I.J. Modern Education and Computer Science, **2021, 4, 42-54**
Published Online August 2021 in MECS (http://www.mecs-press.org/)
DOI: 10.5815/ijmecs.2021.04.04

A Potrace-based Tracing Algorithm for Prescribing Two-dimensional Trajectories in Inertial Sensors Simulation Software

Bohdan R. Tsizh
Department of General Technical Subjects, Stepan Gzhytskyi National University of Veterinary Medicine and Biotechnologies, Lviv, Ukraine
E-mail: tsizhb@ukr.net

Tetyana A. Marusenkova
Software Department, Lviv Polytechnic National University, Lviv, Ukraine
E-mail: tetyana.marus@gmail.com

Received: 01 December 2020; Accepted: 14 March 2021; Published: 08 August 2021

Abstract: Inertial measurement units based on microelectromechanical systems are perspectives for motion capture applications due to their numerous advantages. A motion trajectory is restored using a well-known navigation algorithm, which assumes integration of the signals from accelerometers and gyroscopes. Readings of both sensors contain errors, which quickly accumulate due to integration. The applicability of an inertial measurement unit for motion capture depends on the trajectory being tracked and can be predicted due to the simulation of signals from inertial sensors. The first simulation step is prescribing a motion trajectory and corresponding velocities. The existing simulation software provides no user-friendly graphical tools for the completion of this step. This work introduces an algorithm for the simulation of accelerometer signals upon a two-dimensional trajectory drawn with a computer mouse and then vectorized. We propose a modification of the Potrace algorithm for tracing motion trajectories. Thus, a trajectory and velocities can be set simultaneously. The obtained results form a basis for simulating three-dimensional motion trajectories since the latter can be represented by three mutually orthogonal two-dimensional projections.

Index Terms: Inertial Sensor, Simulation, Bezier Curve, Tracing, Two-dimensional Trajectory, Potrace, Accelerometer.

1. Introduction

Inertial measurement units (IMUs) based upon microelectromechanical systems (MEMS) are widely used in a broad range of applications including human-machine interaction [1], sport [2], user authentication [3], robotics, and telemedicine [4]. Unfortunately, MEMS-based IMUs produce inevitably faulty readings [5]. To tackle the problem, different filters have been already designed and new filters are still emerging [6, 7]. The same IMU may be suitable for tracking one kind of motion and insufficient for another. Particularly, fast movements are irreproducible at low sample frequencies [8]. To decide on the utility of a specific IMU for a specific task one may take a real IMU sample, move it, calculate the trajectory and compare the real motion and computed one. However, this method is tedious, time-consuming, and expensive. Simulation of output signals from accelerometers and gyroscopes helps the researchers and engineers save experimental costs both on navigation system construction and filter evaluation.

The basic idea of an IMU simulation tool is as follows. First, the user presets a motion trajectory, velocities, rotation angles, and measurement error characteristics. The tool simulates the signals of sensors and then restores possible trajectories upon these signals. Finally, the software evaluates how much the outcomes deviate from the prescribed trajectory. The level of uncertainty demonstrated by the simulation results characterizes the IMU utility.

Many researchers address the problem of building motion trajectories and simulating IMU signals. However, they provide no simple ways for the user to set a trajectory. A computer mouse is a ubiquitous device that allows us to capture both coordinates and motion velocities in 2D space and thus induces the idea to use it as a tool for setting 2D motion trajectories in IMU simulation software. This study is aimed at the adoption of image tracing techniques for simulating accelerometer signals upon coordinates and velocities provided by drawing with a mouse.

2. Related Work

Mainly researchers' efforts are focused on the enhancement of path-planning techniques for robots and self-driving cars. Robots may be equipped with a broad range of sensors including inertial ones and can be provided with a map of the environment where obstacles are marked. A comprehensive and detailed overview of state-of-the-art algorithms for building a smooth, continuous collision-free motion trajectory has been presented in [9]. The authors mention path smoothing due to Lagrange and Hermite polynomial interpolation and usage of Bezier curves, cubic splines, B-splines, non-uniform rational B-splines, parabolas, ellipses, hyperbolas, cardioids, limacons, hypocycloids, cycloids, pedal curves, spirals, Dubin's curves, etc. A comparison of different techniques is provided. Clothoids have been utilized for path planning in [10, 11]. The authors of [12] apply path planning to unmanned marine vehicles. In [13] Fermat's spirals were utilized. Bezier curves were applied for trajectory construction in [14, 15]. Optimized A* algorithm fused with the Artificial potential field method has been recently introduced in [16]. In [17] a real-time path construction algorithm that ensures avoidance of multiple moving obstacles has been presented. Image processing techniques for the recognition of both steady and moving obstacles are dealt with in [18]. The work uses an optimal Rapidly-Exploring Random Tree algorithm. Path planning algorithms have been increasingly evolving due to the growing popularity of unmanned vehicles. The optimal path planning techniques are of crucial importance for machines but scarcely can be used for humans whose movements are far less predictive. The mathematical background provided is still valuable for the construction of trajectories given a set of points to be interpolated in 3D space.

In [19] the authors propose an analytical IMU signal generation algorithm, which can boast higher accuracy than competitive modern algorithms. The algorithm utilizes dual quaternion interpolation and involves sophisticated equations. A new magnetometer-augmented simulator for generating inertial sensor measurements upon a specific input trajectory has been proposed in [20]. The work emphasizes error simulation and analysis. The architecture of an IMU simulator tool has been described in [21]. An online trajectory generator has been used in [22] as a stage between image processing algorithms and robot motion controllers for controlling robot arms. An IMU simulation framework, NaveGo, has been introduced in [23] and evaluated in [24]. The work provides a comprehensive mathematical model and is corroborated by a freely distributable implementation in MATLAB. However, none of the above-mentioned literature sources assumes any simple graphical tools for the user to set a trajectory.

The authors of [25] have presented a new visualization software, MTTV, intended for trajectory measurement, analysis, comprehension, and manipulation. The work does not deal with velocities or signals of inertial sensors.

The literature review shows a lack of software tools for simulating the signals of inertial sensors upon a user-drawn trajectory, even though path generation itself is a well-elaborated research area.

3. Mathematical Background for Simulating Signals of Inertial Sensors

3.1. General Navigation Algorithm

A triaxial accelerometer measures the acceleration experienced by a body due to both non-gravitational and gravitational forces:

$$\overline{a_i} = \overline{a} - \overline{g}\,, \tag{1}$$

where $\overline{a}$ is the so-called true acceleration, i.e., acceleration with respect to the inertial reference system; $\overline{g}$ is the gravity vector expressed in the inertial reference frame. We assume that this vector is [0 0 *g*], where *g* is approximately 9.81 m/s^2. The vector is written with the minus sign because an accelerometer reacts to the force that prevents it from free-falling, i.e., the force that balances gravity. If the accelerometer moves in a sharp, uneven manner, it will sense both components of acceleration (due to motion and gravity), measured with respect to the inertial frame but expressed in its body frame. A triaxial accelerometer is composed of three one-axis accelerometers whose sensitivity axes are mutually orthogonal. The subscript "i" means "inertial", the subscript "b" denotes "body".

