Translating SAR Coordinates into Geographic Coordinates

Ishuwa Sikaneta

December 11, 2008

Contents

This note details a Newton method like approach for transforming SAR coordinates, i.e. (azimuth, range) into geographic coordinates

1 The system of equations

Given, the time of imaging, t_0, the radar squint angle, θ_{sq}, and the state vector of the radar platform, position $\vec{x}_{sat}(t_0) = \vec{x}_{sat} = [x_p, y_p, z_p]^T$, velocity $\vec{v}_{sat}(t_0) = \vec{v}_{sat} = [v_x, v_y v_z]^T$, in Cartesian ECEF coordinates, one can construct a system of equations that allows the image plane coordinates to be expressed as ECEF Cartesian coordinates.

We construct three equations that constrain the target position, $\vec{x} = [x, y, z]^T$. First the target should lie upon the surface of Earth. This is ensured by the contraint that its coordinates satisfy the equation of the ellipsoid.

$$f_1(x, y, z) = \frac{x^2}{r_a^2} + \frac{y^2}{r_a^2} + \frac{z^2}{r_b^2} - 1 = 0. \tag{1}$$

We choose for this note, the WGS84 ellipsoid, $r_a = 6,378,137.0$, $r_b = 6,356,752.3$.

Next, given the range to the target, r, we observe the constraint that

$$\begin{aligned} f_2(x,y,z) &= r^2 - |\vec{x}_{sat} - \vec{x}|^2, \\ &= r^2 - (x_p - x)^2 - (y_p - y)^2 - (z_p - z)^2, \\ &= 0. \end{aligned} \tag{2}$$

Finally, given the squint angle of the system, we ensure that the beam is centred on the target by observing the constraint that

$$\begin{aligned} f_3(x,y,z) &= (\vec{x} - \vec{x}_{sat}) \cdot \vec{v}_{sat} - r|\vec{v}_{sat}| \sin\theta_{sq}, \\ &= xv_x + yv_y + zv_z - x_p v_x - y_p v_y - z_p v_z - r|\vec{v}_{sat}| \sin\theta_{sq}, \\ &= 0. \end{aligned} \tag{3}$$

These three equations are neatly summarized in the vector equation

$$\vec{f}(\vec{x}) = \begin{bmatrix} f_1(\vec{x}) \\ f_2(\vec{x}) \\ f_3(\vec{x}) \end{bmatrix} = \begin{bmatrix} \frac{x^2}{r_a^2} + \frac{y^2}{r_a^2} + \frac{z^2}{r_b^2} - 1 \\ r^2 - (x_p - x)^2 - (y_p - y)^2 - (z_p - z)^2 \\ xv_x + yv_y + zv_z - x_p v_x - y_p v_y - z_p v_z - r|\vec{v}_{sat}| \sin\theta_{sq} \end{bmatrix} = \vec{0}. \tag{4}$$

2 Newton method

To solve the vector equation in 4 one can use Newton's iterative formula defined as

$$\vec{x}_{n+1} = \vec{x}_n - \mathbf{J}^{-1}(\vec{x}_n)\vec{f}(\vec{x}_n), \tag{5}$$

where

$$\begin{aligned} \mathbf{J}(\vec{x}) &= \begin{bmatrix} \frac{\partial f_1}{\partial x} & \frac{\partial f_1}{\partial y} & \frac{\partial f_1}{\partial z} \\ \frac{\partial f_2}{\partial x} & \frac{\partial f_2}{\partial y} & \frac{\partial f_2}{\partial z} \\ \frac{\partial f_3}{\partial x} & \frac{\partial f_3}{\partial y} & \frac{\partial f_3}{\partial z} \end{bmatrix} \\ &= \begin{bmatrix} 2x/r_a^2 & 2y/r_a^2 & 2z/r_b^2 \\ 2(x_p - x) & 2(y_p - y) & 2(z_p - z) \\ v_x & v_y & v_z \end{bmatrix}. \end{aligned} \tag{6}$$

