GPS, Conditioning and Non-Linear Least Squares

By Matthew Fried and Zeyu Wang

Introduction

Solving for GPS coordinates requires the use of several important numerical methods. Below we use Multivariate Newton's Method and the Gauss-Newton Method to solve for different variations of coordinates. We initially start with four satellites and find the intersection of these, leaving out the secondary extraneous point. The below problems follow the questions as outlined in Sauer, with both Python and Matlab calculations. Each problem below discusses the specific methodology and experiment.

Methodology and Experiments

1.) We solved the problem using Multivariate Newton's Method (as was instructed). We used the formula presented first in section 2.7: $x_{k+1} = x_k - DF(x_k)^{-1}*F(x_k)$ (which in matlab this *should* be done using A\b instead of $A^{-1}b$, it is left here and throughout for mathematical clarity, not coding purposes). The Multivariate Newton formula is the first non-linear system of equations approach mentioned in the book. It is similar to Newton's Method (in the linear case) insofar as it is also derived from Taylor's theorem. Newton's method is: $x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$. Taylor's theorem says that when we solve for when y = 0, we have: $0 = f'(x_n)(x_{n+1} - x_n) + f(x_n).$, which, reorganized, is both the above formula and the equation for a line. The Taylor expansion for vector-valued functions (or a non-linear case) is $F(x) = F(x_0) + DF(x_0) \cdot (x - x_0)$, using the same reorganization for the regular case and recognizing that the D is the Jacobian, we get: $x_{k+1} = x_k - (DF(x_k))^{-1}F(x_k)$. While solving for an inverse may be difficult, the equation can be reorganized to solve the system of equations: $J_F(x_n)(x_{n+1} - x_n) = -F(x_n)$ as such.
This method has quadratic convergence and is quite simple and fast to implement. There are several successive-relaxation techniques that are applied to Newton's method due to its tendency to overshoot (and due to the slower convergence for cases of multiplicity greater than 1).
We solved the problem in python and matlab - comparing approaches and results. Below is the matlab code and the link for the python code is here. The python code used numpy libraries such as np.linalg.lstsq, which solves for least squares, while the matlab code solved the problems without the use of libraries.

```
init_vec = [0; 0; 6370; 0];

for i = 1:100

   x = init_vec(1,1);
   y = init_vec(2,1);
   z = init_vec(3,1);
   d = init_vec(4,1);
   c = 299792.458;

   f = [((x - 15600)^2 + (y - 7540)^2 + (z - 20140)^2 - (c*(.07074 - d))^2);
      ((x - 18760)^2 + (y - 2750)^2 + (z - 18610)^2 - (c*(.0722 - d))^2);
      ((x - 17610)^2 + (y - 14630)^2 + (z - 13480)^2 - (c*(.0769 - d))^2);
      ((x - 19170)^2 + (y - 610)^2 + (z - 18390)^2 - (c*(.07242 - d))^2)];

   Jacobian = [(x - 15600), (y - 7540), (z - 20140), c^2*(.07074 - d);
            (x - 18760), (y - 2750), (z - 18610), c^2*(.0722 - d);
            (x - 17610), (y - 14630), (z - 13480), c^2*(.0769 - d);
            (x - 19170), (y - 610), (z - 18390), c^2*(.07242 - d)];

   Jacobian = 2*Jacobian;
   init_vec = init_vec -Jacobian\f;

end

Init_vec
```

<u>Output:</u>

```
   1.0e+03 *

  -0.041772709570849
  -0.016789194106530
   6.370059559223343
  -0.000003201565830
```

We see that this method is quite effective in finding the GPS coordinate. It seems that our matlab approach was quite successful, as well as the Python code.

3.) We solved the same problem using the syms and solve functions. The process was straightforward. The optimized matlab output returned both the answer for an earth-bound solution and a (far-away) space bound solution. Below is the code and output:

