Structural Analysis of Automating Measurements of Floor Gradients

by

Noah S. Caplan

B.S. Mechanical Engineering
Massachusetts Institute of Technology, 2011

SUBMITTED TO THE DEPARTMENT OF MECHANICAL ENGINEERING IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

BACHELOR OF SCIENCE IN MECHANICAL ENGINEERING AT THE MASSACHUSETTS INSTITUTE OF TECHNOLOGY

JUNE 2011

© 2011 Noah S. Caplan. All rights reserved.

The author hereby grants to MIT permission to reproduce and to distribute publicly paper and electronic copies of this thesis document in whole or in part in any medium now known or hereafter created.

Signature of Author: 

Department of Mechanical Engineering
June 2011

Certified by: 

Dr. Barbara Hughey
Thesis Supervisor

Accepted by: 

John H. Lienhard
Samuel C. Collins Professor of Mechanical Engineering
Undergraduate Officer
Structural Analysis of Automating Measurements of Floor Gradients

by

Noah S. Caplan

SUBMITTED TO THE DEPARTMENT OF MECHANICAL ENGINEERING IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

BACHELOR OF SCIENCE IN MECHANICAL ENGINEERING

0. Abstract

It is useful for one owning or buying a house to be able to assess its structure and identify the existence and severity of any damage. No previously existing method appears to make this assessment easily available. This thesis predicts that architecture will fail in some combination of eleven predictable ways that a simple robot can observe and distinguish by measuring the slope of select points on the floor. This prediction was tested on a case study house, and the model predicted 78.7% of the observed contour. A compact robot was fabricated and measurements of inclination were compared with those of a standard digital inclinometer. The ratio of the angle measured with the robot to that measured with the inclinometer was found to be 1.034 ± 0.193. This proof-of-concept study indicates that an inexpensive robot could be developed as a commercial product capable of assessing the structural safety of common houses.

Thesis Supervisor: Dr. Barbara Hughey
# Table of Contents

- Abstract                                                                 2
- Contents                                                                 3
- Definition of terms                                                       4
- 1. Introduction                                                          4
- 2. Structural analysis of a house                                        4
  - 2.0 Structural model of a house                                         5
  - 2.1 Analysis of structural model of a house                            6
  - 2.2 Gradient and Contour Maps                                           17
  - 2.3 Manual Analysis                                                     20
  - 2.4 Validation of model                                                 21
- 3. Automated Gradient Map Measurement                                     22
  - 3.0 Hardware                                                            22
  - 3.1 Control algorithms                                                  24
  - 3.2 Testing                                                             25
- 4. Conclusions                                                           26
- 5. References                                                            26
- Appendix: software                                                       27
1. Introduction

This thesis demonstrates how the structural stability of a building can be estimated using measurements of the slope of its floors. This method for structural analysis is inexpensive and non-damaging. Section 2 defines the analysis process that makes this possible. Section 3 considers the automation of the floor measurement process process and describes the results of proof-of-concept measurements using a compact robot.

2. Structural analysis of a house
This section focuses on the structural analysis process. Subsection 2.0 defines the theoretical model used to represent a building. Subsection 2.1 defines the shape of floors predicted by that theoretical model. Subsection 2.2 describes the numerical finite element model used for analysis in practice. Subsection 2.3 briefly summarizes an example use of this analysis. Subsection 2.4 demonstrates how the reliability of results obtained from theoretical analysis can be verified and does so for the example use.

2.0 Structural model of a house

Gradient-based structural analysis depends on an assumed model of the underlying structure. A house could, in principle, be deliberately constructed with uneven floors, heavy weights hidden in the ceiling, or anything else not anticipated by the author. To avoid application of this analysis technique in an inappropriate context, it is necessary to define the model around which this algorithm was developed.

The structural component of a house is considered to have three parts: foundation, skeleton, and surfaces.

A correctly built foundation provides a level surface. The most likely source of failure of the foundation is the earth it rests on. If the ground's support fails, the foundation ceases to be level. Several familiar examples can be found in Pisa, Italy.

A skeleton is composed of straight beams. It is entirely or almost entirely supported by the foundation. All horizontal beams rest at a height of an integer number of floors. At the end of every beam there is at least one other beam or at least two other beams' ends. A correctly built skeleton arranges the beams orthogonally. The most likely failures are deformations due to gravity: axial strain in the skeleton's vertical beams and transverse strain in the skeleton's horizontal beams. Figure 1 shows a simple example of a skeleton.
FIGURE 1: Skeleton model of a house's architectural structure.

The surfaces of a house consist of floors, walls, and ceilings. Because they don’t need to meaningfully contribute to structural stability, surfaces tend to be built very weak relative to the rest of the skeleton. Surfaces have uniform thickness. The most likely failure in floors, sagging, is also due to gravity. Deformations in the walls are both negligible and irrelevant to the analysis discussed here.

2.1 Analysis of structural model of a house

The typical failure modes of the model described in section 2.0 all predict floor shape. To analyze these shapes, two maps will be used: the gradient map and the contour map. The gradient map, $\nabla h(x,y)$, denotes the slope of the floor at $(x,y)$. The gradient map is – as its name implies – the gradient of the continuous contour map. The contour map, $\delta h(x,y)$, denotes the difference between the height of a particular point $(x,y)$ on the floor from a constant height. In practice, the gradient map is easy to measure but hard to interpret meaningfully. Instead, the gradient map is
used to compute the easier to interpret contour map.

To interpret the contour map computed from the gradient map, it is necessary to know what contour maps typical failure modes predict. The model of the house's structure predicts five likely structural failures: weak walls, uneven foundation, sagging floorboards, transverse strain in a skeleton's horizontal beam, and axial strain in a skeleton's vertical beam. Because the deformations in the floor are small compared to its size, it is reasonable to assume that the floor’s bending moment does not vary due to curvature in the orthogonal direction. This implies that it is approximately accurate to use the one-dimensional linear elastic sheet-bending model\textsuperscript{1}.

\begin{align}
\frac{\partial^2 \delta h}{\partial x^2} &= \frac{M_x}{EI_x} \\
\frac{\partial^2 \delta h}{\partial y^2} &= \frac{M_y}{EI_y}
\end{align}

where $\delta h(x,y)$ is the height of the floor above some reference point, $M_x$ and $M_y$ are the floor’s internal bending moments, $I_x$ and $I_y$ are the floor’s area moments of inertia, and $E$ is the Young’s modulus of the material used to construct the floor. Each of the five failure modes listed above has its own distinctive corresponding contour map, the first’s being flat, and thus irrelevant to the present discussion.

The uneven foundation is the simplest structural failure to detect. If $\vec{r}$ denotes the direction in which the foundation is inclined (the “uphill” direction), the contour map it produces is trivial.

\begin{equation}
\delta h_{\text{foundation}}(\vec{\rho}) \propto \vec{\rho} \cdot \vec{r}
\end{equation}

where $\vec{\rho} = (x, y)$ is an arbitrary position on the floor.

A sagging floor produces two contour maps. One is based on the assumption that the joints connecting the skeleton’s beams force them to remain orthogonal; the other is based on the
assumption that the joints impose no such restriction. In reality, all joints fall between these extremes, so the true result is a superposition of these two cases with variable amplitudes.

In the case of joints that do not mechanically impose orthogonality, the mechanical analysis is straightforward. A rectangular area of the floor supported at all four corners has less support in one corner than the others. The floor hangs entirely under its own weight. Assuming the floor’s thickness to be uniform implies both moments are given by

$$M_{x, \text{non-orthogonal}} = \frac{\sigma y_0}{2} (x_0 x - x^2)$$

$$M_{y, \text{non-orthogonal}} = \frac{\sigma x_0}{2} (y_0 y - y^2)$$

where $\sigma$ is the surface density (in mass/area) of the floor and $x_0$ and $y_0$ are the dimensions of the floor.

The general solution to equations (1), (2), (4), and (5) is

$$\delta h_{\text{sag, non-orthogonal}} = \frac{\sigma g}{24EI} \left(2x_0 x^3 - x^4 + 2y_0 y^3 - y^4\right) + Ax + By + Cxy + D$$

$$\bar{I} = \frac{I_x}{y_0} = \frac{I_y}{x_0},$$

where $A - D$ are constants dictated by the skeleton. This can be solved for zero displacement at the corners.

$$\delta h_{\text{sag, non-orthogonal}} \propto -x^4 + 2x_0 x^3 - x_0^3 x - y^4 + 2y_0 y^3 - y_0^3 y$$

This solution is presented graphically in Figure 2.
FIGURE 2: Graphical representation of equation (8): a floor sagging under its own weight supported by joints that do not impose orthogonality. Note that the horizontal members are not perpendicular to the vertical axis at the corners. The $x$ and $y$ axis are in units of $x_0$ and $y_0$, respectively, the floor’s dimensions. The maximum magnitude of the vertical deflection is $\frac{5\sigma g}{384EI}(x_0^4 + y_0^4)$, exaggerated here to two to three orders of magnitude greater than typical deformations.

