TopoDroid loop closure error compensation

marco corvi - May 2021

Loop closure error compensation is probably the only data-reduction core function that differs among the various PC cave programs. The formulas of data raduction to convert polar values (distance, azimuth, and inclination) to rectangular values (east, north, and vertical) are standard and common to every PC cave program. The program might differ in the way they compute the magnetic declination from the given geographical coordinates of survey stations (usually cave entrances) are the date of the survey, depending on the underlying library that is used. However the differences are usually rather small. On the other hand, there is no commonly agreed approach to loop closure error compensation, and different programs have different implementions. Compass [1] follows an incemental approach in which small loops are compensated first. Survex [2] has a global approach with a minimum least squared. Therion [3] ...

TopoDroid does not perform loop closure error compensation by default, because, by leaving loops opens it makes it easier to spot closure errors and to get a feeling for the size of the misclosure by inspecting the survey centerline in the sketch plan and profile views.

Nevertheless, Topodroid includes a loop closure error compensation function.

The survey graph

The survey centerline forms a network with "branches" (sequences of legs) connecting "joins" (stations with three or more legs). We can includes among the joins the end-stations, with only one leg. Matematically this is called a *graph*, ie, a set of *nodes* (the joins) and *edges* (the branches) where each edge connects two nodes. If a survey is disconnected the loop closure error compensation applies indipendently in every connected piece. In other words we can restrict our attention to a connected survey, ie, a *connected graph*. In graph theory a loop is called *cycle*.

There is a difference between the leg from station A to station B, and the leg from station B to station A. The second is the opposite of the first. Mathematically we can write Leg(B,A) = - Leg(A,B). Infact the coordinates of B are obtained by adding the (ractangular) values of Leg(A,B) to the coordinates of A,

B = A + Leg(A,B)

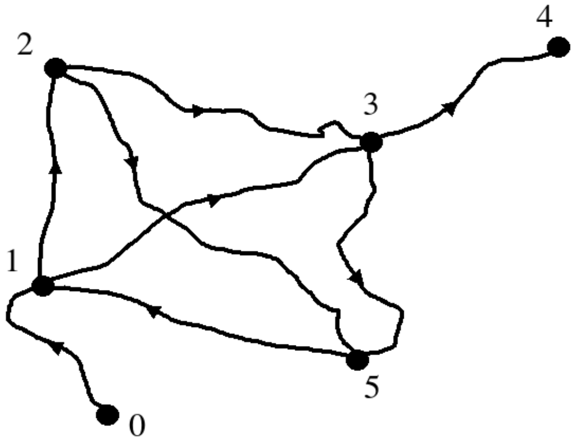
And so

A = B + Leg(B,A)

implies

Leg(B,A) = - Leg(A,B).

In graph language we say that the graph is *directed*. When we write Edge(A,B), or (A-B), we mean the edge from node A to node B.



The figure above shows a connected graph with six nodes and eight edges, namely 0-1, 1-2, 2-3, 2-5, 1-3, 3-4, 3-5, and 5-1. A sequence of edges such that the second node of each edge coincides with the first node of the next edge, is a *path*. For example

(1-2) + (2-3) + (3-4)

is a path from node 1 to node 4. We write (2-1) to denote the edge (1-2) traversed in the reversed direction. Therefore the path that goes from node 1 to node 2 and then back to node 1 is

(1-2) + (2-1).

If the first node of the first edge coincides with the second node of the last edge, the path is a *cycle*. The last example is therefore a cycle. Another example of cycle is

(1-2) + (2-3) + (3-5 + (5-1).

