1m$ in the definition of WGS84 is sufficient for nearly all air navigation applications but can be a problem when working with smaller environments, for example when we want to estimate the position of a pedestrian in the road ([15]).
In order to use accelerometers estimations it is necessary to convert from our EFEC coordinates to a local system. The most used local system is known as ENU (East North Up). The local ENU coordinates are formed from a plane tangent to the Earth's surface fixed to a specific location (local "geodetic" plane). By convention the east-axis is labeled x, the north is labeled y, the up z.

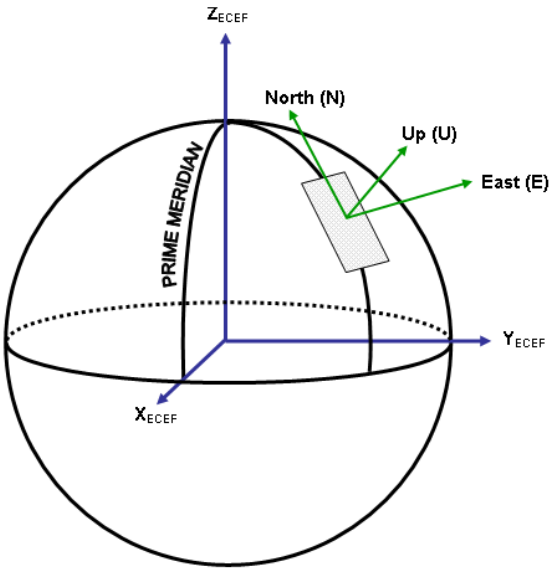


Figure 9: ECEF and ENU systems.

Conversion between ECEF coordinates and ENU coordinates can be done by using the expressions:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} -sin(\lambda) & cos(\lambda) & 0 \\ -sin(\phi)cos(\lambda) & -sin(\phi)sin(\lambda) & cos(\phi) \\ cos(\phi)cos(\lambda) & cos(\phi)sin(\lambda) & sin(\phi) \end{bmatrix} \begin{bmatrix} X_p - X_r \\ Y_p - Y_r \\ Z_p - Z_r \end{bmatrix}$$

Where (X_r, Y_r, Z_r) are the ECEF coordinates of the local reference point (ϕ is the geodetic latitude). Once this local coordinates have been computed it is necessary to convert them in the local system of coordinates of the accelerometer (which is usually not oriented in the same way as the system ENU). In order to do this we must know with high precision the orientation of our device with respect to the ENU system. Small errors in angle estimation can lead to huge errors in position tracking. Computing the local coordinates from the ENU ones can be easily done using a rotational matrix where the angles are the ones we can estimate using gyroscopes (see Appendix B). Once we have computed our position in this local system we need to go back to the GPS coordinates converting from our local system to ENU coordinates and then from ENU coordinates to ECEF one using the reverse transformation:

$$
\begin{aligned}
&r = \sqrt{(X^2 + Y^2)} \\
&E^2 = a^2 - b^2 \\
&F = 54b^2Z^2 \\
&G = r^2 + (1 - e^2)Z^2 - e^2E^2 \\
&C = \frac{e^4Fr^2}{G^3} \\
&S = \sqrt[3]{1 + C + \sqrt{C^2 + 2C}} \\
&P = \frac{F}{3\left(S + \frac{1}{S} + 1\right)^2 G^2} \\
&Q = \sqrt{1 + 2e^4P} \\
&r_0 = \frac{-(Pe^2r)}{1+Q} + \sqrt{\frac{1}{2}a^2(1 + 1/Q) - \frac{P(1-e^2)Z^2}{Q(1+Q)} - \frac{1}{2}Pr^2} \\
&U = \sqrt{(r - e^2r_0)^2 + Z^2} \\
&V = \sqrt{(r - e^2r_0)^2 + (1 - e^2)Z^2} \\
&Z_0 = \frac{b^2Z}{aV} \\
&h = U\left(1 - \frac{b^2Z}{aV}\right) \\
&\phi = arctan\left(\frac{Z + e'^2Z_0}{r}\right) \\
&\lambda = arctan2[Y, X]
\end{aligned}
$$

$$
\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} -sin(\lambda) & -sin(\phi)cos(\lambda) & cos(\phi)cos(\lambda) \\ cos(\lambda) & -sin(\phi)sin(\lambda) & cos(\phi)sin(\lambda) \\ 0 & cos(\phi) & sin(\phi) \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} X_r \\ Y_r \\ Z_r \end{bmatrix}
$$

Converting from ECEF coordinates to GPS ones (using the standard WGS84) is a much harder problem. The most used algorithm for doing this conversion is the one presented by J. Zhu in ([25]).