If the true acceleration and initial velocity V_0 are known, then the velocity of the body can be computed as:

$$\overline{v(t)} = v_0 + \int_{t_0}^{t} \overline{a}dt\,. \tag{2}$$

If the initial position $\begin{bmatrix} x_0 & y_0 & z_0 \end{bmatrix}$ is known, the whole trajectory can be determined by the velocity integration:

$$\begin{bmatrix} x(t) & y(t) & z(t) \end{bmatrix} = \begin{bmatrix} x_0 & y_0 & z_0 \end{bmatrix} + \int_{t_0}^{t} \bar{v} dt \ . \tag{3}$$

Formulas (2) and (3) can only be applied if the acceleration is expressed in the same frame. The most convenient way of keeping measurements from the accelerometer aligned is to transform the acceleration from the body frame to the inertial frame.

Presentation of the same (acceleration) vector in different frames can be performed in several ways. Using direction cosine matrices, one applies the following formula:

$$\overline{a_i} = C_b^I \overline{a_b} \ , \tag{4}$$

where $\overline{a_i}$ is the total acceleration vector in the inertial frame, $\overline{a_b}$ is the same vector in the accelerometer body frame, and C_b^I is the direction cosine matrix that relates the body frame to the inertial frame. Matrix C_b^I describes 3 rotations, performed one after another around axes *z*, *y* and *x*. Angles of rotations, ψ about axis *z*, θ about axis *y*, and ϕ about axis *x* are called yaw (heading), pitch, and roll (bank), correspondingly and collectively referred to as Euler or Tait-Bryan angles. When using the direction cosine matrices, it is worth bearing in mind that the order of rotations matters. For instance, rotations around *z*, *y*, and *x*, in general, do not yield the same result as rotations around *x*, *y*, and *z*. Moreover, when the pitch is around ±π/2, a gimbal lock is observed, i.e., one degree of freedom will be lost. On the contrary, quaternions, which are an alternative form of rotation representation, are free from this drawback.

Thus, to use the readings of an accelerometer, one needs to know the orientation (attitude) of the body frame with respect to the inertial frame. Attitude information is commonly provided by a gyroscope. A triaxial rate gyroscope measures the components of the angular velocity vector $\begin{bmatrix} \omega_x & \omega_y & \omega_z \end{bmatrix}$.

If the initial orientation of the body frame with respect to the inertial frame is known, then its current orientation at any time moment can be computed using the Poisson equation:

$$\dot{C}_b^I = C_b^I \begin{pmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{pmatrix}, \tag{5}$$

whose quaternion analog is

$$\dot{q} = \frac{1}{2} q \otimes \omega \ , \tag{6}$$

where q is the current quaternion, $\dot{q}$ is the derivative, ω is the quaternion $\omega = \begin{bmatrix} 0 & \omega_x & \omega_y & \omega_z \end{bmatrix}$, and $\otimes$ denotes the quaternion product.

3.2. Basic algorithm for signals simulation

The signals of inertial sensors can be simulated using (1) – (6) due to the following steps.

1. Prescribe a translational motion scenario. The results of this step are 7 one-dimensional arrays: *T*, which represents the motion time period, *X*, *Y*, and *Z*, which define the *x*-, *y*- and *z*-coordinates of the body at each time moment *t*, and V_x, V_y, V_z which indicate the motion velocity along the three axes. In the simplest case V_x velocity can be calculated as $\left(x_{k+1} - x_k\right)/\,dt$, where $dt = 1/\, f_{sample}$. Other velocity vector components may be computed similarly. The simplest way of setting the *x*-, *y*- and *z*-coordinates of the trajectory is to define analytical dependencies $x(t)$, $y(t)$, and $z(t)$, and perform their time-quantization.

2. Using V_x, V_y, V_z calculate A_x, A_y, A_z which are the acceleration vector components expressed in the inertial frame. The acceleration along the *x*-axis is computed as $A_{x_k} = \left(V_{x_{k+1}} - V_{x_k}\right)/\,dt$, and the two other components are found similarly.

3. Preset a rotational motion scenario.

4. At each step calculate the attitude of the body frame with respect to the inertial frame.

5. Obtain the angular velocity vector. All the angular velocity vector components are stored in three arrays G_x, G_y, G_z. In this way, the signals of a triaxial gyroscope are simulated.

6. Having figured out the current attitude of the body, express the acceleration vector components in the body frame. With direction cosine matrices, one writes:

$$\left[a_{bx},\quad a_{by},\quad a_{bz}\right]=\left(C_{b_{k+1}}^{I}\right)^{T}\left(a_{ix}\quad a_{iy}\quad a_{iz}-g\right). \tag{7}$$

In the case of quaternions, one obtains:

$$\left[0\quad a_{bx}\quad a_{by}\quad a_{bz}\right]=\left(q_{k+1}\otimes\left[0,\quad a_{ix},\quad a_{iy},\quad a_{iz}-g\right]\right)\otimes \overset{*}{q}_{k+1}. \tag{8}$$

In this way, the signals of a triaxial accelerometer are simulated.

To restore the original trajectories, the following steps should be taken.

1. Input simulated signals A_x, A_y, A_z, G_x, G_y, G_z, initial velocities V_{x_0}, V_{y_0} and V_{z_0}, initial coordinates x_0, y_0 and z_0, and initial Euler angles that define the orientation of the body frame with respect to the inertial frame.

2. Calculate the current body attitude.

3. Represent the simulated accelerometer signals in the inertial frame and compensate for gravity.

4. Using (2), obtain the velocity vector.

5. Using (3), obtain the coordinates *x*, *y*, *z*.

3.3. Problems of defining a motion trajectory

The above-mentioned way of presetting analytical solutions for the coordinates and velocities is a textbook case, which may not always be implemented in practice. Firstly, real motion trajectories are not always easily described by analytical dependencies. Of course, the trajectory of a bouncing ball or a body thrown horizontally obeys well-known physical laws and is easy to preset. On the contrary, human motion is far less predictable. Secondly, an object may move along a curve in plenty of ways. For instance, it can move smoothly and uniformly, with a constant velocity, or on the contrary, intermit sharp movements with abrupt halts. Handwriting is an obvious example of such a non-uniform motion. Taking into account these restrictions, the user of the IMU simulation software should be provided with a convenient way of both prescribing motion scenarios and assigning velocities to trajectories. “Convenient” means that the software should require as few actions from the user as possible.

All the motion trajectories fall into three groups: 1-dimensional, 2-dimensional, and 3-dimensional (1D, 2D, and 3D). 3D trajectories are of the highest importance and most complicated for modeling. Though 2D trajectories are of less interest, they are projections of 3D trajectories on the coordinate planes and can also have standalone applications. Some examples of 2D trajectories usage include tracking handwriting (a piece of paper presumably represents a plane) [26]; motion of cars and other vehicles on the road; motion of pedestrians over flat surfaces.