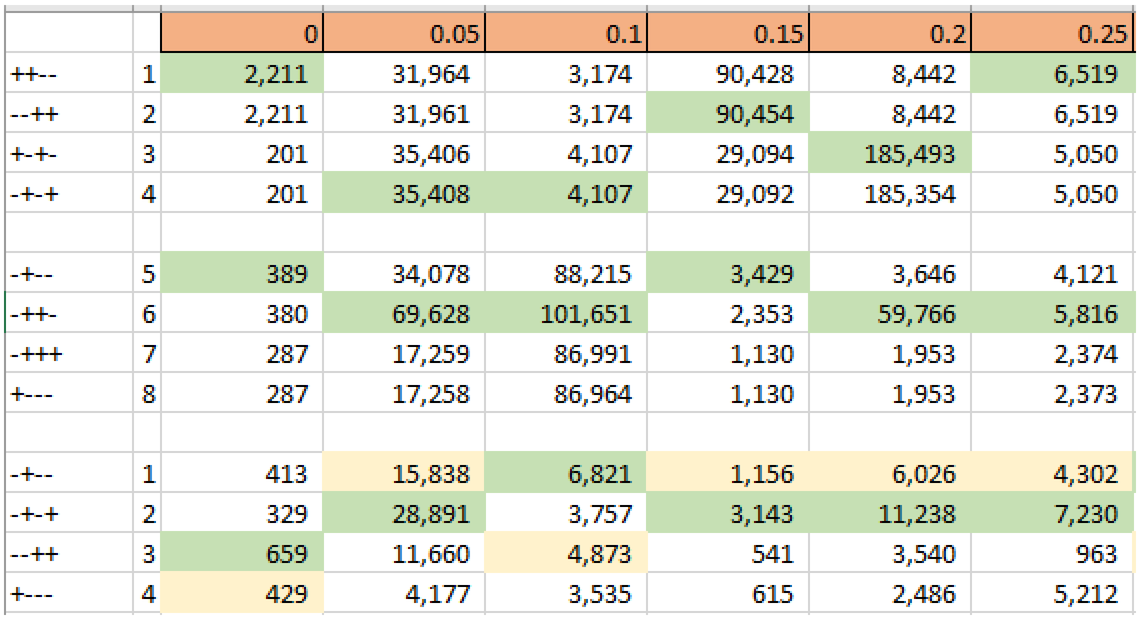
```
syms x y z d
c = 299792.458;
eq1 = (x - 15600)^2 + (y - 7540)^2 + (z - 20140)^2 - (c*(.07074 - d))^2 == 0;
eq2 = (x - 18760)^2 + (y - 2750)^2 + (z - 18610)^2 - (c*(.0722 - d))^2 == 0;
eq3 = (x - 17610)^2 + (y - 14630)^2 + (z - 13480)^2 - (c*(.0769 - d))^2 == 0;
eq4 = (x - 19170)^2 + (y - 610)^2 + (z - 18390)^2 - (c*(.07242 - d))^2 == 0 ;

sol = solve([eq1,eq2,eq3,eq4],[x,y,z,d]);
solution = vpa([sol.x; sol.y; sol.z; sol.d]);
solution
```

<u>Output:</u>

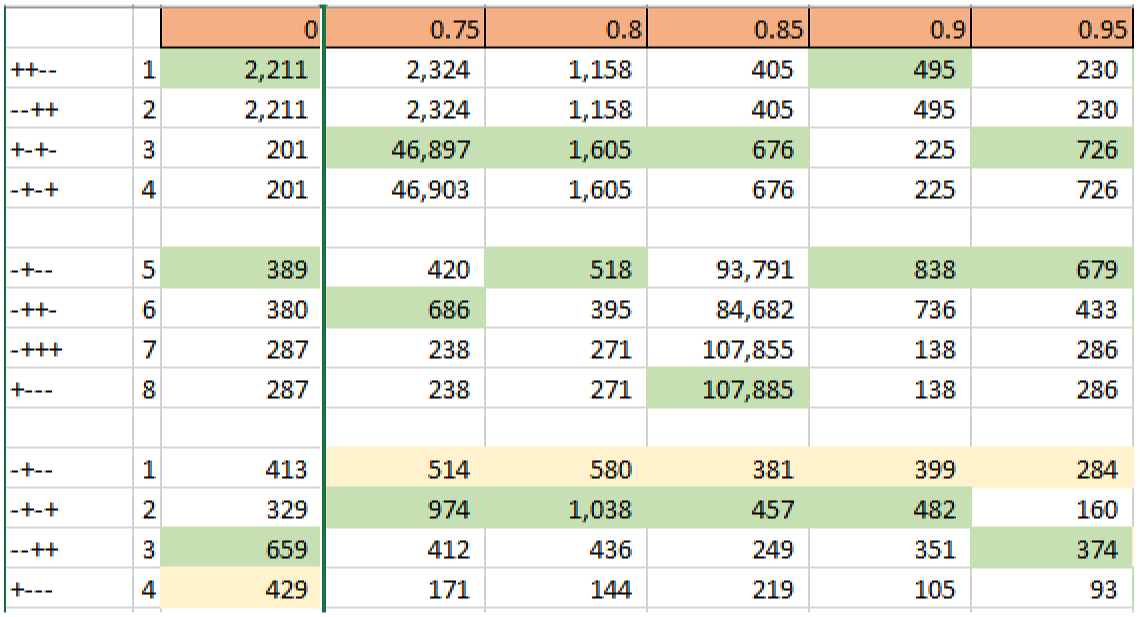
```
 -41.772709570817240628596764262 77
-39.747837348220756696709315336828
-16.789194106518448860393898727409
-134.27414436068348100885939770666
 6370.0595592233524366817718914811
-9413.6245537358225775255225643754
-0.0032015658295940652841373399815865
0.18517304709594597102024833371776
```