Geodetics parameters (a, b, e, e') are considered as known.
This conversion formula has an average error of $0.9nm$ (maximum error $632nm$) for displacement. Each conversion between systems of coordinates leads to errors due to inaccuracies in the model or in the orientation estimation.
From the input to the output we apply six different conversions, each one of introduces errors. The output GPS coordinates will be a poor estimation of the real position of the device we are implementing.

9 Appendix B: Gyroscopes

Gyroscopes are necessary to retrieve the orientation of our device with respect to a certain reference system. A gyroscope is a device for measuring or maintaining orientation, based on the principles of angular momentum. By measuring both the angular velocity of the system and the current linear accelerations (using gyroscopes and accelerometers) it is possible to estimate the linear acceleration of the local system in the inertial reference frame.

Performing integration on the inertial accelerations (using the original velocity as an initial condition) using the correct equations allows computing the inertial velocities of the system. Integrating again (knowing the initial position) yields the inertial position. Angular position is usually measured through an integration of the angular rate from the gyroscopes. This produces biases that will be propagated when computing the position in the inertial system.

Gyroscopes are not error-free. The bias, usually measured in °/s, is the offset error output and can be measured when the error is static. The scale factor error is due to the difference between the measured angular rate and the true rate. Gyroscopes are also highly affected by noise which leads to a phenomenon known as 'Random-Walk' behavior. This kind of error is usually measured in °/sqrt(h). A random walk is a mathematical formalization of a trajectory which consists of taking successive random steps. It's a highly non-deterministic behavior and can lead to relevant errors in angular estimation. The main problem when using these kinds of inertial navigation systems is the phenomenon known as "integration drift": small errors when measuring acceleration and angular velocity are integrated into progressively larger errors in velocity and subsequently into greater errors in position (Kim, et al. 1996).

A GPS system, which gives an absolute drift-free position, can be used to adjust the position estimation periodically. The angular orientation of the local system can also be inferred through a series of position updates from the GPS. The introduction of a Kalman filter can highly improve the quality of the results (as shown previously). By properly combining the information from accelerometers, gyroscopes and the GPS system, the errors in position and velocity can be made stable. This procedure can be done, for example, with Kalman filters as in [5].

10 Appendix C: Numerical Integration Matlab Code

simulation.m

```
% Parameters
Vdd=3.3;
latitude=0;
longitude=0;
altitude=0;
sensitivity=Vdd/5;
zero_ref=Vdd/2;

syms x y z

% First simulation
sampling_period=0.01;
end_simulation=5;
speed_x_init=0;
speed_y_init=0;
speed_z_init=0;

fun_x=zero_ref+sensitivity;    % 9.81 m/s^2
fun_y=zero_ref;
fun_z=zero_ref;
title_str='Constant acceleration on x (sampling period=0.01)';
[observed_x,observed_y,observed_z]=acceleration(Vdd,fun_x,fun_y,fun_z,speed_x_init,
speed_y_init,speed_z_init,sampling_period,end_simulation,title_str);

% Second simulation
% Sinusoidal acceleration on y (period=1s)
sampling_period=0.01;
end_simulation=5;

speed_x_init=0;
speed_y_init=0;
speed_z_init=0;

fun_x=zero_ref;    % 9.81 m/s^2
fun_y=(zero_ref+sensitivity)*sin(y*2*pi);
fun_z=zero_ref;

figure
title_str='Sinusoidal acceleration on y (sinus period=1s,sampling period=0.01)';
acceleration(Vdd,fun_x,fun_y,fun_z,speed_x_init,speed_y_init,speed_z_init,
sampling_period,end_simulation,title_str);

% Third simulation
% Sinusoidal acceleration on y (period=1s)
sampling_period=0.05;
end_simulation=5;
```

```
speed_x_init=0;
speed_y_init=0;
speed_z_init=0;

fun_x=zero_ref;    % 9.81 m/s^2
fun_y=(zero_ref+sensitivity)*sin(y*2*pi);
fun_z=zero_ref;

figure
title_str='Sinusoidal acceleration on y (sinus period=1s,sampling period=0.05)';
acceleration(Vdd,fun_x,fun_y,fun_z,speed_x_init,speed_y_init,speed_z_init,
sampling_period,end_simulation,title_str);

% Fourth simulation
% Sinusoidal acceleration on y (period=1s) & constant acceleration on z
end_simulation=5;
sampling_period=0.01;

speed_x_init=0;
speed_y_init=0;
speed_z_init=0;

fun_x=zero_ref;    % 9.81 m/s^2
fun_y=(zero_ref+sensitivity)*sin(y*2*pi);
fun_z=zero_ref-sensitivity/3;

figure
title_str='Sinusoidal acceleration on y (period=1s) & constant acceleration on
 z (sampling period = 0.01)';
acceleration(Vdd,fun_x,fun_y,fun_z,speed_x_init,speed_y_init,speed_z_init,
sampling_period,end_simulation,title_str);

% Fifth simulation
% No acceleration, constant speed on x-axis and y-axis
sampling_period=0.01;
end_simulation=5;
speed_x_init=10;    % 10 m/s
speed_y_init=5;     % 5 m/s
speed_z_init=0;

fun_x=zero_ref;    % 9.81 m/s^2
fun_y=zero_ref;
fun_z=zero_ref;

figure
title_str='Constant velocity on y & constant velocity on x (sampling period=0.01)';
acceleration(Vdd,fun_x,fun_y,fun_z,speed_x_init,speed_y_init,speed_z_init,
sampling_period,end_simulation,title_str);
```