Setting velocities in control points may be cumbersome and time-consuming. Ideally, the user simply draws a trajectory thus simulating motion with a mouse or touchpad, which are parts of PCs/notebooks of practically any configuration. A computer mouse provides *x* and *y* coordinates, and the current cursor positions can be accompanied with timestamps. The combination of these data allows velocities to be computed. The trajectory the user has drawn can be then scaled to desirable physical dimensions, with velocities re-calculated proportionally. Thus, the IMU simulation software should provide a canvas where the user draws as they might do in MS Paint. However, the obtained image is a raster rather than a vector. It is preferable to have a trajectory in its vector form for several reasons. Firstly, in their attempts to communicate the motion temp and character to the software tool the user may obtain uneven, faulty lines, especially in the parts drawn hastily. The opportunity to edit the line would be highly appreciated. Undoubtedly, vector graphics are more pliable to editing than raster ones. Secondly, an increase/decrease of the sampling frequency in the further modeling process makes even piecewise analytical line representation more preferable than a collection of pixels. Thirdly, vector graphics are compact. That is why any user drawing should be traced.

4. Potrace: Limited Applicability for Tracing Motion Trajectories

One of the well-known and widely used tools for tracing a raster image is Potrace, a polygon-based tracing algorithm [27]. The algorithm was introduced by Peter Selinger in 2003. The author has implemented a library available under the GPL license. The functionality was included in Inkscape, a professional vector graphics software tool. Since its first appearance in 2003, the library has evolved in several versions, the latest of which emerged in September 2019. A detailed and comprehensible algorithm description and a 17-year-long history of successful usage and intensive testing suggest that Potrace is a reasonable choice for the presentation of a motion trajectory in its vector form.

The compiled Potrace library is available as well as its source code. With the purpose of code tracing and analysis, one should opt for source code. When working with MS Visual Studio some minor amendments to the source code may be needed (like correcting syntax of inline functions, replacing string comparison functions with their Windows analogs, and changing the way of passing the algorithm parameters to avoid the command line). Potrace accepts a .pnm or a .bmp file and returns a vector graphics file in either of .pdf, .svg, .geojson, .pgm, .xfig, .eps, postscript or .dxf formats depending on the specified parameter.

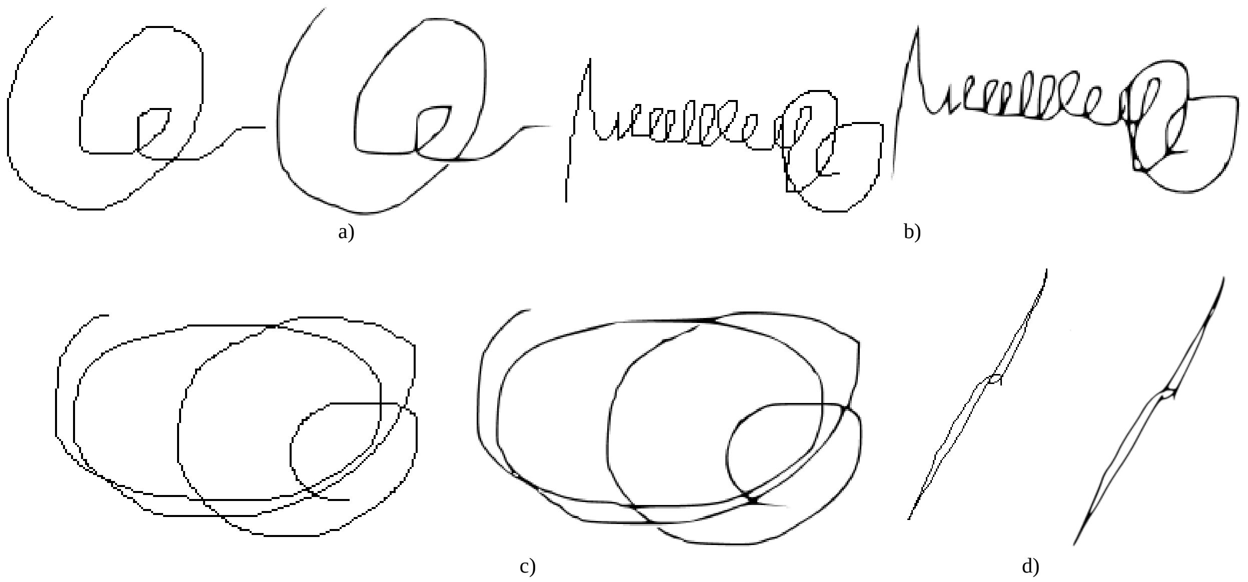
The output file contains curves composed of short quadratic Bezier curves and straight lines. Peter Selinger introduced straight lines ("corners") to prevent the output traced image from looking too rounded. It is possible to dispose of straight lines by setting one of the algorithm parameters, *alphamax*, to any number greater than 4/3. In this case, the output will be comprised solely of Bezier curves. The output file itself is not essential for tracing a motion trajectory. Instead, the resulting Bezier curves should be extracted and processed directly in the IMU simulation software tool, with no metadata imposed by any vector graphics format.

The authors have checked the outcomes of Potrace for a range of input drawings, some of which are illustrated in Fig. 1. We selected the output file format to be .svg. As can be easily seen, the lines originally drawn solidly do not look so any longer after tracing. We must underline that such cases are rather uncommon. We observed them in no more than ten percent of sample drawings. Nevertheless, for tracing motion trajectories no introduction of occasional interruption to originally solid lines can be acceptable because it distorts the velocities meant by the user.

Then we extracted the Bezier curves formed by the Potrace library code. A sample result is shown in Fig. 2. Fig. 2, a shows a bitmap. It is mirrored with respect to the Potrace outcome since the bmp format assumes storing images starting from the bottommost line up; this effect can be easily compensated for. Fig. 2, b depicts the optimized Potrace outcome. Curve optimization is one of the optional algorithm stages that consists of combining several consecutive Bezier curves into a single long curve whenever possible. Fig. 2, c shows the output image somewhat enlarged to scrutinize the details. Fig. 2, d illustrates the enlarged vector outline of the original image with Bezier curve optimization switched off. As one can see, the vector representation of the drawing is far from being solid. It is a collection of curves not connected smoothly and seamlessly end to end. Curve optimization introduces a slight deflection of the original raster image.

Consequently, the Potrace algorithm cannot be applied as-is for tracing a motion trajectory, since one needs an uninterrupted line, which may be piecewise but smooth. Instead, the raster line is approximated by two series of lines, surrounding the original image like "stitches".

This fact is attributed to the very working principle of the algorithm as will be explained further in the text. The shown issues are not the drawback of the algorithm because the latter is intended for creating nice vector outlines of bitmaps, not for tracing trajectories. Nevertheless, Potrace provides Bezier curves that closely fit the original raster images, and it can be seen from the examples shown in Fig. 2 that the algorithm can be applied to the problem of motion trajectory tracing if one manages to overcome the following problems: 1) eliminate the interruptions introduced to lines which have been drawn solid; 2) remove redundant Bezier curves; 3) connect the endpoints of adjacent Bezier curves.



a) b) c) d)

Fig. 1. Four pairs of bitmaps (to the left) and their traced versions (to the right): illustration of interruptions introduced to the solid curves.