4/5.) For this problem there was no more math involved than using the above Multivariate Newton method and then calculating the error using: $\text{error magnification factor} = \dfrac{||(\Delta x, \Delta y, \Delta z)||_\infty}{c||(\Delta t_1, \ldots, \Delta t_m)||_\infty}$.
Our results include several important points.

- Firstly, we note that when the satellites are within a short range of each other the error is significantly greater.
- Secondly, we note that for the error possibilities as discussed in the video either +-+-, -+-+ resulted in more error than ++--, or --++. And -++- resulted in greater error than when it is a 3:1 plus/minus (minus/plus) scenario.
- The only thing we seem to have found is that, in general, having more variation in the error resulted in a greater error magnification. (Specifically, -+-- and +--- have consistent errors which lead to less overall error than -+-+).



		0	0.05	0.1	0.15	0.2	0.25
++--	1	2,211	31,964	3,174	90,428	8,442	6,519
--++	2	2,211	31,961	3,174	90,454	8,442	6,519
+-+-	3	201	35,406	4,107	29,094	185,493	5,050
-+-+	4	201	35,408	4,107	29,092	185,354	5,050
-+--	5	389	34,078	88,215	3,429	3,646	4,121
-++-	6	380	69,628	101,651	2,353	59,766	5,816
-+++	7	287	17,259	86,991	1,130	1,953	2,374
+---	8	287	17,258	86,964	1,130	1,953	2,373
-+--	1	413	15,838	6,821	1,156	6,026	4,302
-+-+	2	329	28,891	3,757	3,143	11,238	7,230
--++	3	659	11,660	4,873	541	3,540	963
+---	4	429	4,177	3,535	615	2,486	5,212



		0	0.75	0.8	0.85	0.9	0.95
++--	1	2,211	2,324	1,158	405	495	230
--++	2	2,211	2,324	1,158	405	495	230
+-+-	3	201	46,897	1,605	676	225	726
-+-+	4	201	46,903	1,605	676	225	726
-+--	5	389	420	518	93,791	838	679
-++-	6	380	686	395	84,682	736	433
-+++	7	287	238	271	107,855	138	286
+---	8	287	238	271	107,885	138	286
-+--	1	413	514	580	381	399	284
-+-+	2	329	974	1,038	457	482	160
--++	3	659	412	436	249	351	374
+---	4	429	171	144	219	105	93

6.) We used the Gauss-Newton method (as instructed), as described in the book

$$A^T A v^k = -A^T r(x^k)$$

$$x^{k+1} = x^k + v^k$$

, with A equal to the Jacobian. This method is derived by finding the left pseudoinverse of the matrix and solving, as we would in any case least squares case where the direct inverse doesn't exist: $(\mathbf{J_f}^{\mathrm{T}}\mathbf{J_f})^{-1}\mathbf{J_f}^{\mathrm{T}}$. We set 8 cases to test for error (different variations of adding and subtracting the error, however, regardless of the variation of error, it is clear that adding more satellites does a significant job of reducing any error.