acceleration.m

```
function [observed_x,observed_y,observed_z]=acceleration(Vdd,fun_x,fun_y,fun_z,
speed_x_init,speed_y_init,speed_z_init,integration_step,end_simulation,title_str)
sstatex=Vdd/2;
sstatey=Vdd/2;
sstatez=Vdd/2;
sensitivity=Vdd/5;

pos_x(1)=0;
pos_y(1)=0;
pos_z(1)=0;

j=1;
syms x y z
% We sample analog values

for i=0:integration_step:end_simulation
if(mod(i,0.25)<0.0001)
    i
end
acc_mems_x(j)=subs(fun_x,i);
acc_mems_y(j)=subs(fun_y,i);
acc_mems_z(j)=subs(fun_z,i);
j=j+1;
end

% Substract zero_level acceleration
fun_x=fun_x-sstatex;
fun_y=fun_y-sstatey;
fun_z=fun_z-sstatez;

% Compute acceleration in meters using sensitivity
fun_x=(fun_x/sensitivity)*9.81;    % m/s^2
fun_y=(fun_y/sensitivity)*9.81;    % m/s^2
fun_z=(fun_z/sensitivity)*9.81;    % m/s^2

j=1;
for i=0:integration_step:end_simulation
if(mod(i,0.25)<0.0001)
    i
end
speed_final_x=speed_x_init+int(fun_x,x);
speed_final_y=speed_y_init+int(fun_y,y);
speed_final_z=speed_z_init+int(fun_z,z);

pos_final_x=pos_x(1)+int(speed_final_x,x,0,i);
pos_final_y=pos_y(1)+int(speed_final_y,y,0,i);
pos_final_z=pos_z(1)+int(speed_final_z,z,0,i);

expected_x(j)=double(pos_x(1)+int(speed_final_x,x,0,i));
expected_y(j)=double(pos_y(1)+int(speed_final_y,y,0,i));
```

```
expected_z(j)=double(pos_z(1)+int(speed_final_z,z,0,i));
j=j+1;
end

% Error on acceleration reading and conversion
noise_x=0.0025*randn(length(acc_mems_x),1)';
noise_y=0.0025*randn(length(acc_mems_y),1)';
noise_z=0.0025*randn(length(acc_mems_z),1)';

acc_mems_x=acc_mems_x+noise_x;
acc_mems_y=acc_mems_y+noise_y;
acc_mems_z=acc_mems_z+noise_z;

% Error on speed reading
noise_x=0.001*randn(1,1)';
noise_y=0.001*randn(1,1)';
noise_z=0.001*randn(1,1)';
speed_x(1)=speed_x_init+noise_x;
speed_y(1)=speed_y_init+noise_y;
speed_z(1)=speed_z_init+noise_z;

% Error on Zero-g Level
noise_x=0.099*randn(1,1)';   %(6 per cent of Vdd/2 as stated in the datasheet)
noise_y=0.099*randn(1,1)';
noise_z=0.099*randn(1,1)';

sstatex=sstatex+noise_x;
sstatey=sstatey+noise_y;
sstatez=sstatez+noise_z;

acc_x(1)=acc_mems_x(1)-sstatex;
acc_y(1)=acc_mems_y(1)-sstatey;
acc_z(1)=acc_mems_z(1)-sstatez;

% Error on sensitivity
noise=0.066*randn(1,1);
sensitivity=sensitivity+noise;