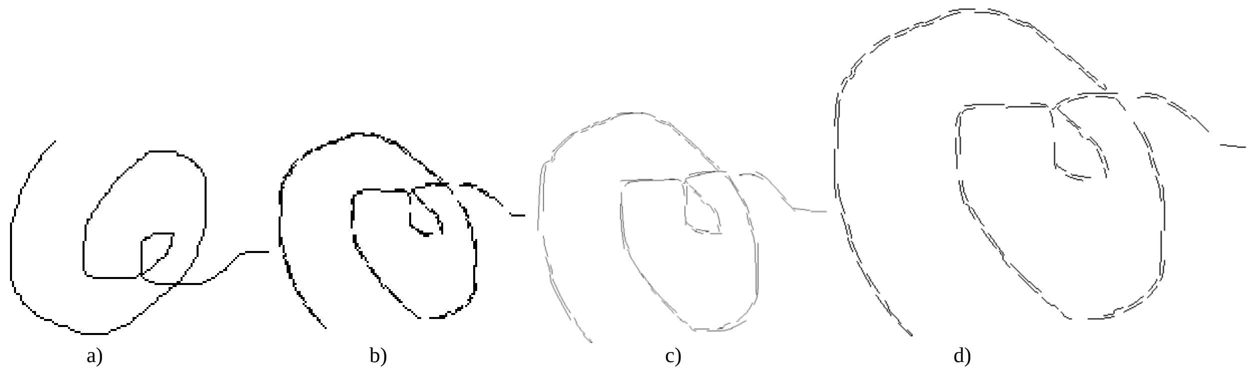


Fig. 2. A bitmap (a) and Bezier curves comprising its traced version: normal size (b), slightly enlarged optimized (c), and enlarged non-optimized (d).

Potrace algorithm has several stages. At the first stage, an input raster image is split into closed paths. A path is a directed graph whose vertices and edges meet the following conditions. A vertex is a point with integer coordinates that is adjacent to four pixels, not all of which have the same color. An edge is a straight line that connects two vertices, separates a black pixel from a white one, and has a length equal to the length (width) of one pixel. The black pixel should be to the left side of the edge when traveling from the current vertex to the next one. Moreover, all the edges in the same path should be distinct. A path is closed if its last vertex coincides with the first one. Fig. 3, a shows a sample closed path, consisting of 21 vertices and 20 edges (gray squares represent pixels).

Path decomposition starts with the selection of a pair of pixels that have different colors (such a pair is found by the library function called *findnext*). Each time Potrace finds a closed path, it is deleted from the copy of the bitmap being processed. The deletion procedure consists of inverting the color of the pixels enclosed by the path so that they turn white. The procedure is repeated until there are no black pixels left in the bitmap copy. The result of the path decomposition stage is a list of closed paths. The library, which is written in C language, utilizes dynamic memory for storing the path list.

Starting with the leftmost black pixel, one obtains the first vertex of the path being constructed. Then one moves along the edges between pixels in such a manner that the black pixel remains to the left of the edge with respect to the motion direction. It is clear from the description that if one constantly moves along a series of black pixels trying to bypass them to their right, they will end up where they have started. However, the choice of the next vertex may be ambiguous. Fig. 3, b illustrates such a situation (the picture is borrowed from Peter Selinger's original paper [27]).

Potrace decides on the next vertex and edge choice based on the selected turn policy. The algorithm supports 7 turn policies: left or right, black or white, minority or majority, and random. Left and black turn policies mean that in any doubtful conditions the left and right turns will be taken correspondingly. Black and white turn policies are aimed at the connection of black and white components respectively. Minority turn policy tends to connect pixels of the rarest color in a given neighborhood. A turn policy is set by a parameter whose default value is a minority. On the contrary, the majority of turn policy opts for pixels met more frequently. Random turn policy, as its name implies, assumes more or less random choices. The turn policy and ambiguity of the next vertex/edge choice are the reasons why we observed line disruptions in Fig. 1.

Tracing the library code proved that the visible disruptions match the pixels in the bitmap where the algorithm opted for a turn opposite to the direction of line drawing. If the left turn policy is chosen, a bitmap will be split into a much greater number of paths (48 instead of 4). The effect of the left turn policy choice is shown in Fig. 4, a. Among our test cases, the no turn policy provided completely acceptable results. The default value, minority, seemed to be the best one.

Hence, to apply Potrace to motion trajectory tracing, one should modify the algorithm in such a way that the only path corresponds to a trajectory provided that the latter is obtained by a continuous mouse moving with its left button kept pressed.

Fig. 4, b shows the results of connecting adjacent Bezier curves within one separate path. If the whole trajectory were represented by a single path, the only step one would need to take is to discard half of the Bezier curves and connect the remaining adjacent curves.

The next stages of the algorithm may remain unaffected. The second stage assumes the approximation of paths by optimal polygons. Among a variety of polygons approximating the same path, those with a fewer number of segments are singled out. Among them, the best variant is chosen using the concept of a penalty. The formulae for the penalty are based on the Euclidean distances. Vertices are no longer required to have integer coordinates – real numbers (type double) are used instead. At the next algorithm phase, the edges of the optimal polygon are replaced by Bezier curves governed by three control points. As has been mentioned previously, the algorithm may not apply a Bezier curve to avoid unnaturally rounded images. On the contrary, Bezier curves may be eliminated if *alphamax* is set to 0. The default value of the parameter is 1 which indicates a possible mix of Bezier curves and corners. The three first stages are the only important for motion trajectory tracing. Then optional curve optimization and building an output file follow. The latter includes scaling and possible rotation.

We omit output graphics files and instead read the formed Bezier curves out of the dynamic memory for further processing.

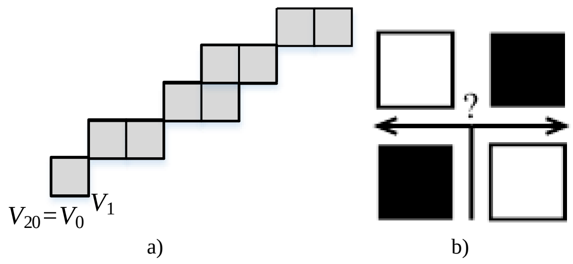


Fig. 3. A sample of a closed path (a) and an ambiguous turn (b).

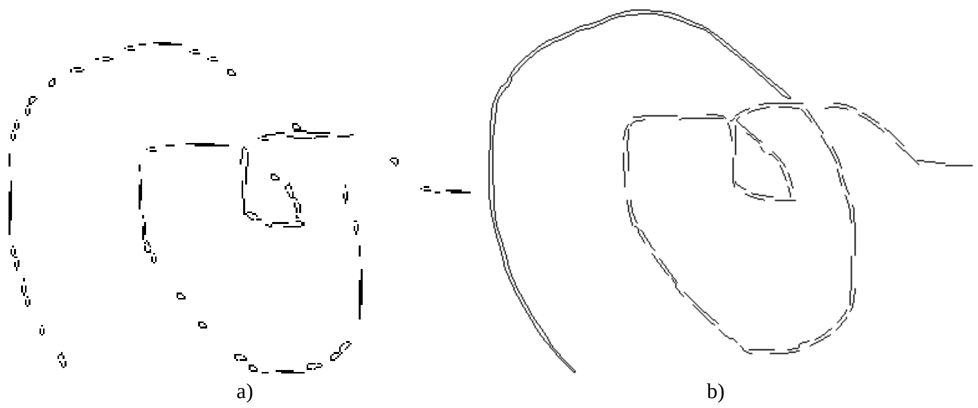


Fig. 4. The result of the left turn policy (a) and Bezier curves in a single path connected end to end (b).