In the case of joints that do impose orthogonality, the one-dimensional linear elastic sheet-bending model remains valid. The internal moment modeled must include arbitrary moment at each corner.
The general solution to equations (1), (2), (9), and (10) is

\[
\delta h_{\text{sag, orthogonal}} = -\frac{\sigma g}{24EI} \left( x^4 + y^4 \right) + A x^3 y + B x y^3 + C x^3 + D y^3 + E x^2 y + F x y^2 + G x^2 + H y^2 + I x + J y + K x y + L
\]

This can be solved for displacements and gradients of zero at the corners.

\[
\delta h_{\text{sag, orthogonal}} \propto \left( x^2 - x x_0 \right)^2 - \left( y^2 - y y_0 \right)^2
\]

This solution is presented graphically in Figure 3.

**FIGURE 3:** Graphical representation of equation (12): a floor sagging under its own weight supported by joints that impose orthogonality. The axes are normalized as described in the caption to Figure 2. The maximum magnitude of the vertical deflection here is \(\frac{5 \sigma g}{384EI} (x_0^4 + y_0^4)\).
Transverse strain in a skeleton’s horizontal beam also generates two contour maps, one for orthogonality imposing and one for non-orthogonality-imposing joints. In both cases, however, the beam deforms exactly according to the one-dimensional linear elastic sheet-bending model; there is no geometric distinction between a floor sagging under its own weight and a floor resting on horizontal beams that are sagging under their own weight.

Axial strain in a vertical beam of the skeleton also produces two contour maps. In the case of joints that do not mechanically impose orthogonality, the general form of the contour map remains that given in equation (6). This can be solved for the origin’s unit displacement.

\[
\delta h_{00, \text{non-orthogonal}} = \frac{\sigma g}{24EI} \left( -x^4 + 2x_0x^3 - x_0^3x - y^4 + 2y_0y^3 - y_0^3y \right) + \left[ \frac{x_0 - x}{x_0} \frac{y_0 - y}{y_0} \right] \tag{13}
\]

The first term is the shape of the floor sagging under its own weight; the second term is the contribution of the origin’s unit displacement. It follows that the component of the contour map generated by axial strain in a vertical beam of the house’s skeleton under the assumption that joints impose no orthogonality between beams is the second term, as one would expect.

\[
\delta h_{00, \text{non-orthogonal}} \propto \left[ \frac{x_0 - x}{x_0} \frac{y_0 - y}{y_0} \right] \tag{14}
\]

This solution is presented graphically in Figure 4.
FIGURE 4: Graphical representation of equation (14): a non-sagging floor supported by uneven joints that do not impose orthogonality. The axes are normalized as described in the caption to Figure 2.

The other corners have similar results.

\[ \delta h_{y,\text{non-orthogonal}} \propto \begin{bmatrix} x_0 - x \\ x_0 \\ y_0 \end{bmatrix} \]  
\[ \delta h_{x,\text{non-orthogonal}} \propto \begin{bmatrix} y_0 - y \\ x_0 \\ y_0 \end{bmatrix} \]
In the case of joints that do impose orthogonality, the general form of the contour map remains that given in equation (11). This can be solved for the origin’s unit displacement.

\[ \delta h_{xy, \text{non-orthogonal}} \propto \begin{bmatrix} \frac{x}{x_0} & \frac{y}{y_0} \end{bmatrix} \] (17)

The first term is the shape of the floor sagging under its own weight; the second term is the contribution of the origin’s unit displacement. It follows that the component of the contour map generated by axial strain in a vertical beam of the house’s skeleton under the assumption that joints impose orthogonality between beams is the second term.

\[ \delta h_{00, \text{orthogonal}} = -\frac{\sigma_0}{24EI} \left( \left( x^2 - xx_0 \right)^2 + \left( y^2 - yy_0 \right)^2 \right) + \left[ 1 - \frac{3x^2}{x_0^2} - \frac{3y^2}{y_0^2} + \frac{2x^3}{x_0^3} + \frac{2y^3}{y_0^3} - \frac{xy}{x_0 y_0} \left( \frac{x_o - 2x}{x_0} \frac{x_0 - x}{x_0} + \frac{y_0 - 2y}{y_0} \frac{y_0 - y}{y_0} - 1 \right) \right] \] (18)

This solution is presented graphically in Figure 5.
FIGURE 5: Graphical representation of equation (19): a non-sagging floor supported by uneven joints that impose orthogonality. The axes are normalized as described in the caption to Figure 2.

The other corners have similar results.

\[
\delta h_{by, \text{orthogonal}} = \left[ 1 - \frac{3x^2}{x_0^2} - \frac{3(y_0 - y)^2}{y_0^2} + \frac{2x^3}{x_0^3} + \frac{2(y_0 - y)^3}{y_0^3} \right] \frac{x(y_0 - y)}{x_0y_0} \left( \frac{x_0 - 2x}{x_0} \frac{x_0 - x}{x_0} + \frac{y - y_0}{y_0} \frac{y - y_0}{y_0} - 1 \right)
\]

\[
\delta h_{x0, \text{orthogonal}} = \left[ 1 - \frac{3(x_0 - x)^2}{x_0^2} - \frac{3y^2}{y_0^2} + \frac{2(x_0 - x)^3}{x_0^3} + \frac{2y^3}{y_0^3} \right] \frac{(x_0 - x)y}{x_0y_0} \left( \frac{x - x_0}{x_0} \frac{x - x_0}{x_0} + \frac{x_0 - 2y}{x_0} \frac{x_0 - y_0}{y_0} - 1 \right)
\]

\[
\delta h_{xy, \text{orthogonal}} = \left[ 1 - \frac{3(x_0 - x)^2}{x_0^2} - \frac{3(y_0 - y)^2}{y_0^2} + \frac{2(x_0 - x)^3}{x_0^3} + \frac{2(y_0 - y)^3}{y_0^3} \right] \frac{(x_0 - x)(y_0 - y)}{x_0y_0} \left( \frac{x - x_0}{x_0} \frac{x - x_0}{x_0} + \frac{x_0 - y}{x_0} \frac{x_0 - y}{x_0} + \frac{y - y_0}{y_0} \frac{y - y_0}{y_0} - 1 \right)
\]
The corresponding discrete contour map and the discrete contour map corresponding to an uneven foundation are not orthonormal; there is no geometric distinction between the side of a room getting lower because the vertical beams of a the skeleton supporting it are getting shorter and the side of the foundation supporting that side of the room sinking into insufficiently supportive earth. However, because either situation implies an unsafe structure, it is reasonable to treat any such contour as a skeletal defect.

Because an observed contour map must be decomposed into these components, each must be orthonormalized with respect to a uniform offset of $\delta h$. After this orthonormalization, the continuous contour map of a floor predicted by this model is
\[ \delta h(x,y) = \]

\[ A_1 \left[ \frac{x_0 - x}{x_0} \frac{y_0 - y}{y_0} - \frac{1}{4} \right] + A_2 \left[ \frac{x}{x_0} \frac{y}{y_0} - \frac{1}{4} \right] + A_3 \left[ \frac{x}{x_0} \frac{y_0 - y}{y_0} - \frac{1}{4} \right] + A_4 \left[ \frac{x_0 - x}{x_0} \frac{y}{y_0} - \frac{1}{4} \right] + \]

\[ A_5 \left[ \frac{3}{4} \frac{x_0 - x}{x_0} \frac{3y^2 - 3y}{y_0} + \frac{2x^2}{x_0} + \frac{2y^3}{y_0} - \right] \]

\[ \frac{xy}{x_0y_0} \left( \frac{x_0 - 2x}{x_0} \frac{y_0 - y}{y_0} + \frac{y_0 - 2y}{y_0} \frac{y_0 - y}{y_0} - 1 \right) \]

\[ A_6 \left[ \frac{3}{4} \frac{3(x_0 - x)^2}{x_0^2} - \frac{3(y_0 - y)^2}{y_0^2} + \frac{2(x_0 - x)^3}{x_0^3} + \frac{2(y_0 - y)^3}{y_0^3} - \right] \]

\[ \frac{(x_0 - x)(y_0 - y)}{x_0y_0} \left( \frac{x - x_0}{x_0} \frac{x}{x_0} + \frac{y - y_0}{y_0} \frac{y}{y_0} - 1 \right) \]

\[ A_7 \left[ \frac{3}{4} \frac{3(x_0 - x)^2}{x_0^2} - \frac{3y^2}{y_0^2} + \frac{2(x_0 - x)^3}{x_0^3} + \frac{2y^3}{y_0^3} - \right] \]

\[ \frac{(x_0 - x) y}{x_0y_0} \left( \frac{x - x_0}{x_0} \frac{x}{x_0} + \frac{y_0 - 2y}{y_0} \frac{y_0 - y}{y_0} - 1 \right) \]

\[ A_8 \left[ \frac{3}{4} \frac{3x^2}{x_0^2} - \frac{3(y_0 - y)^2}{y_0^2} + \frac{2x^3}{x_0^3} + \frac{2(y_0 - y)^3}{y_0^3} - \right] \]