% Compute acceleration in meters using sensitivity
acc_x(1)=(acc_x(1)/sensitivity)*9.81;    % m/s^2
acc_y(1)=(acc_y(1)/sensitivity)*9.81;    % m/s^2
acc_z(1)=(acc_z(1)/sensitivity)*9.81;    % m/s^2

observed_x(1)=pos_x(1);
observed_y(1)=pos_y(1);
observed_z(1)=pos_z(1);

for i=2:length(acc_mems_x)
if(mod(i,100)==0)
    i
```

```
end
% Substract zero-reference
acc_x(2)=acc_mems_x(i)-sstatex;
acc_y(2)=acc_mems_y(i)-sstatey;
acc_z(2)=acc_mems_z(i)-sstatez;

% Compute acceleration in meters using sensitivity
acc_x(2)=(acc_x(2)/sensitivity)*9.81;    % m/s^2
acc_y(2)=(acc_y(2)/sensitivity)*9.81;    % m/s^2
acc_z(2)=(acc_z(2)/sensitivity)*9.81;    % m/s^2

% Double integration (compute displacement)
speed_x(2)=speed_x(1)+((acc_x(2)+acc_x(1))*integration_step/2);
pos_x(2)=speed_x(1)*integration_step+pos_x(1)+((speed_x(2)-speed_x(1))*integration_step/2);

speed_y(2)=speed_y(1)+((acc_y(2)+acc_y(1))*integration_step/2);
pos_y(2)=speed_y(1)*integration_step+pos_y(1)+((speed_y(2)-speed_y(1))*integration_step/2);

speed_z(2)=speed_z(1)+((acc_z(2)+acc_z(1))*integration_step/2);
pos_z(2)=speed_z(1)*integration_step+pos_z(1)+((speed_z(2)-speed_z(1))*integration_step/2);

% Save results
observed_x(i)=pos_x(2);
observed_y(i)=pos_y(2);
observed_z(i)=pos_z(2);

% Update status
pos_x(1)=pos_x(2);
pos_y(1)=pos_y(2);
pos_z(1)=pos_z(2);

speed_x(1)=speed_x(2);
speed_y(1)=speed_y(2);
speed_z(1)=speed_z(2);

acc_x(1)=acc_x(2);
acc_y(1)=acc_y(2);
acc_z(1)=acc_z(2);
end

time=[0:integration_step:end_simulation];

subplot(4,1,1)
plot(time,observed_x,'r','LineWidth',4);
hold on
plot(time,expected_x,'b','LineWidth',4);
title(title_str);
xlabel('Time')
ylabel('Position x')
legend('Observed','Expected','Location','NorthWest')
```

```
subplot(4,1,2)
plot(time,observed_y,'r','LineWidth',4);
hold on
plot(time,expected_y,'b','LineWidth',4);
xlabel('Time')
ylabel('Position y')

subplot(4,1,3)
plot(time,observed_z,'r','LineWidth',4);
hold on
plot(time,expected_z,'b','LineWidth',4);
xlabel('Time')
ylabel('Position z')

subplot(4,1,4)
error=sqrt((observed_x-expected_x).^2+(observed_y-expected_y).^2+(observed_z-expected_z).^2);
plot(time,error,'k','LineWidth',4);
title('Position error');
xlabel('Time')
ylabel('Error')

end
```