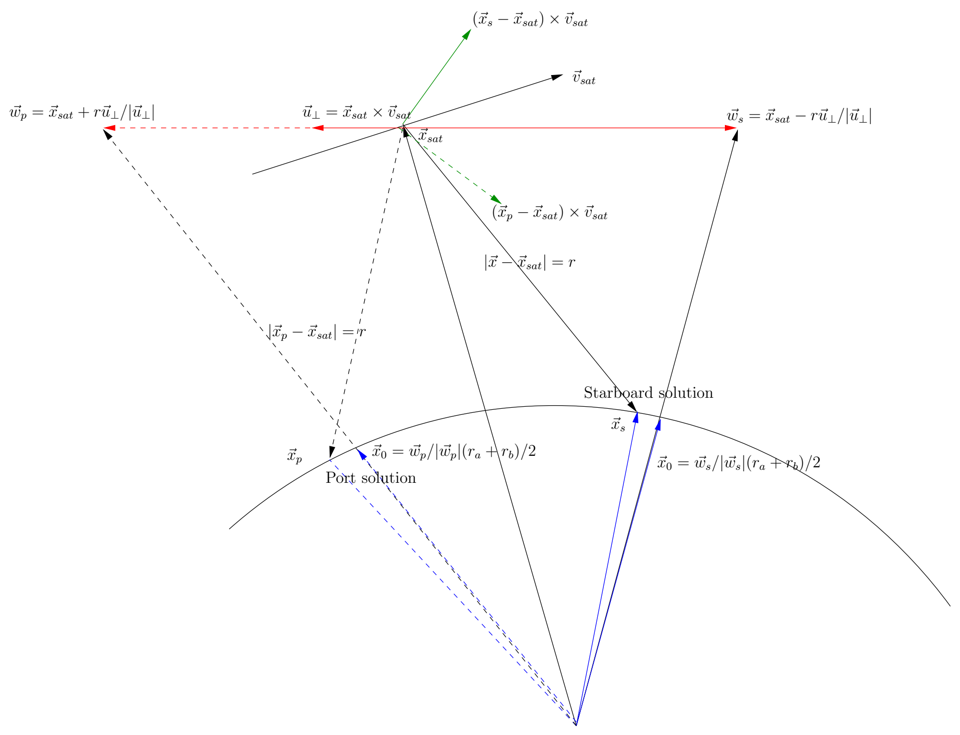
To find the inverse of $\mathbf{J}(\vec{x})$, one can use the expression for the cofactor, C_{ij}, formula of the inverse of a 3×3 matrix

$$\begin{aligned} \mathbf{J}^{-1} &= \frac{1}{|\mathbf{J}|} \begin{bmatrix} C_{11} & C_{21} & C_{31} \\ C_{12} & C_{22} & C_{32} \\ C_{13} & C_{23} & C_{33} \end{bmatrix} \\ &= \frac{1}{|\mathbf{J}|} \begin{bmatrix} J_{22}J_{33} - J_{23}J_{32} & -J_{21}J_{33} + J_{23}J_{31} & J_{21}J_{32} - J_{22}J_{31} \\ -J_{12}J_{33} + J_{13}J_{32} & J_{11}J_{33} - J_{13}J_{31} & -J_{11}J_{32} + J_{12}J_{31} \\ J_{12}J_{23} - J_{13}J_{22} & -J_{11}J_{23} + J_{13}J_{21} & J_{11}J_{22} - J_{12}J_{21} \end{bmatrix}^T. \end{aligned} \tag{7}$$

A C/C++ implementation of the Newton method is provided in the following.

3 The initial guess and differentiating starboard from port

As seen in figure 3, there are two possible solutions to the system defined by equation 4. Presumably, by choosing an initial guess close to the desired solution, the Newton method will converge to the desired solution. This is not proved.



$(\vec{x}_s - \vec{x}_{sat}) \times \vec{v}_{sat}$
$\vec{v}_{sat}$
$\vec{w}_p = \vec{x}_{sat} + r\vec{u}_\perp/|\vec{u}_\perp|$
$\vec{u}_\perp = \vec{x}_{sat} \times \vec{v}_{sat}$
$\vec{w}_s = \vec{x}_{sat} - r\vec{u}_\perp/|\vec{u}_\perp|$
$\vec{x}_{sat}$
$(\vec{x}_p - \vec{x}_{sat}) \times \vec{v}_{sat}$
$|\vec{x} - \vec{x}_{sat}| = r$
$|\vec{x}_p - \vec{x}_{sat}| = r$
Starboard solution
$\vec{x}_s$
$\vec{x}_p$
$\vec{x}_0 = \vec{w}_p/|\vec{w}_p|(r_a + r_b)/2$
$\vec{x}_0 = \vec{w}_s/|\vec{w}_s|(r_a + r_b)/2$
Port solution

Figure 1: Geometry of the solution space.