\[ \frac{x(y_0 - y)}{x_0y_0} \left( \frac{x_0 - 2x}{x_0} \frac{x}{x_0} + \frac{y - y_0}{y_0} \frac{y}{y_0} - 1 \right) \]

\[ A_9 \left[ \frac{x_0^5 y_0 + x_0 y_0^5}{5} - x^4 + 2x_0x^3 - x_0^3 x - y^4 + 2y_0y^3 - y_0^3 y \right] + \]

\[ A_{10} \left[ \frac{x_0^5 y_0 + x_0 y_0^5}{30} - \left( x^2 - xx_0 \right)^2 - \left( y^2 - y y_0 \right)^2 \right] + A_{11} \]

where \( A_1, A_2, A_3, \ldots, A_{11} \), are adjustable constants used to fit the measured shape of the floor and \( x_0 \) and \( y_0 \) are the dimensions of the floor. A possible value of this solution is presented graphically in figure 6.
FIGURE 6: Graphical representation of equation (23): an arbitrary floor for randomly generated values of $A_1=0.10$, $A_2=0.31$, $A_3=0.42$, $A_4=0.46$, $A_5=0.67$, $A_6=0.17$, $A_7=0.37$, $A_8=0.97$, $A_9=0.42$, $A_{10}=0.02$, $A_{11}=2.65$

Note that all coefficients except $A_{11}$ are assumed positive. Note also that none of the failure modes can be expressed as a weighted sum of positive multiples of any subset of the other ten.

2.2 Gradient and Contour Maps

Subsection 2.1 defines the contour map predicted by a theoretical model of the structure of a typical house. This subsection describes how a contour map can be obtained for a particular house. Direct measurement is nearly impossible because it would require measuring absolute height to the tenth of a millimeter. Such precision vastly exceeds that of any altimeter. Thus, a
gradient map measured with an inclinometer is used instead to compute the contour map.

The continuous gradient map, \( \nabla h(x, y) \), denotes the slope of the floor. It must be approximated with a discrete gradient map: \( \nabla \delta h(x, y) \in P_{\text{gradient}} \). The author’s experience directly measuring gradient maps indicate that the accuracy of the discrete gradient map does not substantially improve for more than four samples per square meter.

Once the set of sampling locations of the discrete gradient map has been determined, measuring the gradient at each location is simple. An acceptable precision is one to two milliradians, common for low-cost inclinometers, so the gradient can be easily measured on any clear floor.

The continuous gradient map is – as its name implies – the gradient of the continuous contour map. The continuous contour map, \( \delta h(x, y) \), denotes the difference between the height of a particular point on the floor from a constant reference height, where the reference height is a different constant for each room. The predicted contour components are orthonormal, so the reference height may be determined arbitrarily. In practice, only a finite number of measurements exist, so the continuous contour map is approximated with a discrete contour map, \( \delta h[x, y] \in P_{\text{contour}} \). Because the discrete gradient map’s accuracy does not substantially improve for more than four samples per square meter and the discrete contour map has twice the density of locations per area as the discrete gradient map (see discussion below), the accuracy of the discrete contour map does not substantially improve for more than eight samples per square meter. It is useful to note that a discrete contour map with \( n \) locations is also an \( n \)-dimensional continuous vector space.

The discrete contour map is the minimum squared error solution to an overdetermined system of linear equations. This system of equations comes from the discrete gradient map. For
each location in the discrete gradient map, there are three equations.

\[
\delta h\left((x,y) + \frac{u_1}{2}\right) - \delta h\left((x,y) - \frac{u_1}{2}\right) = \nabla \delta h \cdot \vec{u}_1
\]  

(24)

\[
\delta h\left((x,y) + \frac{u_2}{2}\right) - \delta h\left((x,y) - \frac{u_2}{2}\right) = \nabla \delta h \cdot \vec{u}_2
\]  

(25)

\[
\delta h\left((x,y) + \frac{u_1}{2}\right) + \delta h\left((x,y) - \frac{u_1}{2}\right) = \delta h\left((x,y) + \frac{u_2}{2}\right) + \delta h\left((x,y) - \frac{u_2}{2}\right)
\]  

(26)

Here, \( \vec{u}_1 \) and \( \vec{u}_2 \) are the discrete gradient map's basis vectors. In addition, there is one arbitrary location \( \vec{\rho} \) in the discrete contour map for which \( \delta h(\vec{\rho}) = 0 \). Methods for computing the minimum squared error solution to an overdetermined system of linear equations are readily available from other sources and not included here\(^2\).

The selections of sampling locations of the discrete gradient map and the discrete contour map are neither arbitrary nor independent. For each map, the difference between any two locations is the sum of integer multiples of two basis vectors. The basis vectors of the gradient map are always orthogonal. The discrete contour map's basis vectors are half the sum and half the difference of those of the discrete gradient map. The sum of any location in the discrete gradient map and half of either of the discrete contour map's basis vectors is a location in the discrete contour map. Notice that this implies that the contour map has exactly twice the density of sampling locations per unit area as the gradient map. The relationship between the locations in the gradient map and the contour map is shown graphically in Figure 7.
FIGURE 7: Graphical representation of relative positions of a discrete gradient map and a discrete contour map

2.3 Manual Analysis

In Spring 2010, the author measured the slope of a house in Worcester, MA, constructed in 1914. The only tool used to measure the discrete gradient map was an inclinometer with 0.1-degree precision. Measurements were taken at eighteen-inch to twenty-four-inch (0.457 to 0.610 meter) increments in most rooms. Because the ideal measurement density was not known a priori, one area’s measurements were taken at nine-inch (0.229) increments.

The analysis described above, when applied to the measured discrete gradient map, indicated that a -0.2% strain in a crucial vertical beam of the skeleton was the most substantial strain. Because the wood used to build this beam can strain roughly -0.5% before yielding, this suggests that, barring intervention, this house will collapse under its own weight around 2154. The contour map implying this conclusion is shown in figure 8.
2.4 Validation of model

Once a discrete contour map with \( n \) locations has been obtained, it can be considered an \( n \)-dimensional vector for which each component is equal to the relative height of one location. The theoretical model of a house presented in subsection 2.1 predicts a contour map with eleven adjustable constants, so the predicted discrete contour map also has eleven adjustable constants. For any floor large enough to have a discrete contour map with over eleven locations (roughly a four foot square), the model cannot fit all possible contour maps. A discrete contour map may easily have over a hundred locations. An arbitrary hundred-dimensional vector cannot be decomposed into eleven arbitrary basis vectors.

*The inability of the model to explain every possible contour map is essential in validating the model.* If eleven basis vectors predict, with high accuracy, a point in an \( n \)-dimensional vector space, then those eleven are likely to be intelligently chosen. If \( n \) is high, any arbitrary eleven
basis vectors are very unlikely to predict the contour accurately, so the chosen eleven basis
vectors predicting accurately is strong evidence that the model providing them is valid.

This also implies that every analysis includes self-assessment. After the components of
the discrete contour map in the direction of each basis vector are removed, the remaining
component of the contour is unexplained by the theoretical model. However, if the theoretical
structural model is accurate, then this residual will be very small.

For the house in Worcester, MA', the discrete gradient map had an average of eighty-four
locations for each room, so the average residual of a zero-information model is \((84-11)/84\), or
roughly 86.9%. On average, the residual actually accounted for 21.3% (standard deviation
9.83%) of the discrete contour map. This confirms that the model is valid for this particular
house.

3. Automated Gradient Map Measurement

The repetitiveness of the measuring process suggests automation. A robot must be
capable of moving, tracking its own position and orientation, and measuring the slope of the
floor it rests on to automatically measure the discrete gradient map. The author constructed a
robot with these capabilities. This section provides an overview of its design. Subsection 3.0 lists
the essential mechanical details of construction and equipment. Subsection 3.1 summarizes the
software algorithms used to control the robot. Subsection 3.2 compares a few gradient
measurements made by a hand inclinometer to those made by the robot itself.

3.0 Hardware

Position control uses some basic form of vision. The type selected as best suited for this
application is a long-range IR distance-measuring sensor.

An inclinometer is used to measure the floor's slope. It is located between the two coaxial
driven wheels by necessity; if the robot rotates in place, the inclinometer continues measuring the same location. The inclinometer was constructed by hanging a weight from a position encoder with 1600 counts/revolution, so it measures in increments of 0.225 degrees. As mentioned above, the hand inclinometer had a resolution of 0.1 degrees.

FIGURE 9: Photograph of constructed robot.

FIGURE 10: Close-up of the robot’s inclinometer and the hand inclinometer
3.1 Control algorithms

Because high precision control of position is essential, it is useful to consider two feedback systems: one to control position and another to control rotation.

Because the robot measures a gradient map whose samples lie on a grid orthogonal to the walls, iteratively driving until the perceived wall is at the desired distance and rotating ninety degrees reaches each successive position. This prevents accumulation of dead reckoning errors.