11 Appendix D: Kalman Filter Matlab Code

kalman.m

```
clear all;
format long;
s.x=[0 0 0 0 0 0 0 0 0]';
f=1000;
dt=1/f;
s.A=[1 dt (1/2*dt^2) 0 0 0 0 0 0;0 1 dt 0 0 0 0 0 0;0 0 1 0 0 0 0 0 0;
0 0 0 1 dt (1/2*dt^2) 0 0 0;0 0 0 0 1 dt 0 0 0;0 0 0 0 0 1 0 0 0;
0 0 0 0 0 0 1 dt (1/2*dt^2); 0 0 0 0 0 0 0 1 dt; 0 0 0 0 0 0 0 0 1];
qc=0.166^2;
q1=[(1/20*qc*dt^5) (1/8*qc*dt^4) (1/6*qc*dt^3);(1/8*qc*dt^4) (1/3*qc*dt^3)
 (1/2*qc*dt^2); (1/6*qc*dt^3) (1/2*qc*dt^2) (qc*dt)];
o3=zeros(3);
s.Q=[q1 o3 o3;o3 q1 o3; o3 o3 q1];
a1=[0 0 0;0 0 0; 0 0 1]
s.H=[a1 o3 o3;o3 a1 o3; o3 o3 a1];
s.R=s.Q
s.B=zeros(9)
s.u=zeros(9,1)
s.P=s.Q
iter=10/dt;
for k=1:iter
t=[0 0 1 0 0 1 0 0 1];
s(end).z=(t'+[0 0 randn 0 0 randn 0 0 randn]'*0.166)*9.81;
s(end+1)=kalmanf(s(end));
end

for k=1:iter
xe(k)=s(k).x(1);
ye(k)=s(k).x(4);
ze(k)=s(k).x(7);

xr(k)=1/2*9.81*(k*dt)^2;;
yr(k)=1/2*9.81*(k*dt)^2;
zr(k)=1/2*9.81*(k*dt)^2;
error(k)=sqrt((xe(k)-xr(k))^2+(ye(k)-yr(k))^2 +(ze(k)-zr(k))^2);
end

title_str='Constant acceleration on all axis (sampling period 0.001)';

time=[1:iter]*dt;

subplot(4,1,1)
plot(time,xe,'r','LineWidth',2);
hold on
plot(time,xr,'b','LineWidth',2);
title(title_str);
xlabel('Time [s]')
```

```
ylabel('Position x [m]')
legend('Observed','Expected','Location','NorthWest')

subplot(4,1,2)
plot(time,ye,'r','LineWidth',2);
hold on
plot(time,yr,'b','LineWidth',2);
xlabel('Time [s]')
ylabel('Position y [m]')

subplot(4,1,3)
plot(time,ze,'r','LineWidth',2);
hold on
plot(time,zr,'b','LineWidth',2);
xlabel('Time [s]')
ylabel('Position z [m]')

subplot(4,1,4)
plot(time,error,'k','LineWidth',2);
title('Position error');
xlabel('Time [s]')
ylabel('Error [m]')
```

kalmanf.m

```
function s = kalmanf(s)

% set defaults for absent fields:
if ¬isfield(s,'x'); s.x=nan*z; end
if ¬isfield(s,'P'); s.P=nan; end
if ¬isfield(s,'z'); error('Observation vector missing'); end
if ¬isfield(s,'u'); s.u=0; end
if ¬isfield(s,'A'); s.A=eye(length(x)); end
if ¬isfield(s,'B'); s.B=0; end
if ¬isfield(s,'Q'); s.Q=zeros(length(x)); end
if ¬isfield(s,'R'); error('Observation covariance missing'); end
if ¬isfield(s,'H'); s.H=eye(length(x)); end

if isnan(s.x)
   % initialize state estimate from first observation
   if diff(size(s.H))
      error('Observation matrix must be square and invertible for state autointialization.');
   end
   s.x = inv(s.H)*s.z;
   s.P = inv(s.H)*s.R*inv(s.H');
else

   % This is the code which implements the discrete Kalman filter:

   % Prediction for state vector and covariance:
   s.x = s.A*s.x + s.B*s.u;
```

```
    s.P = s.A * s.P * s.A' + s.Q;

    % Compute Kalman gain factor:
    K = s.P*s.H'*inv(s.H*s.P*s.H'+s.R);

    % Correction based on observation:
    s.x = s.x + K*(s.z-s.H*s.x);
    s.P = s.P - K*s.H*s.P;

end

return
```