5. A Potrace-based Algorithm: Path Construction

Taking into account all the above-said, we need to modify the first stage of the algorithm. Precisely, we need to avoid any premature turning in a direction different from that defined by the user's motions. The name Potrace is reserved by the author to refer to the original version. Thus we will refer to our modified version as the Potrace-based tracing algorithm. Saving a trajectory into a bitmap file can be omitted – all necessary information can be stored in three arrays – x coordinates, y coordinates, and *timestamps*. The latter is not essential for the Potrace-based algorithm work but is needed for the calculation of velocities. The indices of arrays storing the coordinates reflect the order in which the corresponding pixels were drawn. Since the second and third stages should remain the same as in Potrace, one should construct a closed path fully compliant with what the implementation of the second phase, polygon approximation, takes as its input.

We propose the following algorithm for constructing a path. First let us suppose for simplicity that a curve consisting of black pixels has been drawn in such a manner that each pixel P_i is adjacent both to P_{i-1} and P_{i+1}, where P_{i-1} is the last among all the previously drawn pixels and P_{i+1} is the first among all the pixels drawn after P_i. The only exception is the first and the last pixels in the curve, P_0 and P_n, which lack a predecessor and a successor correspondingly. We assume also, that each pixel has no more than one predecessor and no more than one successor. In other words, a curve is solid and is drawn with a pen of one-pixel width and, consequently, only one pixel can be produced at each moment. Thus, since each pixel has eight possible neighbors, there are eight possible ways to choose the next pixel.

The displacements along axes x and y with respect to the current pixel P_i can be described by eight possible pairs: (–1,–1), (–1,0), (–1,1), (0,–1), (0,1), (1,–1), (1,0), (1,1). Four pairs, (–1,0), (0,–1), (0,1), (1,0), represent horizontal or vertical displacements, whereas the remaining four pairs indicate diagonal displacements. Since the path can contain only edges of length 1, diagonal movements should be denoted by two edges. For instance, a displacement (1,1) should be represented either as a displacement (0,1) followed by (1,0) or as a displacement (1,0) followed by (0,1). Put simply, we either move first right, then up, or vice versa. The order matters, which is demonstrated in Fig. 5, a. The variant Fig. 5, b outlines the black pixels closer than Fig. 5, a. (black pixels are displayed in a lighter color for illustrative purposes).

We start with the left bottom corner of a pixel. Then any steps in a horizontal or vertical direction and any pairs of steps representing a diagonal motion end up in the left bottom corner of the next pixel. Let us consider four possible diagonal movements depicted in Fig. 6. In cases a) and d) there is no ambiguity: for case a) first one step up, then one step to the left should be taken whereas for case d) first one step to the right then one step down should be done. With cases b) and c) any order of steps outline the black pixels equally close. We opt for the original Potrace logic: we leave the black pixel to the left. Thus, in case b) first, one step to the right is taken followed by one step up; in case c) one step to the left is followed by one step down. Hence, we move along the horizontal direction first in any case except case a). Starting with the first pixel in the curve, a path is being constructed until the last pixel, P_n, is reached.

In this way, the meaningful part of the path is obtained, which follows the original curve closely. Its approximation by Bezier curves will be used for further editing. However, a path for the next algorithm stage should be closed. Consequently, one needs to construct the remaining part of the path to comply with the logic of the algorithm. A simple way of building the remaining part of the path is to take each newly constructed edge of the meaningful part of the path and increase its y-coordinate by 1. In this way the set of edges displaced one pixel up will fit the upper corners of pixels comprising the curve. What remains is to reverse the order of vertices in this mirrored part of the path and make sure that $y_n = y_0$ as it is required by Potrace. The proposed algorithm violates some of the requirements of Potrace. Not all black pixels are always kept to the left of an edge. However, this breach of the path construction rule is compensated for by the way in which the backward route (from P_n to P_0) is built. Besides, an occasional shift by the width of one pixel will not be noticeable.

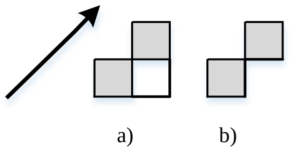


Fig. 5. Importance of selecting the right order.

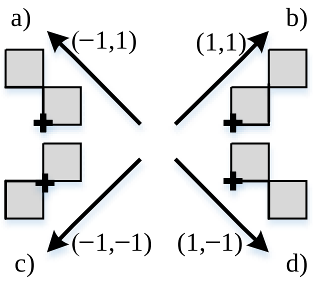


Fig. 6. Order of edges for diagonal steps.

Our initial assumptions about the curve may not be met. Firstly, the curve will scarcely consist of adjacent pixels. In practice, sharp movements of a mouse mean that either $|x_{i+1} - x_i|$ or $|y_{i+1} - y_i|$, or adjacent coordinates in x and y directions simultaneously differ more than by one pixel. It can be observed if current mouse coordinates are shown rather than a line connecting consecutive points. To fill in the missing points, we apply Bresenham's algorithm to each consecutive ~~pairs~~ pair of the mouse coordinates. With the purpose to avoid replication of the same vertices, the starting point fed to Bresenham's algorithm is not included in the computed array of points that comprise the line defined by its start and endpoints. The very first point in the curve should be an exception to this rule. Then we check that the conditions are fulfilled for any valid index value.

Our next assumption that the next pixel is one of the eight possible neighbors of the current pixel may not be fulfilled as well. Firstly, there can be the absence of motion, i.e., the user was keeping the left mouse button pressed in the same position. It corresponds to step (0,0); no edge is involved. In this case, all the pixels where the user's hand rested are ignored when constructing the path but taken into account when calculating the velocities. Secondly, there can be returning to the previous pixel, i.e., P_i is followed by P_{i+1}, which is followed again by P_i. In this case, the requirement of non-repeated edges is violated. To avoid this situation a simple check for such "pixel loops" should be added. If the previous and the next pixels coincide, then ignore the current pixel but take into account the corresponding timestamp.

A more elaborate way of handling such situations may be invented. However, occasional discarding of one pixel would not affect the whole trajectory much and no additional complexity is introduced. The described algorithm for building a Potrace-compliant path upon arrays of mouse coordinates is illustrated in Fig. 7.