To set up the initial guess for $\vec{x}$, we propose the following. First construct the cross product $\vec{u}_\perp = \vec{x}_{sat} \times \vec{v}_{sat}$ which points port from the radar platform. We then construct $\vec{w} = \vec{x}_{sat} \pm r\vec{u}_\perp/|\vec{u}_\perp|$. The positive sign corresponds to port looking, the negative sign to starboard looking. The point defined by $\vec{w}$ now lies above the planet so we move it back down, closer to the ellipsoid, by constructing $\vec{x}_0 = \vec{w}/|\vec{w}|(r_a + r_b)/2$.

One possibility to test the obtained solution, to ensure that it is either to port or starboard, is to test the sign of $[(\vec{x} - \vec{x}_{sat}) \times \vec{v}_{sat}] \cdot \vec{x}_{sat}$. A positive sign indicates the starboard solution, a negative sign indicates the port solution.

4 Modifications required to include a height estimate from a DEM

A slight modification has to be made when the target of interest cannot be assumed to lie on the WGS-84 ellipsoid. A correction must be made to account for the extra ellipse radius imparted by the additional height. This section proposes a method to modify equation 1 to account for the extra height. As illustrated in figure 4, let this height above the ellipsoid be denoted by h.

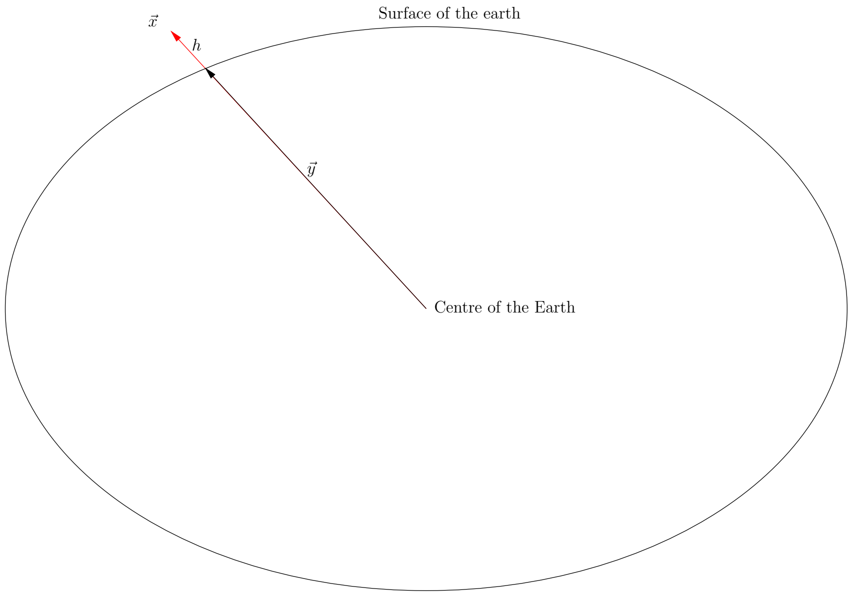


Figure 2: A target with height above the ellipsoid.

Assume that $\exists \vec{y}$ that satisfies equation 1. That is, in vector notation,

$$\vec{y}^T \mathbf{A} \vec{y} = 1, \tag{8}$$

where

$$\mathbf{A} = \begin{bmatrix} r_a^{-2} & 0 & 0 \\ 0 & r_a^{-2} & 0 \\ 0 & 0 & r_b^{-2} \end{bmatrix}. \tag{9}$$

From figure 4, one can see that the point that we are really interested in satisfies the relation

$$\vec{x} = \frac{|\vec{y}| + h}{|\vec{y}|} \vec{y}, \tag{10}$$

from which one can deduce that

$$\begin{aligned}\vec{x}^T\mathbf{A}\vec{x} &= \left[\frac{|\vec{y}|+h}{|\vec{y}|}\right]^2 \vec{y}^T\mathbf{A}\vec{y},\\ &= \left[\frac{|\vec{y}|+h}{|\vec{y}|}\right]^2,\end{aligned} \tag{11}$$

and since $|\vec{x}| = h + |\vec{y}|$ from equation 10, one concludes that

$$\begin{aligned}f_1(x,y,z) &= \vec{x}^T\mathbf{A}\vec{x} - \left[\frac{|\vec{x}|}{|\vec{x}|-h}\right]^2,\\ &= \frac{x^2}{r_a^2}+\frac{y^2}{r_a^2}+\frac{z^2}{r_b^2}-\frac{x^2+y^2+z^2}{[\sqrt{x^2+y^2+z^2}-h]^2},\\ &= 0.\end{aligned} \tag{12}$$