12 Appendix E: AVR C Source Code

```
#include <stdio.h>
#include <avr/io.h>
#include <avr/interrupt.h>
#include <math.h>
#define ZEROTOSTOP 25
#define MINNOZERO 3
#define PERIOD 1
#define PI 3.141592653589793

//Used to identify an end of movement condition
unsigned char countx, county, countz;
//Current and previous accelerations to compute integration
signed int accelerationx[2], accelerationy[2], accelerationz[2];
signed long velocityx[2], velocityy[2], velocityz[2];
signed long positionX[2], positionY[2], positionZ[2];
//Bias for readings, zero reference (useful with calibration, zero otherwise)
unsigned long sstatex,sstatey,sstatez;
//input from accelerometer
double x,y,z;
unsigned char countConv;
double x0; // Spherical
double y0; // Spherical
double z0; // Spherical
double x_out;   // Spherical
double y_out;   // Spherical
double z_out;   // Spherical
double r0;
double p0;
double t0;
double n;

void init_ADC(void);
void init_delay(void);
void convertToSpherical(double x,double y,double z,double* r,double* p,double* t);
void convertToCartesian(double r,double p,double t,double* x,double* y,double* z);
void read_ADC(double* x,double* y,double* z);
void movement_end_check(void);
void mainRoutine(void);
void computePosition(void);
void lowerErrorIntegral(void);
void Calibrate(void);
void cartesianToNMEA(void);

int main (void){
    //init everything
    //if not calibration
    sstatex=0;
    sstatey=0;
```

```
    sstatez=0;
    unsigned char seconds;
    sei();
    DDRC=0x01;  //bit 0 is output
    init_ADC();
    init_delay();
    Calibrate();
    // PORTD. PORTE, PORTF, PORTG are used as simulation ports for the MEMS data
    // They are all configured as inputs
    // acc_x : PORTD.1 PORTD.0 PORTE.7 PORTE.6 PORTE.5 PORTE.4 PORTE.3 PORTE.2 PORTE.1 PORTE.0
    // acc_y : PORTD.3 PORTD.2 PORTA.7 PORTA.6 PORTA.5 PORTA.4 PORTA.3 PORTA.2 PORTA.1 PORTA.0
    // acc_z : PORTD.5 PORTD.4 PORTG.7 PORTG.6 PORTG.5 PORTG.4 PORTG.3 PORTG.2 PORTG.1 PORTG.0
    DDRD=0x00;
    DDRA=0x00;
    DDRC=0x00;
    DDRE=0x00;
    // ADC = Vin * 1023/ Vref (Vref =2.56)
    seconds=0;
    while(1){
        while((PORTB & 0x00) == 0){
            seconds++;
        }
    }
    return 0;
}

ISR(TIMER1_COMPA_vect){
    //Start conversion of x
    //ADCSRA = ADCSRA | (1<<ADSC);
    //countConv++;
    mainRoutine();
}

void mainRoutine(void){
    unsigned char count2;
    count2=0;
    do{
    read_ADC(&x,&y,&z);

    //Moving average (sum 64 samples)
    accelerationx[1]=accelerationx[1] + x;
    accelerationy[1]=accelerationy[1] + y;
    accelerationz[1]=accelerationz[1] + z;

    count2++;
    }while (count2!=0x40);

    //divide by 64
    accelerationx[1]= accelerationx[1]>>6;
    accelerationy[1]= accelerationy[1]>>6;
```

```
    accelerationz[1]= accelerationz[1]>>6;

    //subtract zero reference
    accelerationx[1] = accelerationx[1] - (int)sstatex;
    acceleration y[1] = acceleration y[1] - (int)sstatey;
    accelerationz[1] = accelerationz[1] - (int)sstatez;

    //Almost-null values set to 0
    if((accelerationx[1] <=MINNOZERO)&&(accelerationx[1] >= -MINNOZERO)){
        accelerationx[1] = 0;
    }
    if((acceleration y[1] <=MINNOZERO)&&(acceleration y[1] >= -MINNOZERO)){
        acceleration y[1] = 0;
    }
    if((accelerationz[1] <=MINNOZERO)&&(accelerationz[1] >= -MINNOZERO)){
        accelerationz[1] = 0;
    }

    computePosition();

    //State update
    accelerationx[0] = accelerationx[1];
    acceleration y[0] = acceleration y[1];
    accelerationz[0] = accelerationz[1];
    velocityx[0] = velocityx[1];
    velocityy[0] = velocityy[1];
    velocityz[0] = velocityz[1];

    //velocity_[1] ready for output

    //Check movement and eventually reset current velocity
    movement_end_check();
    //complete state update
    positionX[0] = positionX[1];
    positionY[0] = positionY[1];
    positionZ[0] = positionZ[1];


}


void computePosition(void){
    lowerErrorIntegral();
    //trapezoidIntegral();
}

void Calibrate(void){