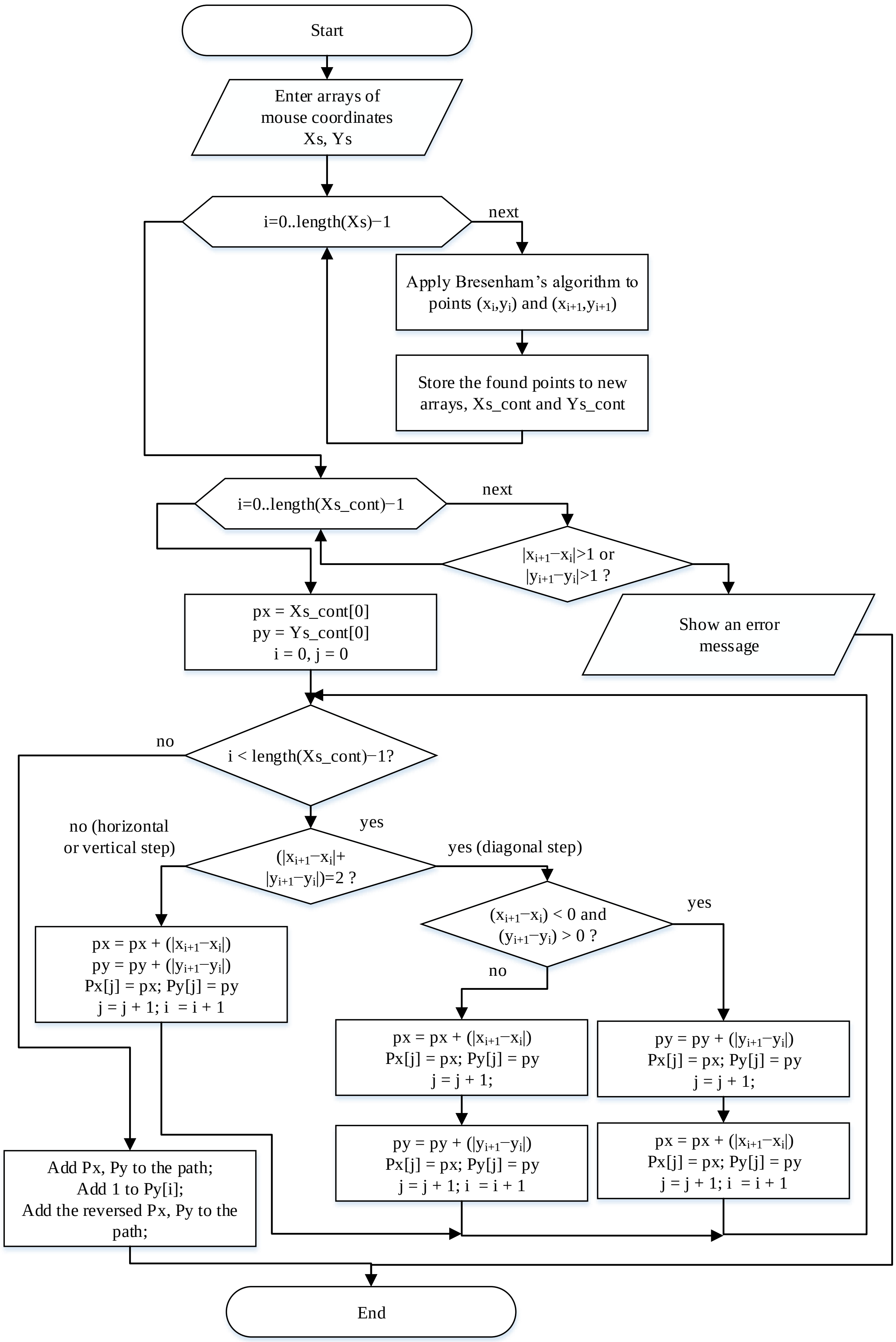

Start
Enter arrays of mouse coordinates Xs, Ys
i=0..length(Xs)−1
next
Apply Bresenham's algorithm to points (x_i,y_i) and (x_{i+1},y_{i+1})
Store the found points to new arrays, Xs_cont and Ys_cont
i=0..length(Xs_cont)−1
next
$|x_{i+1}-x_i|>1$ or $|y_{i+1}-y_i|>1$?
Show an error message
px = Xs_cont[0]
py = Ys_cont[0]
i = 0, j = 0
no
i < length(Xs_cont)−1?
yes
no (horizontal or vertical step)
$(|x_{i+1}-x_i|+|y_{i+1}-y_i|)=2$?
yes (diagonal step)
$(x_{i+1}-x_i) < 0$ and $(y_{i+1}-y_i) > 0$?
yes
no
px = px + ($|x_{i+1}-x_i|$); py = py + ($|y_{i+1}-y_i|$); Px[j] = px; Py[j] = py; j = j + 1; i = i + 1
px = px + ($|x_{i+1}-x_i|$); Px[j] = px; Py[j] = py; j = j + 1;
py = py + ($|y_{i+1}-y_i|$); Px[j] = px; Py[j] = py; j = j + 1; i = i + 1
py = py + ($|y_{i+1}-y_i|$); Px[j] = px; Py[j] = py; j = j + 1;
px = px + ($|x_{i+1}-x_i|$); Px[j] = px; Py[j] = py; j = j + 1; i = i + 1
Add Px, Py to the path; Add 1 to Py[i]; Add the reversed Px, Py to the path;
End

Fig. 7. Block diagram of the path forming part of the Potrace-based algorithm for tracing a 2D trajectory.

6. Algorithm for Simulating Signals of Inertial Sensors

The general algorithm for presetting a 2D motion trajectory and simulating signals of inertial signals is comprised of the following steps.

1. Provide the user with a tool for drawing. In MS Visual Studio a PictureBox graphic control can be used. Tracking mouse coordinates starts with the *onmousedown* event and ends with *onmouseup*. In the handler of the *onmousemove* event, new mouse coordinates are captured along with their timestamp expressed as the number of milliseconds elapsed since the previous handler invocation and, consequently, since the previous mouse coordinates. Both coordinates and their timestamps are stored in their corresponding arrays. As has been mentioned earlier, one should not expect adjacent mouse coordinates between consecutive calls of the handler. Lines may be displayed using the in-built functions of the graphic library (e.g., DrawLine), but the missing points are to be calculated by some auxiliary function.

2. Apply the above-proposed algorithm to build a path required by the Potrace-based algorithm. We have used the function *bm_to_pathlist* of the Potrace library. Inside this function, we save our path in *p->priv->pt*, an array of points in the dynamic memory. The array mostly complies with an original Potrace path constructed by analyzing a bitmap. As has been discussed previously, the differences will not affect the success of the rest of the algorithm.

3. Read the Bezier curves formed by the well-tested functions of the original Potrace library. They are saved in the field *dpoint_t (*c)[3]*; of the structure *privcurve_t*; *c[n][i]* stores three control points of each Bezier curve. We get curves from the variable *curve* in function *smooth*.

4. Only the curves that fit the path from the starting and ending points of the bitmap line are needed. The rest of the curves result from the working principle of the algorithm and can be discarded since they are useless for the problem of tracing a motion trajectory.

5. Connect all the Bezier curves end to end. I.e., the coordinates of the first control point of each Bezier curve except the first one are set to the coordinates of the last control point in the previous Bezier curve. In this way, one obtains a smooth line with no interruptions.