In the example of the figure there are many cycles. For example,

(1-2) + (2-5) + (5-1)
(2-3) + (3-5) + (5-2)
(1-2) + (2-3) + (3-5) + (5-1)
(1-2) + (2-3) + (3-5) + (5-2) + (2-1)
(2-3) + (3-5) + (5-2) + (2-3) + (3-5) + (5-1) + (1-2).
(1-2) + (2-3) + (3-2) + (2-1)

The fourth cycle contains the edge (1-2) twice, traversed in opposite directions, so it does not "cycle" over it. The fifth cycle contains the edge (2-3) twice, in the same direction, so it "cycles" twice over it. The last cycle is "anomalous": first it traverses the edge (1-2) and the edge (2-3), then it traces back them in reserved order. This cycle is "contratible" to a single node, namely the node 1, ie, a cycle with no edge. Such a cycle is called "null cycle" (at the node 1).

The cycles in the previous example are not "independent". For example we can "compose" cycle (1-2)+(2-5)+(5-1) with cycle (2-3)+(3-5)+(5-2): the edge between node 2 and node 5 is traversed in opposite directions in the two cycles and thus they "calcel out" and there remain the cycle (1-2)+(2-3)+(3-5)+(5-1). Formally one writes

(1-2)+(2-5)+(5-1) + (2-3)+(3-5)+(5-2) = (1-2)+(2-3)+(3-5)+(5-1).

A set of cycles is a set of *independent cycles* is no linear combination (with integer coefficients) of the cycles is identically 0. For example a cycle C and the opposite cycle -C, obtained by reversing the sign of each edge in C, and not independent. Indeed -C corresponds to traversing C in the opposite direction.

A set of independent cycles is *maximal* if any cycle of the graph can be written as linear combination of the cycles of the set. A maximal set of independent cycles forms a basis for the cycles.

All the maximal sets of independent cycles of a (connected) graph, have the same number of cycles. Therefore we can define the *number of independent cycles* of a graph. The number of cycles of a maximal set of independent cycles is called the number of independent cycles of the graph.

Proof.
if $\{C_i\}$ and $\{G_i\}$ are two maximal sets of independent cycle, we can write

$$G_i = \sum_{j=0..C} A_{ij}\, C_j$$

and

$$C_j = \sum_{k=0..G} B_{jk}\, G_k$$

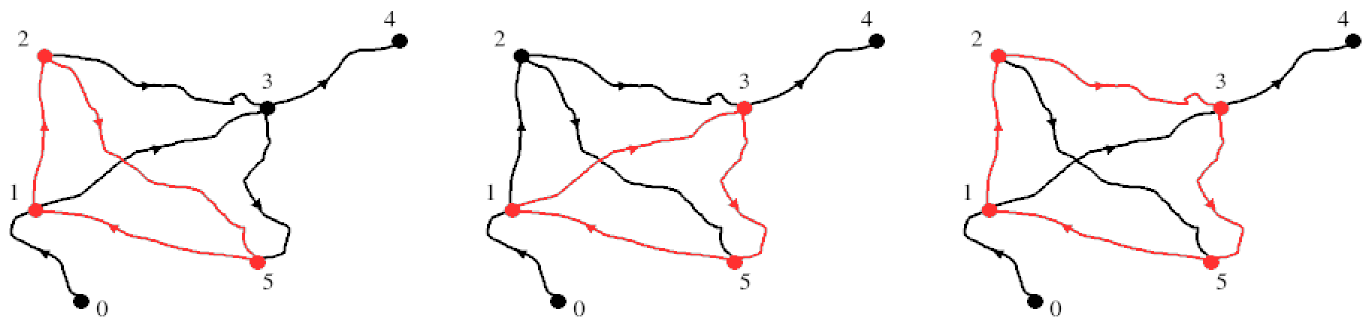
Therefore the matrices A and B are each other inverses and thus they are square matrices.

There is a relation among the number N of nodes, the number E of edges, and the number C of independent cycles in a graph,

$$C = 1 + E - N$$

[This is easily proved by induction on the number of edges].

In the example C = 1 + 8 - 6 = 3, therefore there a basis of cycles contains three independent cycles. An example of basis is (1-2)+(2-5)+(5-1), (1-3)+(3-5)+(5-1) and (1-2)+(2-3)+(3-5)-(5-1), shown in the figure below.