The vector function equation becomes

$$\vec{f}(\vec{x}) = \begin{bmatrix} f_1(\vec{x})\\ f_2(\vec{x})\\ f_3(\vec{x})\end{bmatrix} = \begin{bmatrix} \frac{x^2}{r_a^2}+\frac{y^2}{r_a^2}+\frac{z^2}{r_b^2}-\frac{x^2+y^2+z^2}{[\sqrt{x^2+y^2+z^2}-h]^2}\\ r^2-(x_p-x)^2-(y_p-y)^2-(z_p-z)^2\\ xv_x+yv_y+zv_z-x_pv_x-y_pv_y-z_pv_z-r|\vec{v}_{sat}|\sin\theta_{sq}\end{bmatrix} = \vec{0}, \tag{13}$$

from which the Jacobian matrix can be derived as

$$\mathbf{J} = \begin{bmatrix} 2x/r_a^2+\frac{2xh}{(\sqrt{x^2+y^2+z^2}-h)^3} & 2y/r_a^2+\frac{2yh}{(\sqrt{x^2+y^2+z^2}-h)^3} & 2z/r_b^2+\frac{2zh}{(\sqrt{x^2+y^2+z^2}-h)^3}\\ 2(x_p-x) & 2(y_p-y) & 2(z_p-z)\\ v_x & v_y & v_z\end{bmatrix}. \tag{14}$$