Rotation control requires seeking an orientation in the direction of the discrete gradient map's basis vectors. These positions directly face walls. Rotating to local minimum visual ranges reliably counts revolutions but lacks precision. However, assuming any gradient exists, the measured slope is proportional to the arcsine of the deviation of the robot's angle from the floor's gradient. To make a precise rotation, the robot seeks a computed slope. This requires that the robot's inclinometer be located directly above the center of the axle of its drive wheels.

With precise driving, the robot simply measures the dimensions of the room and measures each location of the gradient map in the simple pattern shown in Figure 11.
FIGURE 11: Graphical representation of the robot's path measuring a floor. It stops at every intersection of two lines to perform the two slope measurements required to record a gradient.

The high level implementation of common robotic mapmaking was beyond the scope of this thesis project. However, the core algorithms specific to this robot supporting all higher-level functions were developed and are included in the appendix.

3.2 Testing

To test the robot's measurement capabilities, it was placed on a variable-slope incline with a hand inclinometer. In this configuration, both measured exactly the same angle. The incline was moved to several positions and both measurements were compared. Since the hand inclinometer is known to be reliable, this tests the precision of the robot's inclinometer. The results of this testing are plotted in Figure 12.

FIGURE 12: Comparison of inclinometer and robot gradient measurements. The $y=x$ line and the best fit have been plotted for reference.
The robot's measurements are imprecise. Its root-mean-square is 0.4 degrees, only sufficient to observe substantial damage. With 95% confidence, the slope of the best fit to the robot's measurements is $1.0334 \pm 0.193$, statistically indistinguishable from one. A position encoder with finer precision and less viscous damping would be beneficial.

4. Conclusions

A model was developed predicting failures in a house's structure. This model allows damage to be assessed based on the contour map computed from the measured gradient map of the house's floors. The gradient map is measured at specific locations with a density of approximately four measurements per square meter.

This model was validated with a comparison to previous floor gradient measurements of a house built in 1914. The model predicted roughly 78.7% of the measurements; the remainder corresponded roughly to particularly heavy objects and random noise.

A prototype robot was constructed to demonstrate the automatability of measuring a floor's gradient. Although high-level implementation of common robotic mapmaking was outside the scope of this project, the on-board inclinometer was tested and found to be accurate to within a RMS error of 0.4%.

In the future, the prototype could be developed into a simple product and mass-manufactured, enabling homeowners to easily and affordably assess the safety of their homes.

5. References

1 Mechanics of Materials 7th edition, by R. C Hibbeler
4 Most of the robot was provided by MIT's Microcomputer Project Laboratory course: 6.115
Appendix: software

This section includes all programmed code used in this project.

; Printing subroutines provided by 6.115's MINMON

stack equ 2fh ; bottom of stack
    ; - stack starts at 30h -
errorf equ 0 ; bit 0 is error status

; 8032 hardware vectors
org 00h ; power up and reset vector
ljmp start
org 03h ; interrupt 0 vector
ljmp start
org 0bh ; timer 0 interrupt vector
ljmp start
org 13h ; interrupt 1 vector
ljmp start
org 1bh ; timer 1 interrupt vector
ljmp start
org 23h ; serial port interrupt vector
ljmp start
org 2bh ; 8052 extra interrupt vector
ljmp start

; begin main program
org 100h
start:
    lcall init

doneLoop:
    mov a, #5
    lcall driveForwardDistance
    lcall driveUnitTest
    mov R5, #10h
    lcall delay
    sjmp doneLoop

; UNIT TESTS START HERE

; subroutine frontEyeUnitTest
; prints the distance perceived by the front eye, in cm
frontEyeUnitTest:
    lcall frontEye
    lcall prthex
    ret
subroutine backEyeUnitTest
prints the distance perceived by the back eye, in cm

backEyeUnitTest:
   lcall backEye
   lcall prthex
   ret

subroutine driveUnitTest
does the hokey-pokey

driveUnitTest:
   mov R5, #1
   mov R6, #0
   mov R7, #0
   lcall driveForwardTime ; Step forward
   mov R5, #1
   mov R6, #0
   mov R7, #0
   lcall driveBackwardTime ; Step back
   mov R0, #4
   driveUnitTestLoop: ; Shake
      mov R5, #1
      mov R6, #0
      mov R7, #0
      lcall driveLeftTime
      mov R5, #1
      mov R6, #0
      mov R7, #0
      lcall driveRightTime
      djnz r0, driveUnitTestLoop
   mov a, #10
   lcall driveBackwardDistance
   mov a, #10
   lcall driveForwardDistance
   ret

UNIT TESTS END HERE

subroutine init
this routine initializes the hardware
set up serial port with a 11.0592 MHz crystal,
use timer 1 for 9600 baud serial communications

init:

; pg 2-14
; set timers 0 and 1 for auto reload - mode 2
    mov tm0d, #00100010b
; 1st bit: 0, so timer/counter 1 is run by software
; 2nd bit: 0, so timer/counter 1 is a timer
; 3rd and 4th bits: 10, so timer 1 is in mode 2
; 5th bit: 0, so timer/counter 0 is run by software
; 6th bit: 0, so timer/counter 0 is a timer
; 7th to 8th bits: 10, so timer 0 is in mode 2

; pg 2-14
; run timer/counter 0 and 1
    mov tcon, #01000000b  ; Turn timer/counter 1 on
; 1st bit: 0, handled by hardware
; 2nd bit: 1, so timer/counter 1 is on
; 3rd bit: 0, handled by hardware
; 4th bit: 0, so timer/counter 0 is off
; 5th bit: 0, handled by hardware
; 6th bit: 0, TA said so, "no interrupts"
; 7th bit: 0, handled by hardware
; 8th bit: 0, so interrupt 0 triggered by edge

; pg 2-20
; set 9600 baud with xtal=11.059mhz
; th1 = 256 - (11059000/(12*32*9600)) = 253
    mov th1, #11111101b

; pg 2-19
; set serial control reg for 8 bit data and mode 1
    mov scon, #01010000b
; 1st and 2nd bits: 01, so mode 1 is used
; 3rd bit: 0, because there is only one processor
; 4th bit: 1, to enable reception
; 5th bit: 0, will be overwritten anyway
; 6th bit: 0, will be overwritten anyway
; 7th bit: 0, handled by hardware
; 8th bit: 0, handled by hardware

lcall stopMoving
ret

;=================================================================================================
; subroutine sndchr
; this routine takes the chr in the acc and sends it out the serial port.
;=================================================================================================

    subroutine sndchr
    ; clear the tx buffer full flag.
    clr scon.1
    ; put chr in sbuf
    mov sbuf,a
txloop:
    jnb scon.1, txloop  ; wait till chr is sent
    ret

;=================================================================================================
; subroutine getchr

; this routine reads in a chr from the serial port and saves it in the accumulator.

getchr:
    jnb ri, getchr ; wait till character received
    mov a, sbuf ; get character
    anl a, #7fh ; mask off 8th bit
    clr ri ; clear serial status bit
    ret

; subroutine print
; print takes the string immediately following the call and sends it out the serial port. the string must be terminated with a null. this routine will ret to the instruction immediately following the string.

print:
    pop dph ; put return address in dptr
    pop dpl

prtstr:
    clr a ; set offset = 0
    movc a, @a+dptr ; get chr from code memory
    cjne a, #0h, mchrok ; if termination chr, then return
    jmp prtdone

mchrok:
    lcall sndchr ; send character
    inc dptr ; point at next character
    sjmp prtstr ; loop till end of string

prtdone:
    mov a, #1h ; point to instruction after string
    jmp @a+dptr ; return

; subroutine crlf
; crlf sends a carriage return line feed out the serial port

crlf:
    mov a, #0ah ; print lf
    lcall sndchr

crlf:
    mov a, #0dh ; print cr
    lcall sndchr

ret

; subroutine prthex
; this routine takes the contents of the acc and prints it out as a 2 digit ascii hex number.

prthex:
    push acc
    lcall binasc ; convert acc to ascii
    lcall sndchr ; print first ascii hex digit
    mov a, r2 ; get second ascii hex digit
    lcall sndchr ; print it
pop acc
ret

; subroutine binasc
; binasc takes the contents of the accumulator and converts it into two ascii hex numbers. the result is returned in the accumulator and r2.

binasc:
    mov r2, a ; save in r2
    anl a, #0fh ; convert least sig digit.
    add a, #0f6h ; adjust it
    jnc noadj1 ; if a-f then readjust
    add a, #07h

noadj1:
    add a, #3ah ; make ascii
    xch a, r2 ; put result in reg 2
    swap a ; convert most sig digit
    anl a, #0fh ; look at least sig half of acc
    add a, #0f6h ; adjust it
    jnc noadj2 ; if a-f then re-adjust
    add a, #07h

noadj2:
    add a, #3ah ; make ascii
    ret

; subroutine ascbin
; this routine takes the ascii character passed to it in the acc and converts it to a 4 bit binary number which is returned in the acc.