    unsigned int count1;
    count1 = 0;
    do{
```

```
        read_ADC(&x,&y,&z);
        sstatex = sstatex + x;
        sstatey = sstatey + y;
        sstatez = sstatez + z;
        count1++;
    }while(count1!=1024);
    //division by 1024
    sstatex=sstatex>>10;
    sstatey=sstatey>>10;
}

void movement_end_check(void){
    if(accelerationx[1]==0){
        countx++;
    }else{
        countx =0;
    }
    if(countx≥ZEROTOSTOP){
        velocityx[1]=0;
        velocityx[0]=0;
    }
    if(accelerationy[1]==0){
        county++;
    }else{
        county =0;
    }
    if(county≥ZEROTOSTOP){
        velocityy[1]=0;
        velocityy[0]=0;
    }
    if(accelerationz[1]==0){
        countz++;
    }else{
        countz =0;
    }
    if(countz≥ZEROTOSTOP){
        velocityz[1]=0;
        velocityz[0]=0;
    }
}


void init_ADC(void){
    ADMUX=0xE0;          // Input on ADC0
    ADCSRB=0x81;
    ADCSRA=0x98;         // Compare match B

}

void init_delay(void){
```

```
    TIFR1 = 0x00;
    OCR1A=0x00FF;       // Frequency of the timer interrupt
    DDRB=0x21;          // Define the pin OC1A as an output
    TIMSK1=0x02;
    TCCR1A = 0x40;      // Clear Timer On Compare Match (Toggle on match)
    TCCR1B = 0x09;      // No pre-scaler
}

void convertToSpherical(double x,double y,double z,double* r,double* p,double* t){
    *r=sqrt(x*x+y*y+z*z);
    *p=acos(z/(*r));
    if(x<0){
        *t=PI-asin(y/sqrt(x*x+y*y));
    }else{
        *t=asin(y/sqrt(x*x+y*y));
    }
}

void convertToCartesian(double r,double p,double t,double* x,double* y,double* z){
    *x=r*sin(p)*cos(t);
    *y=r*sin(p)*sin(t);
    *z=r*cos(p);
}

void read_ADC(double* x,double* y,double* z){
    // Convert x
    ADMUX=0xE0;
    ADCSRA = ADCSRA | (1<<ADSC);        // start a conversion
    while( (ADCSRA & 0x10) == 0 );      // wait for conversion to be completed
    *x=(PINE) | ((PIND & 0x03)<<8);
    ADCSRA = ADCSRA | 0x10;             // clear the flag

    // Convert y
    ADMUX=0xE1;
    ADCSRA = ADCSRA | (1<<ADSC);        // start a conversion
    while( (ADCSRA & 0x10) == 0 );      // wait for conversion to be completed
    *y=(PINA) | ((PIND & 0x0C)<<6);
    ADCSRA = ADCSRA | 0x10;             // clear the flag

    // Convert z
    ADMUX=0xE2;
    ADCSRA = ADCSRA | (1<<ADSC);        // start a conversion
    while( (ADCSRA & 0x10) == 0 );      // wait for conversion to be completed
    *z=(PINC) | ((PIND & 0x30)<<4);
    ADCSRA = ADCSRA | 0x10;             // clear the flag
}

void cartesianToNMEA(void){
    double a,b,f,p,e2,E2,theta,N,lat,lon,h;
    a=6378137.0;
```

```
    b=6356752.3142;
    f=(a-b)/a;
    e2=2*f - f*f;
    E2=(a*a-b*b)/(b*b);
    p=sqrt(positionX[0]*positionX[0] + positionY[0]*positionY[0]);
    lon = atan2(positionY[0],positionX[0]);
    theta=atan((positionZ[0]*a)/(p*b));
    lat=atan((positionZ[0]+E2*b*pow(sin(theta),3))/(p-e2*a*pow(cos(theta),3)) );
    N=a/sqrt(1-e2*sin(lat)*sin(lat));
    h=p/cos(lat)-N;
}

void lowerErrorIntegral(void){
    velocityx[1]= velocityx[0]+ accelerationx[0]+((accelerationx[1] -accelerationx[0])>>1);
    positionX[1]= positionX[0] + velocityx[0] + ((velocityx[1] - velocityx[0])>>1);
    velocityy[1] = velocityy[0] + accelerationy[0] + ((accelerationy[1] -accelerationy[0])>>1)
    positionY[1] = positionY[0] + velocityy[0] + ((velocityy[1] - velocityy[0])>>1);
    velocityz[1] = velocityz[0] + accelerationz[0] + ((accelerationz[1] -accelerationz[0])>>1)
    positionZ[1] = positionZ[0] + velocityz[0] + ((velocityz[1] - velocityz[0])>>1);
}

void trapezoidIntegral(void){
    velocityx[1]= velocityx[0]+ ((accelerationx[1] -accelerationx[0])>>1);
    positionX[1]= positionX[0] + ((velocityx[1] - velocityx[0])>>1);
    velocityy[1] = velocityy[0] + ((accelerationy[1] -accelerationy[0])>>1);
    positionY[1] = positionY[0] + ((velocityy[1] - velocityy[0])>>1);
    velocityz[1] = velocityz[0] + ((accelerationz[1] -accelerationz[0])>>1);
    positionZ[1] = positionZ[0]  + ((velocityz[1] - velocityz[0])>>1);
}
```