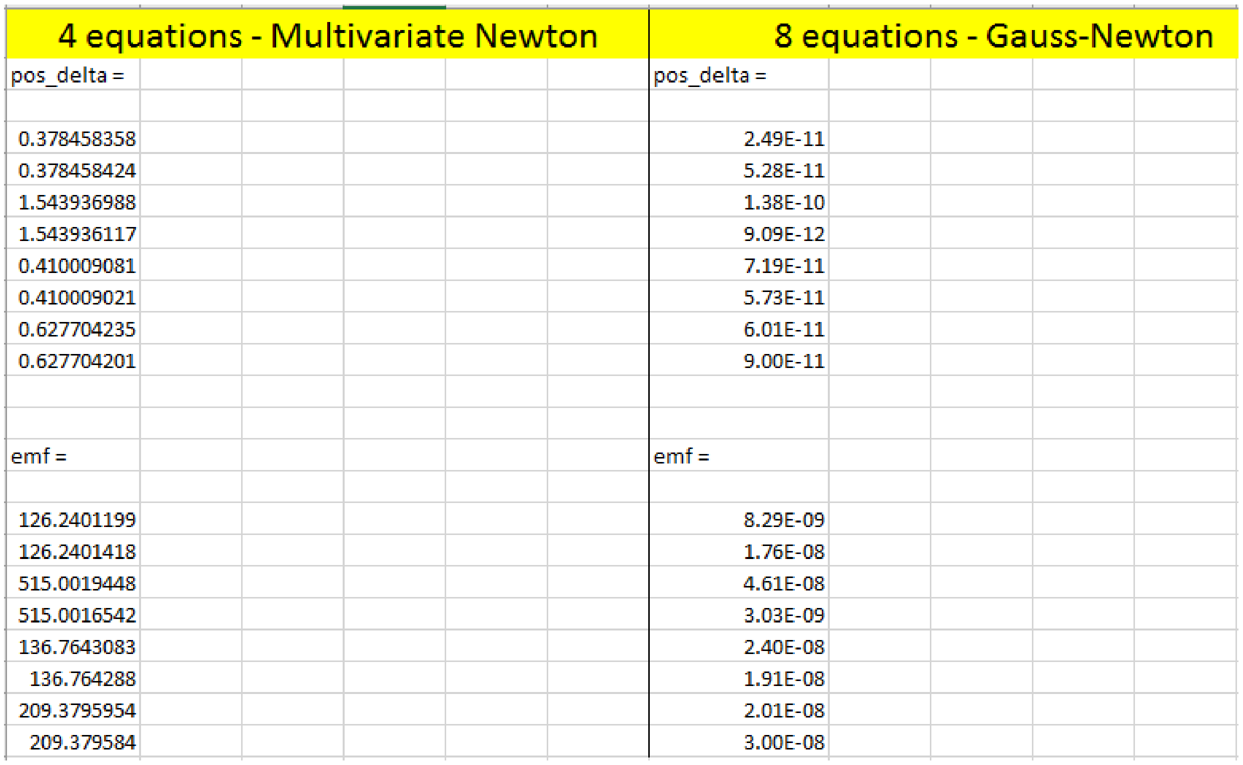
We also tested for different initial vectors (see below). It is obviously best to have a reasonable guess and choose a vector within range. We found that the 4 satellite Multivariate Newton method had significant error in every case but when the initial vector was close to the actual solution. However, the 8 satellite case was almost indifferent as to which initial vector was chosen.

Including the bunched_4, unbunched_4, and unbunched_8, it is clear that the unbunched_8 has error rates which are almost negligible, the unbunched_4 has error rates which are only insignificant when an optimal initial vector is chosen, and the bunched_4 has quite significant error rates, even with an excellent choice for the initial vector.

While the unbunched_8 is already excellent in terms of convergence, if it were to be an ill-conditioned matrix we could use lambda to accentuate the effect of the diagonal by adding a term λ*diag(A^TA) to the right hand side of $A^TAv^k = -A^Tr(x^k)$. This is the Levengberg-Marquardt method.