ascbin:
    clr errorf
    add a, #0d0h ; if chr < 30 then error
    jnc notnum
    clr c ; check if chr is 0-9
    add a, #0f6h ; adjust it
    jc hextry ; jmp if chr not 0-9
    add a, #0ah ; if it is then adjust it
    ret

hextry:
    clr acc.5 ; convert to upper
    clr c ; check if chr is a-f
    add a, #0f9h ; adjust it
    jnc notnum ; if not a-f then error
    clr c ; see if char is 46 or less.
    add a, #0fah ; adjust acc
    jc notnum ; if carry then not hex
    anl a, #0fh ; clear unused bits
    ret

notnum:
    setb errorf ; if not a valid digit
ljmp start

; subroutine delay
; wait for time in R5 R6 R7
; leaves all 0's in R5, R6, and R7

delay:
    inc R5
    inc R6
    inc R7
delayLoop:
    djnz R7, delayLoop
    djnz R6, delayLoop
    djnz R5, delayLoop
ret

; ROBOT ROUTINES START HERE

; P3.2 - left wheel back
; P3.3 - left wheel forward
; P3.4 - right wheel forward
; P3.5 - right wheel back
; FE00 - back sensor
; FE10 - front sensor
; FE20 - position encoder

; subroutine frontEye
; computes the distance from the front IR sensor to the wall
; returns the distance, in cm, in the accumulator

frontEye:
    mov dptr, #0fe10h ; address of front eye
    lcall getEye
    lcall eye2distance
ret

; subroutine backEye
; computes the distance from the back IR sensor to the wall
; returns the distance, in cm, in the accumulator

backEye:
    mov dptr, #0fe00h ; address of back eye
    lcall getEye
    lcall eye2distance
ret
subroutine eye2distance
; takes as input an eye voltage in the accumulator
; computes the distance, in cm
; the result is returned in the accumulator

; eye2distance:
  inc a
  movc a, @a+pc
  ret

; destroys R0, R1, R5, R6, R7

subroutine getEye
; reads the voltage of the specified eye four times, taking the
; average voltage
; the result is returned in the accumulator

getEye:
  lcall getRawEyeVoltage
  mov R0, acc
lcall getRawEyeVoltage
  addc a, R0
  rrc a
  mov R0, acc
  clr c
lcall getRawEyeVoltage
  mov R1, acc
lcall getRawEyeVoltage
  addc a, R1
  rrc a
  clr c
  addc a, R0
  rrc a
ret

;===============================================================================
; destroys R5, R6, R7
; subroutine getRawEyeVoltage
; reads the voltage of the specified eye
; the result is returned in the accumulator
;===============================================================================
getRawEyeVoltage:
  movx @dptr, a ; poke it
  mov R5, #0h
  mov R6, #0h
  mov R7, #20h
  lcall delay ; wait for it
  movx a, @dptr ; read it
ret

;===============================================================================
; destroys R0, R1, R2, R3, R4, and R5
; subroutine setLeftWheelDutyCycle
; takes as input 256*(duty cycle)-1 in the accumulator
; sets the left wheel's duty cycle to the input duty cycle
;===============================================================================
setLeftWheelDutyCycle:
  lcall setWheelDutyCycle
  lcall stepSizeFromDutyCycle
  mov R2, a ; record step size
  mov R3, #2 ; record wheel selection, "ON"
  mov R4, #0FEh ; record wheel selection, "OFF"
  lcall setWheelDutyCycle
ret

;===============================================================================
; destroys R0, R1, R2, R3, R4, and R5
; subroutine setRightWheelDutyCycle
; takes as input 256*(duty cycle)-1 in the accumulator
; sets the right wheel's duty cycle to the input duty cycle
;===============================================================================
setRightWheelDutyCycle:
; subroutine setWheelsDutyCycle
; takes as input 256*(duty cycle)-1 in the accumulator
; sets both wheels' duty cycles to the input duty cycle
;base
setWheelsDutyCycle:
    lcall setWheelDutyCycle
    lcall stepSizeFromDutyCycle
    mov R2, a  ; record step size
    mov R3, #1 ; record wheel selection, "ON"
    mov R4, #OFFh ; record wheel selection, "OFF"
    lcall setWheelDutyCycle
    ret

; subroutine setWheelDutyCycle
;base
setWheelDutyCycle:
    mov RO, #0 ; speed control table iterator
    mov R1, #0 ; speed control pre-table iterator
    setWheelDutyCycleLoop:
        ; Get pre-table value
        mov dph, #21h ; read from the pre-table
        ; determine where to look in pre-table
        mov R1, a
        add a, R2
        mov R1, a
        mov dpl, R1 ; select pre-table value
        movx a, @dptr ; read pre-table value
        anl a, R3 ; only apply to wheel being controlled
        mov R5, a ; save new data in R5

        ; Update table value
        mov dph, #20h ; read from the table
        mov dpl, R0 ; select table value
        movx a, @dptr ; read table value
        anl a, R4 ; remove old data
        orl a, R5 ; update bit (wheel) being modified, but
                  ; don't change any other data
        movx @dptr, a ; save new value
    djnz RO, setWheelDutyCycleLoop ; Next value
    ret
subroutine setWheelDutyCycle
; takes as input 256*(duty cycle)-1 in the accumulator
; fills the speed control pretable (0A100h-0A1FFh) with that
; many highs
; leaves accumulator unchanged

setWheelDutyCycle:
    push acc
    inc acc
    mov R0, a
    mov a, #0FFh
    mov dph, #21h
    setWheelDutyCycleLoop:
        mov dpl, R0
        movx @dptr, a
        djnz R0, setWheelDutyCycleLoop
    pop acc
    ret

subroutine stepSizeFromDutyCycle
; takes as input 256*(duty cycle)-1 in the accumulator
; computes step size of 1's for writing the speed control table
; returns the result in the accumulator

stepSizeFromDutyCycle:
    inc a
    movc a, @a+pc
    ret

    db 001h,003h,003h,005h,005h,007h,007h,009h
    db 009h,008h,008h,000h,000h,000h,000h,000h
    db 011h,013h,013h,015h,015h,017h,017h,019h
    db 019h,018h,018h,010h,010h,010h,010h,010h
    db 021h,023h,023h,025h,025h,027h,027h,029h
    db 029h,028h,028h,020h,020h,020h,020h,020h
    db 031h,033h,033h,035h,035h,037h,037h,039h
    db 039h,038h,038h,030h,030h,030h,030h,030h
    db 041h,043h,043h,045h,045h,047h,047h,049h
    db 049h,048h,048h,040h,040h,040h,040h,040h
    db 051h,053h,053h,055h,055h,057h,057h,059h
    db 059h,058h,058h,050h,050h,050h,050h,050h
    db 061h,063h,063h,065h,065h,067h,067h,069h
    db 069h,068h,068h,060h,060h,060h,060h,060h
    db 071h,073h,073h,075h,075h,077h,077h,079h
    db 079h,078h,078h,070h,070h,070h,070h,070h
    db 07Fh,07Fh,07Dh,07Dh,07Bh,07Bh,079h,079h
    db 077h,077h,075h,075h,073h,073h,071h,071h
    db 06Fh,06Fh,06Dh,06Dh,06Bh,068h,068h,069h
    db 067h,067h,065h,065h,063h,063h,061h,061h
    db 05Fh,05Fh,05Dh,050h,050h,058h,058h,059h
    db 057h,057h,055h,055h,053h,053h,051h,051h
stopMoving:
   clr P3.2
   clr P3.3
   clr P3.4
   clr P3.5
ret

;============================================================================
;        destroys R0, R1, R2, R3, R5, R6, and R7
; subroutine driveForwardDistance
; drive forward the distance in the accumulator, in cm
; if sensors do not indicate this is possible, do nothing
; a successful drive leaves 0 in the accumulator
; a failed drive leaves 1 in the accumulator
;============================================================================
driveForwardDistance:
lcall stopMoving

   mov R2, a            ; save the distance to travel
   lcall frontEye       ; get distance to wall ahead
   mov R3, a            ; save distance to wall ahead

   ; make sure there's room to drive
   clr c
   subb a, R2
   jc failedToDriveForward
   subb a, #20
   jc failedToDriveForward
   addc a, #20          ; compute the desired sensor distance

   ; make sure the robot isn't up against the wall
   setb P3.3            ; move forward
   setb P3.4
   mov R5,#1            ; for a brief time
   mov R6,#0
   mov R7,#0
   lcall delay
   clr P3.3             ; stop
   clr P3.4
   lcall frontEye       ; get distance to wall ahead
   ; it should be less
subb a, R3
jnc failedToDriveForward ; if not, the robot is less than
; eight centimeters from crashing

; compute the desired position
mov a, R3 ; where you are
subb a, R2 ; minus distance to go
inc a ; add 1, so stop when distance to wall
; is less than this
mov R3, a ; save the desired position

clr c

; start driving forward
setb P3.3
setb P3.4

driveForwardDistanceLoop:
lcall frontEye ; see where you are
subb a, R3 ; subtract where you want to be
jnc driveForwardDistanceLoop ; stop when you get there