TopoDroid loop closure error compensation algorithm

The first step of TopoDroid loop closure error compnensation algorithm is the identify the survey graph. First the nodes (station with either one leg, or three or more legs) are identified. Next the legs attached to nodes are followed until another node is reached. By this means the edges between the nodes are identified. Each edge (branch) consists of a sequence of legs with a coefficient +1 if the leg is directed as the edge, or -1 if it is directed oppositely. The last leg of the edges is marked in order not to
add the opposite edge to the graph.

The second step consists in identifying a maximal set of independent cycles.

The cycles can be written as

$$C_i = \sum_{j=0..E} H_{ij}\, E_j$$

where H is the CxE incidence matrix, between cycles and edges. It has entries 0, +1, -1, the values denoting how an edge enters a cycle.

If there were perfect loop closure we would have

$$0 = \sum_{j=0..E} H_{ij}\, X_j$$

where X_j is the displacement vector of edge E_j. If the measured edge displacements have errors, $X'_j = X_j + d_j$ (the error being d_j), the loop closures do not valish,

$$c_i = \sum_{j=0..E} H_{ij}\, X'_j = \sum_{j=0..E} H_{ij}\, d_j \qquad (1)$$

To obtain dj we cannot apply the pseudoinverse. The matrix H_{ij} has rank C, the number of independent cycles, and the ExE matrix H^t H is singular.

Therefore we write the matrix $H_{ij} = (A_{ij},, B_{ik})$ where A_{ij} is square CxC matrix and B_{ik} is CxB matrix (B=E-C). Equation (1) becomes

$$c_j = \sum_{j=0..C} A_{ij}\, d'_j + \sum_{k=0..B} B_{ik}\, d''_k \qquad (2)$$

where d'j and d"k are the (unknown) error-compensation of the first C edges and the last B edges, respectively. By permuting the column of H is always possible to choose A non-singular. Therefore in the above equation the indices j,k are a permutation of 0, .. E-1: (j,k) = p(0,..,E-1). We drop the permutaton from the following to simplify the notation.

The error-corrections are found by imposing the requirement that the weigthed sum of the squared closure errors is minimal (w'_j and w''_k are the weights associated to edges E_j and E_k, respectively):

$$\min E(d'_j, d''_k) = \min(\sum_{j=0..C} w'_j\, d'_j{*}d'_j + \sum_{k=0..B} w''_k\, d''_k{*}d''_k)$$

Equation (2) can be solved to obtain

$$d'_j = \sum_{i=0..C} A^{-1}_{ji}\, C_i - \sum_{i=0..C} \sum_{k=0..B} A^{-1}_{ji}\, B_{ik}\, d''_k. \qquad (3)$$

For notational simplicity we introduce

$$G_j = A^{-1}_{ji}\, C_i$$
$$P_{jk} = A^{-1}_{ji}\, B_{ik}$$

We can now substitute d'_j in the expression of the function E() which becomes a function of d''_k, and take its derivative to find the minimum,

$$w''_k\, d''_k - w'_j\, P^t_{kj} (G_j - P_{jh}\, d''_h) = 0$$

i.e. (I_{kh} is the identity matrix),

$$(w''_k\, I_{kh} + w'_j\, P^t_{kj}\, P_{jh})\; d''_h = w'_j\, P^t_{kj}\, G_j$$

The matrix on the l.h.s. is invertible and this equation is solved to get d''_h. From (3) we then get $d_{'j}$.

Having obtained the values (d'_j, d''_k) that produce the loop misclosure values, these can be subtracted to the measured edges displacements, so that the corrected loop closure errors are compensated.