6. Provide the user with tools for editing the vector curve. New graphic controls are generated with this purpose (basically any control that can handle mouse events may be used). Each graphic control is assigned coordinates that coincide with the coordinates of a control point of a Bezier curve. The user drags and drops the graphic control whose new coordinates are assigned to the corresponding control point. Potrace produces lots of short Bezier curves, which makes the creation of graphic controls for each possible control point pointless. We supplied each tenth Bezier curve with a graphic control. The editing policy is to be elaborated better and this is the subject for future improvements.

7. Find the correspondences between the control points of all the Bezier curves and the original timestamps associated with the mouse coordinates. Now that all the control points match the original timestamps, the velocity in any interim point of any curve can be obtained as the result of interpolation. The Bezier curve parameter, *t*, is used for this purpose. Thus, one can “sample” the values of the velocity vector from the curve with practically any sample frequency. After the traced curve has been edited, all the velocities should be re-calculated.

8. Using the algorithm described in Section II, the output signals of an accelerometer are simulated. Signals of a gyroscope can be modeled independently since translational and rotational motions are independent. Convenient ways of prescribing rotational motion scenarios are the subject of our future work.

7. Results Evaluation

The history of the Potrace library and its internal structure show that a decent tracing algorithm implementation takes months of development time and so does bug fixing. That was the reason why the author opted to analyze and modify an existing library instead of writing its code from scratch. Potrace library is comprised of several dozen files containing hundreds of lines of code each, thus overcoming Linux-to-Windows transition issues and differences in compiler versions and further adoption of the library to tracing trajectories seemed a reasonable solution. The library’s basic code has been available for free downloading for more than a decade. Moreover, it is included in a popular vector graphics editor. Both facts imply that the library has been well-tested and most of its early defects have been already found and fixed.

We have modified only the library functions responsible for the first stage of the original Potrace library. The rest of the code has remained untouched and thus did not require retesting. The output of the functions we did modify is completely compliant with the functions implementing the second stage of Potrace, i.e., formally the same data structures are used. The only difference is that the output of the first algorithm stage is formed differently in our Potrace-based algorithm. For this reason, we performed intensive unit testing of the modified functions and integration testing of the functions implementing the first and the second algorithm stages. Thus, the proposed Potrace-based algorithm inherits the code quality of Potrace and the introduced minor changes have been carefully tested.

We can state the following advantages of the proposed algorithm.

It is fully suitable for tracing motion trajectories. It is vital that if the user meant a continuous trajectory, no disruptions are introduced as a result of the tracing procedure. If this requirement is not met, motion velocities would not be properly calculated. Potrace is aimed at the creation of good-looking vector outlines of any picture and does not care about line continuity as long as it does not impair the visual attractiveness of the result. Our modification is intended for the transformation of a raster line into a continuous editable vector line.

No BMP files are needed for storage of interim results – mouse coordinates are saved directly into dynamic arrays. Thus, the algorithm dispenses with the necessity to analyze an input file, read its metadata, detect the raster graphics format, its version, color scheme, width, and height.

No output vector graphics is needed. A set of library functions for the creation of an output file can be omitted. No metadata is required. Instead, a collection of Bezier curves should be stored. Each Bezier curve is described by six float numbers. Since all the points are connected end to end, only the endpoints of each curve should be saved, which saves memory.

After creation, a curve can be edited by drag-and-dropping markers. How to select a reasonable amount of markers and place them along the curve is subject to improvement. The possibility to edit a curve after it has been created is vital for tracing trajectories because when one draws a line hastily, the result would unlike look like what the user imagined. Nice-looking lines typically result from slow, careful drawing. The intentionally slowed drawing manner contradicts our idea that motion coordinates and velocities should be set simultaneously, i.e., that the drawing manner reflects the simulated motion speed.

Finally, the algorithm uses a computer mouse, which is a both ubiquitous and rather precise device, as a simulator of a two-axial accelerometer. We are aware that mouse coordinates are not ideal, but computer mice are far less erroneous than accelerometers and they are available for every computer user. The very idea of using computer mice for positioning is not novel, but its combination with Potrace and accelerometer signal simulation comprises the novelty of this work.

8. Conclusions

A problem of modeling the signals of inertial sensors has been discussed. This large problem has been decomposed into several smaller problems, one of which is prescribing motion scenarios. A motion scenario involves a trajectory itself and associated velocities and angular velocities. For 2D trajectories, drawing a curve with a mouse is a desirable way of prescribing a motion scenario since it allows the user to set both coordinates and velocities simultaneously by the same action. For further processing, the drawn curve should be traced.

It is shown that a famous tracing algorithm, Potrace, and its reliable implementation cannot be used directly for tracing a motion trajectory. A Potrace-based algorithm has been proposed to overcome the limited utility of the original Potrace for tracing 2D motion trajectories.

There are several ways of improvement. Firstly, the resulting vector outline consists of multiple short Bezier curves, and an efficient approach to editing the outline should be developed. Secondly, drawing a 2D trajectory and then tracing it deals with the coordinates and velocities but not with rotational motion. A convenient way of presetting body attitudes is yet to be found.

The proposed algorithm has been successfully verified for a variety of 2D trajectories. The results can be used also for editing 3D trajectories if one considers a 3D trajectory as its projections onto three orthogonal planes. The described algorithm is intended primarily for human motion description since path planning for machinery is a well-elaborated area.

The obtained results promote further development of IMU simulation software, shifting it from sophisticated equations to a simple graphical interface for prescribing trajectories and associated velocities.

References

[1] M. El-Gohary, "Joint Angle Tracking with Inertial Sensors", Ph.D. dissertation, Portland State University, Portland, USA, 2013. doi: 10.15760/etd.661.

[2] A. Bulling, U. Blanke, and B. Schiele, "A tutorial on human activity recognition using body-worn inertial sensors", ACM Computing Surveys, vol. 46, no. 3, pp. 1-33, 2014. doi: 10.1145/2499621.

[3] T. Ngo, Y. Makihara, H. Nagahara, Y. Mukaigawa, and Y. Yagi, "Similar gait action recognition using an inertial sensor", Pattern Recognition, vol. 48, no. 4, pp. 1289-1301, 2015. doi: 10.1016/j.patcog.2014.10.012.

[4] C. Chen, R. Jafari and N. Kehtarnavaz, "A survey of depth and inertial sensor fusion for human action recognition", Multimedia Tools and Applications, vol. 76, no. 3, pp. 4405-4425, 2015. doi: 10.1007/s11042-015-3177-1.

[5] S. Darooghеha, T. Lasky, and B. Ravani, "Position Measurement Under Uncertainty Using Magnetic Field Sensing", IEEE Transactions on Magnetics, vol. 54, no. 12, pp. 1-8, 2018. doi: 10.1109/tmag.2018.2873158.

[6] Y. Shmaliy, S. Zhao, and C. Ahn, "Optimal and Unbiased Filtering with Colored Process Noise Using State Differencing", IEEE Signal Processing Letters, vol. 26, no. 4, pp. 548–551, 2019. doi: 10.1109/LSP.2019.2898770.