; stop
clr P3.3
clr P3.4

; signal success
mov a, #0
ret

failedToDriveForward:
mov a, #1
lcall stopMoving
ret

;==========================================================================
; destroys R0, R1, R2, R3, R5, R6, and R7
; subroutine driveBackwardDistance
; drive backward the distance in the accumulator, in cm
; if sensors do not indicate this is possible, do nothing
; a successful drive leaves 0 in the accumulator
; a failed drive leaves 1 in the accumulator
;==========================================================================
driveBackwardDistance:
lcall stopMoving

mov R2, a ; save the distance to travel
lcall backEye ; get distance to wall behind
mov R3, a ; save distance to wall behind

; make sure there's room to drive
clr c
subb a, R2
jc failedToDrive
subb a, #20
jc failedToDrive

addc a, #20 ; compute the desired sensor distance

; make sure the robot isn't up against the wall
setb P3.2 ; move backward
setb P3.5
mov R5,#1 ; for a brief time
mov R6,#0
mov R7,#0
lcall delay
clr P3.3 ; stop
clr P3.4
lcall backEye ; get distance to wall behind
; it should be less
subb a, R3
jnc failedToDrive ; if not, the robot is less than
; eight centimeters from crashing

; compute the desired position
mov a, R3 ; where you are
subb a, R2 ; minus distance to go
inc a ; add 1, so stop when distance to wall
; is less than this
mov R3, a ; save the desired position

clr c

; start driving backward
setb P3.2
setb P3.5

driveBackwardDistanceLoop:
lcall frontEye ; see where you are
subb a, R3 ; subtract where you want to be
jnc driveBackwardDistanceLoop ; stop when you get there

; stop
clr P3.2
clr P3.5

; signal success
mov a, #0
ret

failedToDrive:
    mov a, #1
    lcall stopMoving
ret

; ==========================================================================
; destroys R5, R6, and R7
; subroutine driveForwardTime
; drives both wheels forward for duration R5 R6 R7
;============================================================================
driveForwardTime:
    mov dph, #20h ; point to the speed control table
    inc R5
    inc R6
    inc R7
driveForwardTimeLoop:
    inc dpl ; point to the next drive instruction
    movx a, @dptr ; get the next drive instruction
    ; Replacement of what should be "mov P3.3, acc.1"
    jb acc.1, driveForwardTimeLoop1
    clr P3.3
    sjmp driveForwardTimeLoop2
driveForwardTimeLoop1:
    setb P3.3
driveForwardTimeLoop2:
    ; Replacement of what should be "mov P3.4, acc.0"
    jb acc.0, driveForwardTimeLoop3
    clr P3.4
    sjmp driveForwardTimeLoop4
driveForwardTimeLoop3:
    setb P3.4
driveForwardTimeLoop4:
    djnz R7, driveForwardTimeLoop
djnz R6, driveForwardTimeLoop
djnz R5, driveForwardTimeLoop
lcall stopMoving
ret
;============================================================================
;   destroys R5, R6, and R7
; subroutine driveBackwardTime
; drives both wheels backward for duration R5 R6 R7
;============================================================================
driveBackwardTime:
    mov dph, #20h ; point to the speed control table
    inc R5
    inc R6
    inc R7
driveBackwardTimeLoop:
    inc dpl ; point to the next drive instruction
    movx a, @dptr ; get the next drive instruction
    ; Replacement of what should be "mov P3.2, acc.1"
    jb acc.1, driveBackwardTimeLoop1
    clr P3.2
    sjmp driveBackwardTimeLoop2
driveBackwardTimeLoop1:
    setb P3.2
driveBackwardTimeLoop2:

; Replacement of what should be "mov P3.5, acc.0"
   jb acc.0, driveBackwardTimeLoop3
   clr P3.5
   sjmp driveBackwardTimeLoop4
driveBackwardTimeLoop3:
   setb P3.5
driveBackwardTimeLoop4:

djnz R7, driveBackwardTimeLoop
djnz R6, driveBackwardTimeLoop
djnz R5, driveBackwardTimeLoop
lcall stopMoving
ret

;==============================================================================
;    destroys R5, R6, and R7
; subroutine driveLeftTime
; drives left wheel backward, right wheel forward
; for duration R5 R6 R7
;==============================================================================
driveLeftTime:
   mov dph, #20h ; point to the speed control table
   inc R5
   inc R6
   inc R7
driveLeftTimeLoop:
   inc dpl ; point to the next drive instruction
   movx a, @dptr ; get the next drive instruction

   ; Replacement of what should be "mov P3.2, acc.1"
   jb acc.1, driveLeftTimeLoop1
   clr P3.2
   sjmp driveLeftTimeLoop2
driveLeftTimeLoop1:
   setb P3.2
driveLeftTimeLoop2:

   ; Replacement of what should be "mov P3.4, acc.0"
   jb acc.0, driveLeftTimeLoop3
   clr P3.4
   sjmp driveLeftTimeLoop4
driveLeftTimeLoop3:
   setb P3.4
driveLeftTimeLoop4:

djnz R7, driveLeftTimeLoop
djnz R6, driveLeftTimeLoop
djnz R5, driveLeftTimeLoop
lcall stopMoving
ret
; destroys R5, R6, and R7
; subroutine driveRightTime
; drives left wheel forward, right wheel backward
; for duration R5 R6 R7
;
; --------------------
driveRightTime:
    mov dph, #20h ; point to the speed control table
    inc R5
    inc R6
    inc R7

driveRightTimeLoop:
    inc dpl ; point to the next drive instruction
    movx a, @dptr ; get the next drive instruction

; Replacement of what should be "mov P3.3, acc.1"
    jb acc.1, driveRightTimeLoop1
    clr P3.3
    sjmp driveRightTimeLoop2

driveRightTimeLoop1:
    setb P3.3

driveRightTimeLoop2:

; Replacement of what should be "mov P3.5, acc.0"
    jb acc.0, driveRightTimeLoop3
    clr P3.5
    sjmp driveRightTimeLoop4

driveRightTimeLoop3:
    setb P3.5

driveRightTimeLoop4:

; djnz R7, driveRightTimeLoop
; djnz R6, driveRightTimeLoop
; djnz R5, driveRightTimeLoop
; lcall stopMoving
ret

; --------------------
angle:
; subroutine angle
; measures the angle of the encoder
; saves the value in the accumulator
;

angle:
    mov dptr, #0fe20h
    movx a, @dptr
ret

; ROBOT ROUTINES END HERE
;
; Speed control table
; The table at 02000h to 020FFh is used for finely-controlled
; pulse width modulation. The final two bits of each entry are
; set if and only if that wheel is turning for an instant

; While driving, dpl cycles through the table. At steps where a
; wheel's entry is zero, that wheel is not driven

; The 2's bit corresponds to the left wheel, the 1's bit
; corresponds to the right wheel

org 2000h
db 3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3
db 3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3
db 3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3
db 3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3
db 3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3
db 3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3
db 3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3
db 3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3

org 2100h
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
db 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
import math
import random

class floorMap:
    # Defines a rectangular grid corresponding to gradient samples
    # x and y are coordinates used in two separate coordinate systems: global and local
    # Global coordinate system:
    # This coordinate system is defined relative to the house
    # All floorMaps use the same global coordinate system
    # Recommended convention:
    # +x = east, -x = west,
    # +y = north, -y = south
    # x and y are both measured in inches
    # If this convention is changed, it must be changed for
    # All floorMaps
    # Local coordinate system:
    # Each floormap stores samples taken from a rectangular grid nx points by ny points.
    # x and y coordinates are integers

def __init__(self, global_x0, global_y0, dx, dy, nx, ny):
    # global _x0 and _y0 store the x and y location in the global
    # coordinate system corresponding to (0,0) in the local
    # coordinate system
    self.global_x0, self.global_y0 = global_x0, global_y0

    # dx and dy store the gap between successive local
    # coordinate system locations using global coordinate
    # system dimensions
    self.dx, self.dy = float(dx), float(dy)

    # nx and ny store the dimensions of the local coordinate
    # system. The floorMap stores measurements taken from
    # nx*ny locations on the floor
    self(nx, self.ny = nx, ny