References

1. A beginner's guide to accelerometers, 2009. [Online; accessed 26-January-2009].

2. Numerical integration, 2009. [Online; accessed 15-January-2009].

3. J. Collin, G. Lachapelle, and J. Kappi. MEMS-IMU for Personal Positioning in a Vehicle–A Gyro-Free Approach. In *Proceedings of ION GPS*, pages 24–27, 2002.

4. STMicroelectronics Corp. LIS3L02AL MEMS inertial sensor: 3-axis - +/-2g ultracompact linear accelerometer. *Datasheet number CD00068496*, 2006.

5. P. Davidson, J. Hautamäki, and J. Collin. Using low-cost MEMS 3D accelerometer and one gyro to assist GPS based car navigation system.

6. P.J. Davis and P. Rabinowitz. Methods of Numerical Integration. *New York*, 1975.

7. S. Godha, G. Lachapelle, and ME Cannon. Integrated GPS/INS system for pedestrian navigation in a signal degraded environment. In *Proceedings of the 19th International Technical Meeting of the Satellite Division of the Institute of Navigation ION GNSS 2006*, 2006.

8. F.B. Hildebrand. *Introduction to Numerical Analysis*. McGraw-Hill Companies, 1974.

9. S. Hong, M.H. Lee, S.H. Kwon, and H.H. Chun. A car test for the estimation of GPS/INS alignment errors. *Intelligent Transportation Systems, IEEE Transactions on*, 5(3):208–218, 2004.

10. Atmel Inc. AT90CAN128 8-bit AVR Microcontroller with 128K Bytes of ISP Flash and CAN Controller. *Datasheet number 4250H–CAN*, 2006.

11. W. Kahan. Handheld calculator evaluates integrals. *Hewlett-Packard Journal*, 31(8):23–32, 1980.

12. R.E. Kalman. A new approach to linear filtering and prediction problems. *Journal of Basic Engineering*, 82(1):35–45, 1960.

13. E.D. Kaplan. *Understanding GPS: Principles and Applications*. Artech House, 1996.

14. J. Kim, J.G. Lee, G.I. Jee, and T.K. Sung. Compensation of gyroscope errors and GPS/DR integration. In *Position Location and Navigation Symposium, 1996., IEEE 1996*, pages 464–470, 1996.

15. μ-blox ag. Datum Transformations of GPS Positions. 1999.

16. D. Roetenberg. Inertial and Magnetic Sensing of Human Motion. *PhD Thesis*, 2006.

17. S.J. Russell, P. Norvig, J.F. Canny, J. Malik, and D.D. Edwards. *Artificial intelligence: a modern approach*. Prentice Hall Englewood Cliffs, NJ, 1995.

18. K. Seifer and O. Camacho. Implementing Positioning Algorithms Using Accelerometers. AN3397 Rev 0, 02/2007. *Freescale Semiconductors Inc. Application Note*, 2007.

19. D. Simon. Kalman Filtering. *Embedded Systems Programming*, 14(6):72–79, 2001.

20. C. C. Tsang. Error Reduction Techniques for a MEMS Accelerometer-based Digital Input Device. *PhD Thesis*, 2008.

21. M. Unser. Sampling-50 years after Shannon. *Proceedings of the IEEE*, 88(4):569–587, 2000.

22. J. Wang and Y. Gao. The Aiding of MEMS INS/GPS Integration Using Artificial Intelligence for Land Vehicle Navigation. *IAENG International Journal of Computer Science*, 33(1):61–67, 2007.

23. G. Welch and G. Bishop. An Introduction to the Kalman Filter. *University of North Carolina at Chapel Hill, Chapel Hill, NC*, 1995.

24. Wikipedia. Geodetic system — wikipedia, the free encyclopedia, 2009. [Online; accessed 01-February-2009].

25. J. Zhu. Conversion of Earth-centered Earth-fixed coordinates to geodeticcoordinates. *Aerospace and Electronic Systems, IEEE Transactions on*, 30(3):957–961, 1994.