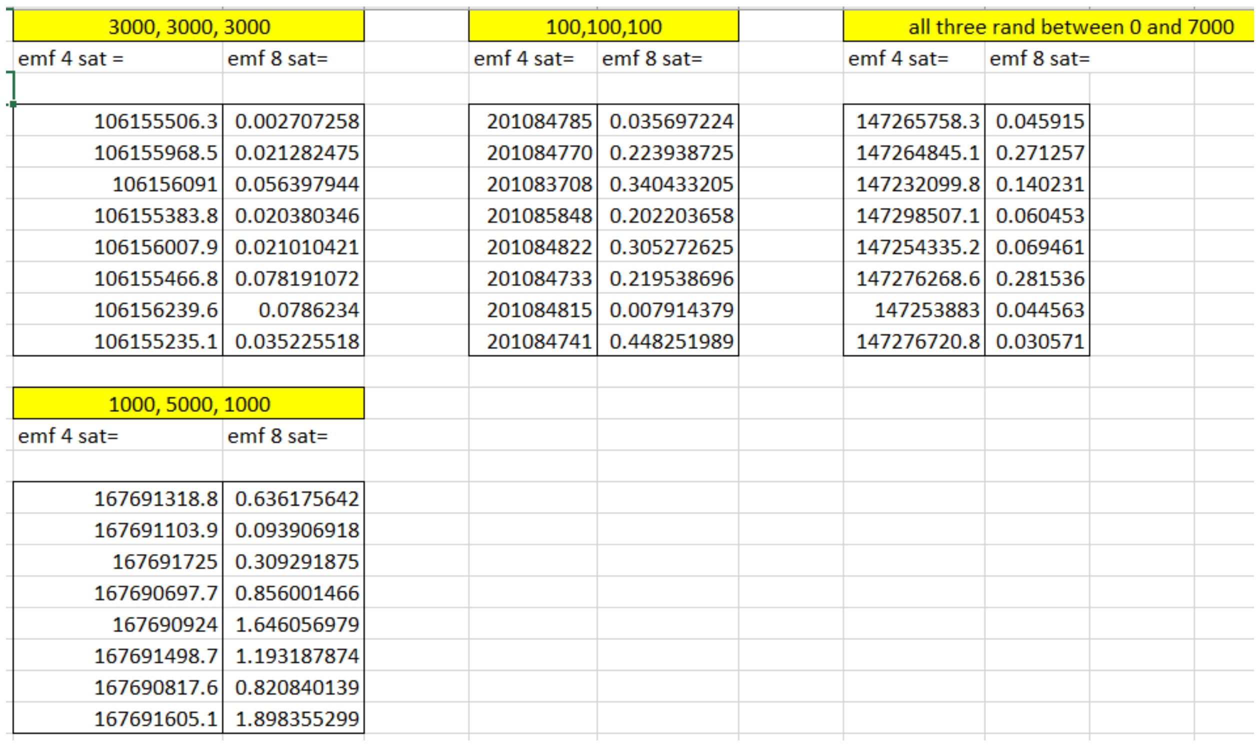
We found this entire project quite surprising in terms of the significance of the error under such slight variations.

Output for the two different cases of satellites.



4 equations - Multivariate Newton	8 equations - Gauss-Newton
pos_delta =	pos_delta =
0.378458358	2.49E-11
0.378458424	5.28E-11
1.543936988	1.38E-10
1.543936117	9.09E-12
0.410009081	7.19E-11
0.410009021	5.73E-11
0.627704235	6.01E-11
0.627704201	9.00E-11
emf =	emf =
126.2401199	8.29E-09
126.2401418	1.76E-08
515.0019448	4.61E-08
515.0016542	3.03E-09
136.7643083	2.40E-08
136.764288	1.91E-08
209.3795954	2.01E-08
209.379584	3.00E-08

Different cases for the initial vector:



3000, 3000, 3000	
emf 4 sat =	emf 8 sat=
106155506.3	0.002707258
106155968.5	0.021282475
106156091	0.056397944
106155383.8	0.020380346
106156007.9	0.021010421
106155466.8	0.078191072
106156239.6	0.0786234
106155235.1	0.035225518

100,100,100	
emf 4 sat=	emf 8 sat=
201084785	0.035697224
201084770	0.223938725
201083708	0.340433205
201085848	0.202203658
201084822	0.305272625
201084733	0.219538696
201084815	0.007914379
201084741	0.448251989

all three rand between 0 and 7000	
emf 4 sat=	emf 8 sat=
147265758.3	0.045915
147264845.1	0.271257
147232099.8	0.140231
147298507.1	0.060453
147254335.2	0.069461
147276268.6	0.281536
147253883	0.044563
147276720.8	0.030571

1000, 5000, 1000	
emf 4 sat=	emf 8 sat=
167691318.8	0.636175642
167691103.9	0.093906918
167691725	0.309291875
167690697.7	0.856001466
167690924	1.646056979
167691498.7	1.193187874
167690817.6	0.820840139
167691605.1	1.898355299

Conclusions

Overall, it seems clear that adding more satellites will give better precision, and it is mostly a trade-off with efficiency. It is quite interesting to see the different methods applied, and how straightforward it is to calculate different real-world possibilities.

As an addendum, Zeyu wrote the Python code (solving problem 1), while Matthew wrote the Matlab code (solving the other problems as well). We did not use any references outside the recommended course book.