    # equations stores all constraints on the floor's contour map
    # in the form (p1,p2,dh)
    # p1 and p2 are points (x1,y1) and (x2,y2) in the global
    # coordinate system
    # dh is the value of h2-h1 implied by the slope
    # Mandatory convention:
    # slope is measured in tenths of a degree
    self.equations = []

def localToGlobal(self, localPoint):
    (nx, ny) = localPoint

    # Linear coordinate transformation
    x = self.global_x0 + self.dx*nx
    y = self.global_y0 + self.dy*ny
globalPoint = (x, y)
return globalPoint

def globalToLocal(self, globalPoint):
    (x, y) = globalPoint

    # Linear coordinate transformation
    nx = (x - self.global_x0) / self.dx
    ny = (y - self.global_y0) / self.dy

    # Local coodinated must be in integers (locations are on a
    #     rectangular grid)
    # Round local coodinated to the nearest integer
    nx, ny = int(nx+.5), int(ny+.5)

    localPoint = (nx, ny)
    return localPoint

def recordReading(self, nx, ny, dzdx, dzdy):
    # Note: A dzdx or dzdy value outside ±90 degrees (± 900)
    #     indicates no data
    if abs(dzdx) < 900:
        # Record x-component of gradient
        p1 = self.localToGlobal( (nx-.5,ny) )
        p2 = self.localToGlobal( (nx+.5,ny) )
        slope = dzdx
        self.equations.append((p1,p2,slope*self.dx))
    if abs(dzdy) < 900:
        # Record y-component of gradient
        p1 = self.localToGlobal( (nx,ny-.5) )
        p2 = self.localToGlobal( (nx,ny+.5) )
        slope = dzdy
        self.equations.append((p1,p2,slope*self.dy))

class floorPlan:
    # A floorPlan is a collection of floorMaps. Each floorMap is a
    #     rectangle, and the floorPlan assembles these
    #     together. A floorPlan represents an entire story of a
    #     building. Each floorPlan has its own coordinatae
    #     system that each of it's floorMaps treat as the
    #     global coordinate system.
    def __init__(self):
        # locations stores the list of all locations whose heights
        #     are included in any equations
        #     locations are in the form (x,y), in inches
        self.locations = []

        # self.equations stores all constraints on the heights of
        #     floor locations
        # Each equation is a (terms, sum) tuple
        #     terms is a list of (locationIndex, coefficient) tuples
        #     location index is the index (in self.locations) of
the location whose height is used in this equation
coefficient is multiplied by said height
sum is the desired sum of all height\times coefficient products
The reason for this format is that the number of
locations whose height is used in this equation is very small compared with the number of locations

```python
self.equations = []
```

All slope equations indicate only relative floor heights.
To create a unique optimal solution, let the height of the first location be 0

```python
self.equations.append( [[(0,1.)],0. ] )
```

floorMaps is a list of all the floorMaps from which this floorPlan was composed

```python
self.floorMaps = []
```
ooms is a list of the physical rooms in the house
They are stored as (min_X, max_X, min_Y, max_Y) tuples

```python
self.rooms = []
```

def locationIndex(self, location):
    # Locations are stored as their (x,y) coordinates, in inches.
    # Two locations are treated as identical in they are within one inch of each other
    # Returns -1 if the location is in any equation
    (x,y) = location
    for i in range(len(self.locations)):
        loc = self.locations[i]
        if abs(x-loc[0]) < 1 and abs(y-loc[1]) < 1:
            return i
    return -1

def addPoint(self, p):
    # Adds a new location to the list (if it isn't already there)
    if self.locationIndex(p) == -1:
        self.locations.append(p)

def addMap(self, floorMap):
    # Record all the data from a new map
    for x in range(floorMap.nx):
        for y in range(floorMap.ny):
            # For each point, record the equation stating the height is locally linear
            p, i = [0]*4, [0]*4
            p[0] = floorMap.localToGlobal((x+.5,y))
            p[1] = floorMap.localToGlobal((x-.5,y))
            p[2] = floorMap.localToGlobal((x,y+.5))
            p[3] = floorMap.localToGlobal((x,y-.5))
            for n in range(4):
                self.addPoint(p[n])
i[n] = self.locationIndex(p[n])
eqn = (((i[0],1.), (i[1],1), (i[2],-1), (i[3],-1)), 0.)
self.equations.append(eqn)

borderPoints = []
for xn in [0, floorMap.nx - 1]:
    for yn in range(floorMap.ny):
        borderPoints.append((xn, yn))
for xn in range(1, floorMap.nx - 1):
    for yn in [0, floorMap.ny - 1]:
        borderPoints.append((xn, yn))
for (xn, yn) in borderPoints:
    # For each point on the edge of this floorMap, see if
    # it's near another floorMap and require it to
    # be continuous.
    p = floorMap.localToGlobal((xn, yn))
    for FM2 in self.floorMaps:
        (x2, y2) = FM2.globalToLocal(p)
        p2 = FM2.localToGlobal((x2, y2))
        if 1 <= x2 and x2 <= FM2.nx and 1 <= y2 and y2 <= FM2.ny:
            # p is the new point, p2 is the nearest point on
            # the previously existing grid
            i1 = self.locationIndex(p)
            i2 = self.locationIndex(p2)
            if i1 != i2:
                # Require that two point very near each other
                # be of similar height
                eqn = (((i1, 1.), (i2, -1.)), 0)
                self.equations.append(eqn)

for eqn in floorMap.equations:
    # Record all explicitly stated dzdx and dzdy equations
    (p1, p2, dh) = eqn
    i1 = self.locationIndex(p1)
    i2 = self.locationIndex(p2)
    C = (math.pi/180.)*.1
    eqn = (((i1, -1.), (i2, 1.)), dh*C)
    self.equations.append(eqn)
self.floorMaps.append(floorMap)
def equationsForMatlab(self):
    # The height equations can be written in the form Ax=b
    # A is an m-by-n matrix
    # m is the number of equations
    # n is the number of heights
    # x is a matrix of the heights
    # Returns (A, b)
    # Matrices here are represented as arrays in which each
    # element is a row
    m = len(self.equations)
    n = len(self.locations)
    A = []
    b = [0]*m
for i in range(m):
    A.append([0]*n)
for i in range(len(self.equations)):
    eqn = self.equations[i]
    terms, val = eqn
    b[i] = val
    for term in terms:
        j, coeff = term
        A[i][j] = coeff
return (A, b)

def addRoom(self, room):
    # Records a room location
    # Format: (x_min, x_max, y_min, y_max) tuple
    self.rooms.append(room)

def removeFloor(self, contourMap):
    # Identifies sagging floor shapes and removes them, leaving
    # only the effects of a damaged skeletal structure
    # contourMap is a list of (x,y,z) tuples
    def inRoom(loc, room):
        x, y, z = loc
        x_min, x_max, y_min, y_max = room
        return (x_min<x and x<x_max and y_min<y and y<y_max)
    def unitRoomHeight(loc, room):
        x, y, z = loc
        x_min, x_max, y_min, y_max = room
        X = float(x-x_min)/float(x_max-x_min)
        Y = float(y-y_min)/float(y_max-y_min)
        return (X**4-2*X**3+X)*(Y**4-2*Y**3+Y)
    contourMapUsed = []
    for loc in contourMap:
        # Only use data points in rooms
        ValidPoint = False
        for room in self.rooms:
            if inRoom(loc, room):
                ValidPoint = True
                if ValidPoint:
                    contourMapUsed.append(loc)
        mean = float(sum([z for x,y,z in contourMapUsed]))/len(contourMapUsed)
    contourMapUsed = [(x,y,z-mean) for x,y,z in contourMapUsed]
    for room in self.rooms:
        # Correlation is the n-space dot product of a unit floor
        # sag with a contour map
        # True: based on contour map
        # Perfect: based on copy of unit floor sag
        True_correlation = 0
        Perfect_correlation = 0
        for loc in contourMapUsed:
            if inRoom(loc, room):
                h = unitRoomHeight(loc, room)
True_correlation += h*loc[2]
Perfect_correlation += h*h
sag_factor = True_correlation/Perfect_correlation
print sag_factor
for i in range(len(contourMapUsed)):
    loc = contourMapUsed[i]
    if inRoom(loc, room):
        h = unitRoomHeight(loc, room)
        x, y, z = loc
        z -= h*sag_factor
        contourMapUsed[i] = (x, y, z)

    self.contourMap = contourMapUsed

def preliminaryResults():
    FP = floorPlan()
    FM = floorMap(0,0,18,18,5,4)
    FM.recordReading(0,3,2,-5)
    FM.recordReading(1,3,27,8)
    FM.recordReading(2,3,29,19)
    FM.recordReading(3,3,33,19)
    FM.recordReading(4,3,37,15)
    FM.recordReading(0,2,-2,3)
    FM.recordReading(1,2,-2,16)
    FM.recordReading(2,2,20,16)
    FM.recordReading(3,2,32,16)
    FM.recordReading(4,2,35,17)
    FM.recordReading(0,1,0,-5)
    FM.recordReading(1,1,3,-2)
    FM.recordReading(2,1,8,-8)
    FM.recordReading(3,1,35,22)
    FM.recordReading(4,1,27,19)
    FM.recordReading(0,0,-7,-2)
    FM.recordReading(1,0,-5,1)
    FM.recordReading(2,0,19,-1)
    FM.recordReading(3,0,8,1)
    FM.recordReading(4,0,19,17)
    FP.addMap(FM)
    A, b = FP.equationsForMatlab()
def display(L):
    res = ""
    for i in L:
        res += str(i)+" "
    return res

    for row in A:
        print display(row)
    print ""
    print display(b)
    print ""
    fubar = [[a, b] for a, b in FP.locations]
    for row in fubar:
print display(row)