5 Code

This section outlines a C/C++ implementation of the above procedure

```
// Necessary Headers
#include <iostream>
#include <stdlib.h>
#include <string.h>
#include <iomanip>
#include <math.h>
#include "myPosition.hpp"
#include "radar.h"

const double myPosition::majaxis = 6378137.0;
const double myPosition::minaxis = 6356752.3142;;
const double myPosition::pi = 3.1415926535;

//Function to transform Latitude, Longitude and Ellipsoidal height
```

```
// to x,y,z cartesian coordinates
void myPosition::LL2xyz()
{
    double N;// Prime Vertical radius of curvature
    double f;// Flattening
    double e;// Eccentricity

    f = (majaxis−minaxis)/majaxis;
    e = 2.0*f−f*f;
    N = majaxis/sqrt(1−e*pow(sin(latitude),2));
    x = (N+height)*cos(latitude)*cos(longitude);
    y = (N+height)*cos(latitude)*sin(longitude);
    z = (N*(1−e)+height)*sin(latitude);
}

//Function to transfer x,y,z cartesian coordinates
//to Latitude, Longitude and Ellipsoidal height
void myPosition::xyz2LL()
{
    double P;//Distance between points???
    double eprime;
    double latitudeP; // utility variable
    double f;// Flattening
    double e;// Eccentricity
    double N;// Prime Vertical radius of curvature

    P = sqrt((xt*xt)+(yt*yt));
    latitude = atan(zt*majaxis/(P*minaxis));
    eprime = (pow(majaxis,2)−pow(minaxis,2))/pow(minaxis,2);
    f = (majaxis−minaxis)/majaxis;
    e = 2*f−f*f;
    longitude = atan2(yt,xt);
    do{
        latitudeP = latitude;
        latitude = atan((zt+eprime*minaxis*pow(sin(latitudeP),3))/(P−e*majaxis*pow(cos(latitudeP),3)));
    }while(fabs(latitude−latitudeP)>1e−9);
    N = majaxis/sqrt(1−e*pow(sin(latitude),2));
    height = (P/cos(latitude))−N;

}

// Newton function to compute coordinates
double myPosition::advance(double *u, double *v, double *w)
{
  // Get the terrain height at the current position. Maybe from DEM
  double h = getTerrainHeight(*u, *v, *w);

  // Compute function values at current point
  double f1 = range*range − (*w−z)*(*w−z) − (*u−x)*(*u−x) − (*v−y)*(*v−y);
  double f2 = vx*(x−*u)+vy*(y−*v)+vz*(z−*w) + sqrt(vx*vx+vy*vy+vz*vz)*sin(squint)*range;
  double f3 = pow(*u,2.0)/pow(majaxis,2.0)+pow(*v,2.0)/pow(majaxis,2.0)+pow(*w,2.0)/pow(minaxis,2.0)
              −(pow(*u,2.0)+pow(*v,2.0)+pow(*w,2.0))/pow(sqrt(pow(*u,2.0)+pow(*v,2.0)+pow(*w,2.0))−h,2.0);

  // Compute Jacobian matrix at current point
  double j11 = −2.0*(*u−x);
  double j12 = −2.0*(*v−y);
  double j13 = −2.0*(*w−z);
  double j21 = −vx;
  double j22 = −vy;
  double j23 = −vz;
  double j31 = 2.0*(*u)/pow(majaxis,2.0)+2.0*(*u)*h/pow(sqrt(pow(*u,2.0)+pow(*v,2.0)+pow(*w,2.0))−h,3.0);
  double j32 = 2.0*(*v)/pow(majaxis,2.0)+2.0*(*v)*h/pow(sqrt(pow(*u,2.0)+pow(*v,2.0)+pow(*w,2.0))−h,3.0);
  double j33 = 2.0*(*w)/pow(minaxis,2.0)+2.0*(*w)*h/pow(sqrt(pow(*u,2.0)+pow(*v,2.0)+pow(*w,2.0))−h,3.0);

  // Compute the determinant of the Jacobian
  double jdet = j11*(j22*j33−j32*j23) − j12*(j21*j33−j23*j31) + j13*(j21*j32−j22*j31);

  // Compute the inverse Jacobian matrix
  double inv11 = +1.0*(j22*j33−j32*j23)/jdet;
  double inv21 = −1.0*(j21*j33−j23*j31)/jdet;
  double inv31 = +1.0*(j21*j32−j22*j31)/jdet;
  double inv12 = −1.0*(j12*j33−j13*j32)/jdet;
  double inv22 = +1.0*(j11*j33−j13*j31)/jdet;
  double inv32 = −1.0*(j11*j32−j12*j31)/jdet;
  double inv13 = +1.0*(j12*j23−j13*j22)/jdet;
  double inv23 = −1.0*(j11*j23−j13*j21)/jdet;
  double inv33 = +1.0*(j11*j22−j12*j21)/jdet;

  // Compute the difference from current point
  double delu = inv11*f1+inv12*f2+inv13*f3;
  double delv = inv21*f1+inv22*f2+inv23*f3;
  double delw = inv31*f1+inv32*f2+inv33*f3;

  *u −= delu;
  *v −= delv;
  *w −= delw;

  return delv*delv+delu*delu+delw*delw;
}

// Compute the coordinates of the target in ECEF
int myPosition::findcoord(int side)
{
    // The member variables (x,y,z) for the radar position and
    // (vx,vy,vz) for the radar velocity vector are already set

    // First determine a vector that points perpendicularly to the
    // track vector and the sub platform point to radar point vector
    // Use the cross product to determine this vector
    double ux=y*vz−z*vy;
    double uy=z*vx−x*vz;
    double uz=x*vy−y*vx;
    double un=sqrt(ux*ux+uy*uy+uz*uz);
    ux/=un;
    uy/=un;
    uz/=un;

    // Compute an initial guess by using (ux,uy,uz)
    // extended to range on desired side.
    // If side = 1, then point to starboard, if -1 then point to port
    double srange =
    sqrt(range*range−pow(sqrt(x*x+y*y+z*z)−majaxis/2.0−minaxis/2.0,2.0));
    xt=x−side*srange*ux;// This will be the initial guess of the x coordinate
    yt=y−side*srange*uy;// This will be the initial guess of the y coordinate
    zt=z−side*srange*uz;// This will be the initial guess of the z coordinate

    // Improve the initial guess
    double shrink = (majaxis+minaxis)/2.0/sqrt(xt*xt+yt*yt+zt*zt);
    xt*=shrink;
    yt*=shrink;
    zt*=shrink;

    int icount = 0;// counter variable
    double tol = 0.1; // Tolerance of convergence
    // Compute the position vector using Newton Method
    while((advance(&xt,&yt,&zt)>tol) & (icount++<100)) {/*Do nothing*/};

    // Test for convvergence
    if(icount=100)
    {
        cout "Error computing coodinate" << endl;
        return 0;
    }

    // Return 1 for starboard coordinate, -1 for port coordinate
    return((x*(vz*(yt−y)−vy*(zt−z))
            +y*(vx*(zt−t)−vz*(xt−x))
            +z*(vy*(xt−x)−vx*(yt−y)))>0.0)? 1 : −1;
}
```