Edge weights

The loop closure algorithm must be applied independently for each displacement coordinates, E (east), S (south), and V (vertical). Each coordinate depends on the measured quantities, distance D, azimuth A, and inclination C, differently,

$$V = D \sin(C)$$
$$H = D \cos(C) \qquad (4)$$

$$E = H \sin(A)$$
$$S = - H \cos(A)$$

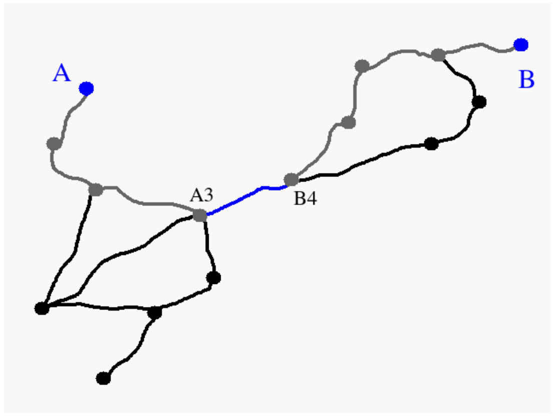
The errors on V,E,S, are obtained by differentiating (4). We assume D,A,C independent, and compose the errors of D,A,C.

$$|dV|^2 = V^2 |dD/D|^2 + H^2 |dC|^2$$
$$|dE|^2 = E^2 |dD/D|^2 + S^2 |dA|^2 + V^2 \sin^2(A)|dC|^2 \qquad (5)$$
$$|dS|^2 = S^2 |dD/D|^2 + E^2 |dA|^2 + V^2 \cos^2(A)|dC|^2$$

Assuming that $|dD/D| << |dA|, |dC|$ we can neglect the first term on the r.h.s of formulas (5).

Geolocalized stations

Geolocalized stations are assumed to have "fixed" coordinates, obtained through the geolocalization. When two caves with geolocalized entrance stations are joined a new kind of cycle occurs. This is a path P in the network graph of the caves that goes from one entrance to the other.



The closure error of a path between two geolocalized stations, A and B, is given by the path displacement minus the displacement between the two stations

$$dX = dX_{path} - (X_B - X_A)$$

and the error compensation is distributed on the path edges.

In the figure on the right, A and B are the geolocalized entrances of two joined caves. There is therefore a path from A to B (that passes through stations A3 and B4), for example the path shown in grey in the figure. Since the coordinates of A and B are known this path is like a cycle and its closure error is given by the displacement B-A along the path plus the displacement to go from B to A on the surface, ie, A-B on the surface. Thus the closure error is

$$(B - A)_{path} + (A - B)_{surface} = (B - A)_{path} - (X_B - X_A).$$

References

[1] L. Fish, The problem with least square loop closures, Compass & Tape, 13(1) n. 41 Apr. 1997, 15-18
[2] http://survex.com
[3] http://therion.speleo.sk