def display(L):
    res = ""
    for i in L:
        res += str(i)+" \\
    return res

def floor1(getA = True, getb = True, getLocs = True, getSkeletonMap = True):
    na = 999
    Gmap1 = [...]
    Gmap2 = [...]
    Hmap = [...]
    Imap = [...]
    F1 = floorPlan()
    G1 = floorMap(14, 175, 18, 18, 6, 8)
    G2 = floorMap(122, 283, 18, 18, 4, 2)
    H = floorMap(14, 21, 24, 18, 6, 8)
    I = floorMap(158, 21, 24, 18, 6, 12)
    for grad, FM in [(Gmap1,G1),(Gmap2,G2),(Hmap,H),(Imap,I)]:
        for x in range(len(grad[0])):
            for y in range(len(grad[0])):
                p = grad[len(grad)-y-1][x]
                FM.recordReading(x,y,p[0],p[1])
    F1.addMap(G1)
    F1.addMap(G2)
    F1.addMap(H)
    F1.addMap(I)
    F1.addRoom((0,108,160,304))
    F1.addRoom((108,180,268,304))
    F1.addRoom((0,146,0,154))
    F1.addRoom((146,278,0,228))

    return F1

A,b = F1.equationsForMatlab()
if getA:
    for row in A:
        print display(row)
if getb:
    print display(b)
if getLocs:
    fubar = [[a,b] for a,b in F1.locations]
    for row in fubar:
        print display(row)
if getSkeletonMap:
    floor1_contourMap = [...]
    F1.removeFloor(floor1_contourMap)
    for x,y,z in F1.contourMap:
        print display([x,y,z])

def floor2(getA = True, getb = True, getLocs = True, getSkeletonMap = True):
    na = 999
Amap = [...]  
Bmap = [...]  
C1map = [...]  
C2map = [...]  
C3map = [...]  
Dmap = [...]  
Emap = [...]  
F2 = floorPlan()  
A = floorMap(14, 216, 24, 24, 3, 4)  
B = floorMap(14, 0, 24, 18, 5, 8)  
C1 = floorMap(81, 142, 9, 24, 4, 7)  
C2 = floorMap(81, 214, 9, 24, 5, 1)  
C3 = floorMap(117, 142, 9, 24, 5, 2)  
D = floorMap(126, 0, 24, 18, 6, 8)  
E = floorMap(168, 151, 24, 18, 6, 9)  
for grad, FM in [(Amap, A), (Bmap, B), (C1map, C1), (C2map, C2), (C3map, C3), (Dmap, D), (Emap, E)]:  
    for x in range(len(grad[0])):  
        for y in range(len(grad)):  
            p = grad[len(grad)-y-1][x]  
            FM.recordReading(x, y, p[0], p[1])  
F2.addMap(A)  
F2.addMap(B)  
F2.addMap(C1)  
F2.addMap(C2)  
F2.addMap(C3)  
F2.addMap(D)  
F2.addMap(E)  
F2.addRoom((0, 72, 216, 294))  
F2.addRoom((0, 114, 0, 138))  
F2.addRoom((72, 150, 140, 294))  
F2.addRoom((120, 276, 0, 138))  
F2.addRoom((162, 294, 150, 294))  
return F2  
A, b = F2.equationsForMatlab()  
if getA:  
    for row in A:  
        print display(row)  
if getb:  
    print display(b)  
if getLocs:  
    fubar = [[a, b] for a, b in F2.locations]  
    for row in fubar:  
        print display(row)  
if getSkeletonMap:  
    floor2_contourMap = [...]  
    F2.removeFloor(floor2_contourMap)  
    for x, y, z in F2.contourMap:  
        print display([x, y, z])
if False: preliminaryResults()

Floor1ContourMap = [...] 
Floor2ContourMap = [...] 

Floor1Rooms = floor1().rooms 
Floor2Rooms = floor2().rooms

def floatRange(start, end = None, step = 1):
    # floatRange replaces range, but inputs don't have to be integers
    # >>> floatRange(1.1, 2.2, .3) # count from 1.1 to 2.2 by .3's
    # [1.1, 1.4, 1.7, 2.0]
    if end == None:
        # range(n) implies range(0,n), so floatRange also implements
        # this feature
        return floatRange(0, start, step)
    if step == 0:
        raise "Error: Step cannot be 0"
    def done(val):
        if step > 0:
            # If the step is positive, the range is done when it's
            # high enough. The "-.1*step" helps avoid round-off
            # error. If python's arithmetic was perfect, it would
            # be unnecessary
            return value > end -.1*step
        return value < end -.1*step
    result = []
    value = start
    while not done(value):
        result.append(value)
        value += step
    return result

def integral(f, a, b, dx = .1**3):
    # Requires floatRange
    # Returns definite integral of f(x) between x = a and x = b
    # >>> def f(x): return x**2
    # >>> integral(f,0,3)
    # 9
    if b < a:
        return -integral(f, b, a, dx)
    if dx*100 > b-a and not(a==b):
        return integral(f, a, b, .01*(b-a))
    result = 0
    for x in floatRange(a, b, dx):
        result += dx * ( f(x) + 4*f(x+.5*dx) + f(x+dx) ) / 6.0
    return result

def doubleIntegral(f, x1, x2, y1, y2, dx = .1):
    # Requires floatRange, integral
    # Returns definite integral of f(x,y) between x1,x2,y1, and y2
    # >>> def f(x,y): return x*x + y*y
doubleIntegral(f, -.5,.5,-.5,.5)
0.166666666667

def row(y):
    return integral(lambda x:f(x,y), x1,x2, dx)

    return integral(lambda y:row(y), y1,y2, dx)

# The eleven floor contours
def f1(x,y):
return (1-x)*(1-y)-0.25
def f2(x,y): return x*y-0.25
def f3(x,y): return x*(1-y)-0.25
def f4(x,y): return (1-x)*y-0.25
def f5(x,y): return 1.075-x*x*(3-x-x)-y*y*(3-y-y)-x*y*((1-x-x)*(1-x)+(1-y-y)*(1-y)-1)
def f6(x,y): return f5(1-x,1-y)
def f7(x,y): return f5(1-x,y)
def f8(x,y): return f5(x,1-y)
def f9(x,y): return 0.4-((x**4-2*x**3+3*x*y**4-2*y**3+3*y**3))
def f10(x,y): return 1./15-((x*(x-1)**2+(y*(y-1)**2))
def f11(x,y): return 1.
deformationList = [f1,f2,f3,f4,f5,f6,f7,f8,f9,f10,f11]

for f in deformationList[:-1]:
    assert( abs(doubleIntegral(f,0,1,0,1,0.01)) < .1**6 )

viewContours = False
if viewContours:
    for f in deformationList:
        for y in floatRange(0,1.2,0.2):
            print f(0.0,y), f(0.2,y), f(0.4,y), f(0.6,y), f(0.8,y), f(1.0,y)
print ""

#assert(False)
def computeCoeff(floorData, fit):
# returns A s.t (data - a*fit) dot fit = 0
    fitDOTfit = 0
    floorDataDOTfit = 0
    for x,y,z in floorData:
        fitZ = fit(x,y)
        fitDOTfit += fitZ*fitZ
        floorDataDOTfit += fitZ*z
    return float(floorDataDOTfit)/fitDOTfit

ModelTest = []
sufficientlyCloseToInfinity = 10

for ContourMap, Rooms in [(Floor1ContourMap,floor1().rooms), (Floor2ContourMap,floor2().rooms)]:
    for minX, maxX, minY, maxY in Rooms:
        XYZlist = []
for x,y,z in ContourMap:
    if minX <= x and x <= maxX and minY <= y and y <= maxY:
        xNorm = float(x-minX)/(maxX-minX)
        yNorm = float(y-minY)/(maxY-minY)
        XYZlist.append((xNorm,yNorm,z))
# XYZlist now only includes the room

# Decompose into components
componentList = [0]*len(deformationList)
for k in range(sufficientlyCloseToInfinity):
    for j in range(len(componentList)):
        deformation = deformationList[j]
        component = computeCoeff(XYZlist,deformation)
        if component + componentList[j] > 0 or j == 1:
            componentList[j] += component
        for i in range(len(XYZlist)):
            x,y,z = XYZlist[i]
            z -= component*deformation(x,y)
            XYZlist[i] = (x,y,z)

# print componentList
Predicted = sum([abs(componentList[j])*sum([abs(deformationList[j](x,y)) for x,y,z in XYZlist]) for j in range(len(componentList))])
Unpredicted = sum([abs(z) for x,y,z in XYZlist])
ModelTest.append(Predicted/Unpredicted)

print ModelTest
print sum(ModelTest)/len(ModelTest)
ModelTest = [1/(n+1) for n in ModelTest]
print ModelTest
mean = sum(ModelTest)/len(ModelTest)
print mean
print math.sqrt(sum([(n-mean)**2 for n in ModelTest])/len(ModelTest))