[7] X. Lin, Y. Jiao, and D. Zhao, "An improved Gaussian filter for dynamic positioning ships with colored noises and random measurements loss", IEEE Access, vol. 6, pp. 6620–6629, 2018. doi: 10.1109/ACCESS.2018.2789336

[8] Dmytro V. Fedasyuk, Tetyana A. Marusenkova, "An Algorithm for Detecting the Minimal Sample Frequency for Tracking a Preset Motion Scenario", International Journal of Intelligent Systems and Applications (IJISA), Vol.12, No.4, pp.1-12, 2020. DOI: 10.5815/ijisa.2020.04.01.
[9] A. Ravankar, A. Ravankar, Y. Kobayashi, Y. Hoshino, and C. Peng "Path Smoothing Techniques in Robot Navigation: State-of-the-Art, Current and Future Challenges", Sensors, Vol. 18, 3170, 2018. DOI: doi:10.3390/s18093170.
[10] S. Gim. Flexible and Smooth Trajectory Generation based on Parametric Clothoids for Nonholonomic Car-like Vehicles. Automatic. Universit éClermont Auvergne; Sung Kyun Kwan university (S éoul), 2017.
[11] M. Brezak, and I. Petrović, "Path Smoothing Using Clothoids for Differential Drive Mobile Robots", in Proceedings of the 18th World Congress The International Federation of Automatic Control, Milano, Italy, 2011, pp. 1133–1138. DOI: 10.3182/20110828-6-IT-1002.02944.
[12] C. Xiong, D. Chen, D. Lu, Z. Zeng, and L. Lian, "Path planning of multiple autonomous marine vehicles for adaptive sampling using Voronoi-based ant colony optimization", Robotics and Autonomous Systems, Vol. 115, 2019, pp. 90–103. DOI: 10.1016/j.robot.2019.02.002.
[13] X. Chen, J. Zhang, M. Yang, L. Zhong, J. Dong, "Continuous-Curvature Path Generation Using Fermat's Spiral for Unmanned Marine and Aerial Vehicles", in Proceedings of the 2018 Chinese Control and Decision Conference (CCDC), Shenyang, China, 2018; pp. 4911–4916.
[14] K. Kawabata, L. Ma, J. Xue, C. Zhu, N. Zheng, "A Path Generation for Automated Vehicle Based on Bezier Curve and Via-points", Robotics and Autonomous Systems, No. 74. pp. 243–252, 2015. DOI: 10.1016/j.robot.2015.08.001.
[15] Z. Liang, G. Zheng, J. Li, "Automatic Parking Path Optimization based on Bezier Curve Fitting", in Proceedings of the 2012 IEEE International Conference on Automation and Logistics, Zhengzhou, China, 2012, pp. 583–587.
[16] Ashutosh Kumar Tiwari, Sandeep Varma Nadimpalli, " New Fusion Algorithm Provides an Alternative Approach to Robotic Path Planning", International Journal of Information Engineering and Electronic Business (IJIEEB), Vol.12, No.3, pp. 1-7, 2020. DOI: 10.5815/ijieeb.2020.03.01.
[17] Vanitha Aenugu,Peng-Yung Woo,"Mobile Robot Path Planning with Randomly Moving Obstacles and Goal", International Journal of Intelligent Systems and Applications(IJISA), vol.4, no.2, pp.1-15, 2012. DOI: 10.5815/ijisa.2012.02.01.
[18] Roudabe Seif, Mohammadreza Asghari Oskoei,"Mobile Robot Path Planning by RRT* in Dynamic Environments", International Journal of Intelligent Systems and Applications (IJISA), vol.7, no.5, pp.24-30, 2015. DOI: 10.5815/ijisa.2015.05.04.
[19] K. Liu, W. Wu, K. Tang, and L. He, "IMU Signal Generator Based on Dual Quaternion Interpolation for Integration Simulation", Sensors, Vol. 18, 2721, 2018. DOI: doi:10.3390/s18082721.
[20] T. Brunner, J. Lauffenburger, S. Changey, and M. Basset, "Magnetometer-Augmented IMU Simulator: In-Depth Elaboration", Sensors, Vol. 15, pp. 5293–5310, 2015. DOI: doi:10.3390/s150305293.
[21] M. Parés, J. Rosales, I. Colomina, "Yet Another IMU Simulator: Validation and Applications" [online]: http://www.isprs.org/proceedings/2008/euroCOW08/euroCOW08_files/papers/20.pdf, last accessed January 2021.
[22] T. Kr öger, J. Padial, "Simple and robust visual servo control of robot arms using an on-line trajectory generator", in Proceedings of the 2012 IEEE International Conference on Robotics and Automation, Saint Paul, MN, USA, 2012, pp. 4862–4869. DOI: 10.1109/ICRA.2012.6225175.
[23] R. Gonzalez, J. Giribet, H. Pati ño, "NaveGo: A simulation framework for low-cost integrated navigation systems", Control Engineering and Applied Informatics, Vol. 17(2), 2015, pp. 110-120.
[24] R. Gonzalez, C. Catania, P. Dabove, J. Taffernaberry, M. Piras, "Model Validation of an Open-source Framework for Post-processing INS/GNSS Systems", in Proceedings of the 3rd International Conference on Geographical Information Systems Theory, Applications and Management (GISTAM 2017), Porto, Portugal, 2017, pp. 201-208. DOI: 10.5220/0006313902010208.
[25] F. Poiesi, and A. Cavallaro, "MTTV - An Interactive Trajectory Visualization and Analysis Tool" in Proceedings of the 6th International Conference on Information Visualization Theory and Applications (IVAPP-2015), pp. 157–162. DOI: 10.5220/0005311001570162.
[26] Y. Li, K. Yao and G. Zweig, "Feedback-based handwriting recognition from inertial sensor data for wearable devices", in 2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), Brisbane, QLD, Australia, 2015. doi: 10.1109/icassp.2015.7178375.
[27] P. Selinger, "Potrace: a polygon-based tracing algorithm" [online]: http://potrace.sourceforge.net/potrace.pdf, last accessed November 2020.

Authors' Profiles

Bohdan R. Tsizh, Dr. Sc. professor, Head of Department of General Technical Subjects, Stepan Gzhytskyi National University of Veterinary Medicine and Biotechnologies, Lviv, Ukraine. Area of scientific interests: Electronics, Sensors Thin-film Heterostructures of Organic and Non-Organic Semiconductors

Tetyana A. Marusenkova, Ph.D., Associate Professor at Software Department, Lviv Polytechnic National University, Lviv, Ukraine. Area of scientific interests: Mathematical Modelling, Inertial Navigation, Data Fusion Algorithms, Embedded Software

How to cite this paper: Bohdan R. Tsizh, Tetyana A. Marusenkova, " A Potrace-based Tracing Algorithm for Prescribing Two-dimensional Trajectories in Inertial Sensors Simulation Software ", International Journal of Modern Education and Computer Science(IJMECS), Vol.13, No.4, pp. 42-54, 2021.DOI: 10.5815/ijmecs.2021.04.04