Appendix. Sample C++ implementation

To make the listing short, structs are used, and includes and calls to deallocation have been dropped.
The construction of the graph, ie, the identification of the nodes and the edges is not included in the listing.

```
struct Node // a node in the graph
{
    int id;  // (unique) identifier of the node
    int use; // whether the mode has been "used"

    Node( int nn ) : id( nn ), use(0) { }
    bool operator==( const Node & other ) const { return this->id == other.id; }
};

struct Edge // and edge in the graph
{
    int id;
    Node * n1;   // first node
    Node * n2;   // second node
    int use;
    double mLength; // edge length (always positive)
    double mX; // X displacement = n2.X - n1.X (it can be negative)
    double dX; // correction to the X displacement

    Edge( int n, Node * nn1, Node * nn2, double l, double t )
      : id( n ), n1( nn1 ), n2( nn2 ), use(0), mLength(t), mX( l ), dX(0)
    { }

    Node * other( Node * n ) const { return ( n1 == n )? n2 : ( n2 == n )? n1 : NULL; }
    double length() const { return mLength; }
    double x() const { return mX; }
    double x_corrected() const { return mX + dX; }
    void setDX( double dx ) { dX = dx; }
};

struct Step // and edge in a cycle
{
    Step * prev;   // previous step
    Edge * mEdge;  // edge
    Node * mNode;  // first node of the step
    int k;

    Step( Step * p, Edge * e, Node * t, int kk ) : prev( p ), mEdge( e ), mNode( t ), k( kk ) { }

    double x() const { return (mNode == mEdge->n1) ? mEdge->x() : - mEdge->x(); } // step displacement

    // step corrected-displacement
    double x_corrected() const { return (mNode == mEdge->n1) ? mEdge->x_corrected() : - mEdge->x_corrected(); }

    // incidence number of the edge with the step: whether this step contins the edge (forward +1, or reversed -1)
    int contains( const Edge * edge ) const
    {
      return ( edge != mEdge )? 0 : ( edge->n1 != mNode )? -1 : 1;
    }
};

struct Cycle /** a cycle is a sequence of steps */
{
    std::vector< Step * > mSteps;
    double mX; // total displacement of the cycle

    Cycle() : mX(0) { }

    void addStep ( Step * step ) // add a step to the cycle
    {
      mSteps.push_back( step );
      mX += step->x();
    }

    double x() const { return mX; } // cycle displcement

    double x_corrected() const // cycle corrected-displacement
    {
      double ret = 0;
      for ( auto step : mSteps ) ret += step->x_corrected();
      return ret;
    }

    int contains( const Edge * edge ) const // incidence number of the edge with the cycle
    {
      int ret = 0;
      for ( auto step : mSteps ) ret += step->contains( edge );
      return ret;
    }
};


void independentCycles( const std::vector< Edge * > & edges, std::vector< Cycle * > & cycles )
{
  int j;
  int bs = edges.size();
  std::stack< Step * > stack;
  for ( int k0 = 0; k0 < bs; ++k0 ) {
    Edge * b0 = edges[ k0 ];
    if ( b0->use == 2 ) continue;

    Node * n0 = b0->n1; // start-mNode for the cycle
    n0->use = 0;        // but start-mNode is not used
    b0->use = 1;        // start-branch is used
    stack.push( new Step( NULL, b0, b0->n2, k0 ) ); // step with b0 to the second node (k0 = where start scan branches
    while ( ! stack.empty() ) {
      Step * s1 = stack.top();
      Node * n1 = s1->mNode;
      s1->k ++;
      int k1 = s1->k;
      if ( n1 == n0 ) {
        Cycle * cycle = new Cycle();
        for ( Step * step = s1; step != NULL; step = step->prev ) {
          cycle->addStep( step );
        }
        cycles.push_back( cycle );
        s1->mEdge->use = 2; // mark the last edge as no longer usable
        s1->mNode->use = 0;
        stack.pop();
      } else {
        int k2 = s1->k;
        for ( ; k2 < bs; ++k2 ) {
          Edge * b2 = edges[ k2 ];
          if ( b2->use != 0 ) continue;
          Node * n2 = b2->other( n1 );
          if ( n2 != NULL && n2->use == 0 ) {
            b2->use = 1;
            n2->use = 1;
            stack.push( new Step( s1, b2, n2, k0 ) );
            s1->k = k2;
            break;
          }
        }
        if ( k2 == bs ) {
          s1->mEdge->use = 0;
          s1->mNode->use = 0;
          stack.pop();
        }
      }
    }
    b0->use = 2;
  }
}

// The incidence matrix has Ne columns and Nc rows.
int * incidenceMatrix( const std::vector< Edge * > & edges, const std::vector< Cycle * > & cycles )
{
  int ne = edges.size();
  int nc = cycles.size();
  int * A = new int[ ne * nc ];
  for ( int i = 0; i < nc; ++i ) { // rows: cycle index
    for ( int j=0; j<ne; ++j ) {   // columns: edge index
      A[ i * ne + j ] = cycles[i]->contains( edges[j] );
    }
  }
  return A;
}

void swapColumns( int i1, int i2, int * A, int nc, int ne ) // swap columns i1 and i2
{
  for ( int j=0; j<nc; ++j ) swap( A[ j*ne + i1 ], A[ j*ne + i2] );
}

// identify a set of independent columns in the incidence matrix and permute the columns of A
void independentColumns( int * A, int nc, int ne, int * permutation )
{
  for ( int k=0; k<ne; ++k ) permutation[k] = k;
  int k1 = nc;
  for ( int i=0; i<nc; ++i ) {
    int k = i;
    for ( ; k<ne; ++k ) {
      if ( A[ i*ne + k ] != 0 ) break;
    }
    for ( ; k<ne; ++k ) {
      int cnt = 1;
      for ( int j=i+1; j<nc; ++j ) if ( A[ j*ne + k ] == 0 ) ++cnt;
      if ( cnt == nc - i ) break;
    }
    if ( k != i ) {
      swapColumns( i, k, A, nc, ne );
      int a = permutation[i];
      permutation[i] = permutation[k];
      permutation[k] = a;
    }
  }
}

// compute the inverse of A in-place (usually, nd = nc)
double computeInverse(double * A, int nr, int nc, int nd)
{
  ... // standard pivot inverse
}

void correctCycles( int * A, double * C, double * w, double * Error, int ne, int nc )
{
  int * permutation = new int[ ne ];
  independentColumns( A, nc, ne, permutation );
  // permutation[0] is the first indep. column, permutation[1] is the second, and so on

  // split incidence matrix in A-part and B-part
  int nb = ne - nc;
  double * A0 = new double[ nc * nc ];
  double * B0 = new double[ nb * nc ];
  for ( int i=0; i<nc; ++i ) {
    for ( int k=0; k<nc; ++k ) A0[i*nc + k] = A[ i*ne + k];      // row i, column k
    for ( int j=0; j<nb; ++j ) B0[i*nb + j] = A[ i*ne + nc + j]; // row i, column j
  }

  computeInverse( A0, nc, nc, nc );  // A0 inverse

  double * A0B = new double[ nc * nb ]; // A0^-1 * B
  double * A0C = new double[ nc ];      // A0^-1 * C
  for ( int i=0; i<nc; ++i ) {
    A0C[i] = 0;
    for ( int k=0; k<nc; ++k ) A0C[i] += A0[i*nc + k] * C[k];
    for ( int j=0; j<nb; ++j ) {
      A0B[ i*nb + j ] = 0;
      for ( int k=0; k<nc; ++k ) A0B[ i*nb + j ] += A0[i*nc + k] * B0[k*nb + j];
    }
  }

  double * F = new double[ nb ]; // F = A0B^t A0C
  for ( int i=0; i<nb; ++i ) {
    F[i] = 0;
    for ( int k=0; k<nc; ++k ) F[i] += A0B[k*nb + i] * w[ permutation[k] ] * A0C[k];
  }

  computeInverse( D, nb, nb, nb ); // D inverse

  double * E0 = new double[ ne ];   // Eps_1 = D^-1 * F: egde correction errors
  for ( int i=0; i<nb; ++i ) { // second part
    E0[nc+i] = 0;
    for ( int k=0; k<nb; ++k ) E0[nc+i] += D[i*nb + k] * F[k];
  }
  for ( int i=0; i<nc; ++i ) { // first part
    E0[i] = A0C[i];
    for ( int k=0; k<nb; ++k ) E0[i] -= A0B[i*nb + k] * E0[nc + k];
  }
  for ( int i=0; i<ne; ++i ) Error[ permutation[i] ] = - E0[i];
}

void errorClosureCompensation( std::vector< Node * > nodes, std::vector< Edge * > edges )
{
  std::vector< Cycle * > cycles;
  independentCycles( edges, cycles );
  int * A = incidenceMatrix( edges, cycles );
  double * C = new double[ cycles.size() ];
  for ( int k=0; k<cycles.size(); ++k ) C[k] = cycles[k]->x();
  double * W = new double[ edges.size();
  for ( int k=0; k < edgese.size(); ++k ) W[k] = 1; // fill the weight with proper values
  double * Err = new double[ edges.size() ];
  correctCycles( A, C, W, Err, edges.size(), cycles.size() );
  for ( int i=0; i<edges.size(); ++i ) edges[ i ]->setDX( Err[i] );
}
```