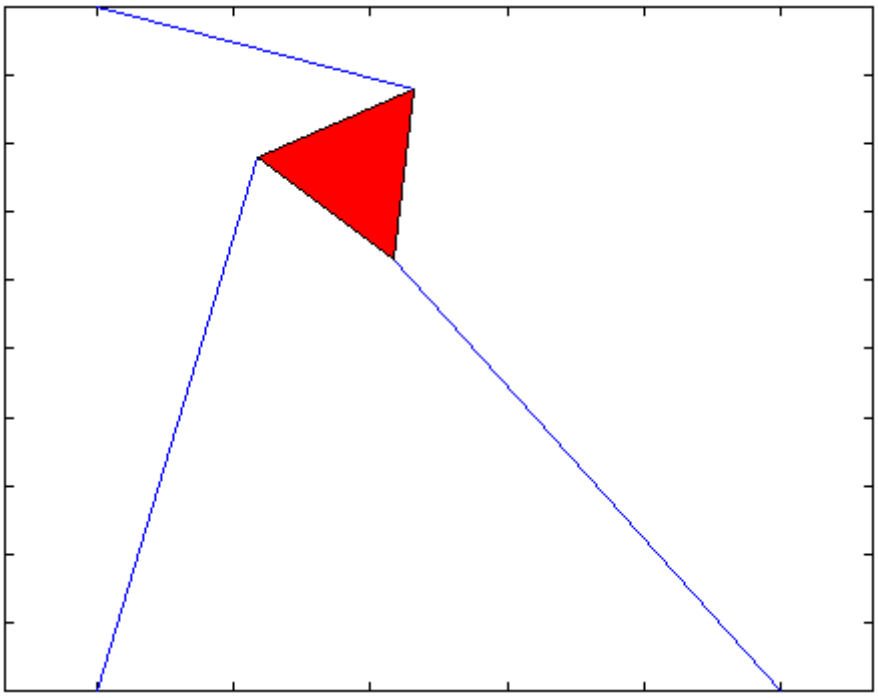
POLYNOMIAL SYSTEMS

FINAL PROJECT



Presented by:

Elias Cohenca

302025358

PROBLEM DESCRIPTION

The problem presented is akin to a simplified 2D Stewart platform. A triangular robot is attached by three actuator arms to a platform (the X-Y plane), and the lengths of the actuators can be changed to reach a desired position.

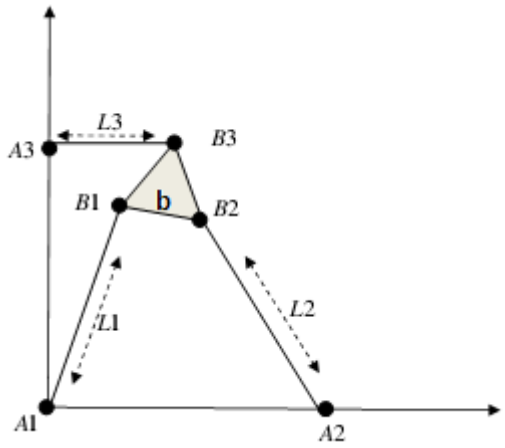


FIGURE 1

The solution being sought is, given the lengths of L1, L2 and L3, what would be the final position of the robot? That is, what are the final coordinates of B1, B2 and B3?

The data given is:

$A_1 = (0,0)$

$A_2 = (1,0)$

$A_3 = (0,1)$

$b = 0.25$ (length of equilateral triangle's side)

$L_i = \|A_i - B_i\| = [0.25,1]$ (actuator's range)

PROBLEM ANALYSIS AND ANALYTICAL SOLUTION

It is required to form a system of polynomial equations that describe said system in order to find the final coordinates. An initial look at the problem yields the following equations that must be always satisfied:

$$(1)\begin{cases} \|B_1 - A_1\|^2 = L_1^2 \\ \|B_2 - A_2\|^2 = L_2^2 \\ \|B_3 - A_3\|^2 = L_3^2 \\ \|B_2 - B_1\|^2 = b^2 \\ \|B_3 - B_1\|^2 = b^2 \\ \|B_3 - B_2\|^2 = b^2 \end{cases}$$

The last three equations are due to the rigidity of the triangle.

This system has six equations and six variables (the x-y coordinates for each B_i). It is solvable, but it is possible to do much better. We can reduce the system to three variables by noting that two coordinates of the triangle can be written in terms of the remaining one by using the angle at which the triangle is rotated. The following figure illustrates this:

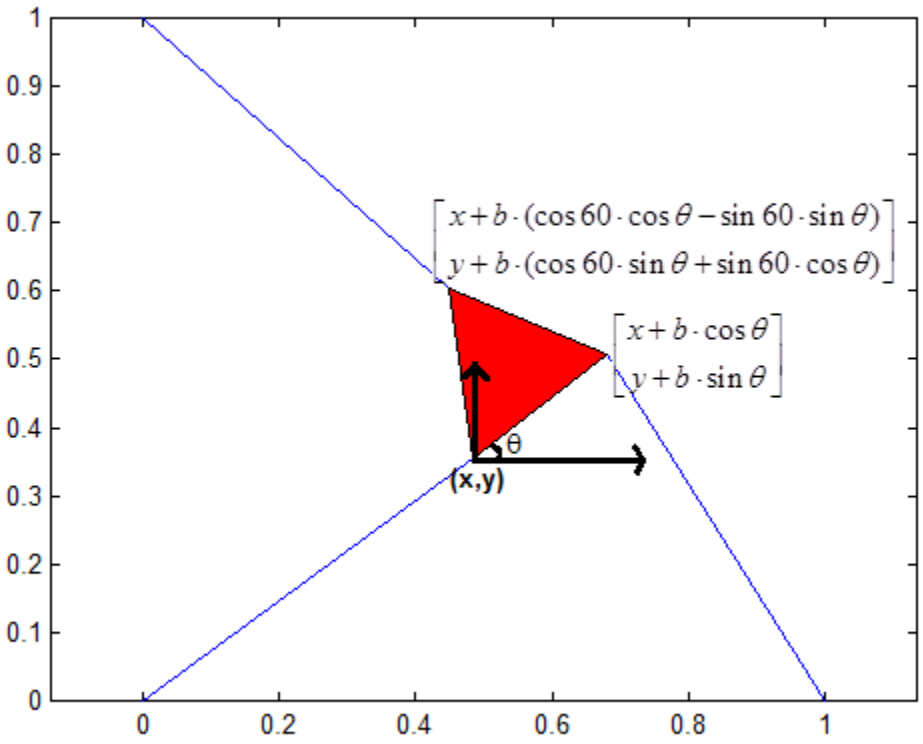


FIGURE 2

Instead of the three pair of coordinates we now have to find only x, y and θ (the rotation angle of the triangle)

$$B_1 = \begin{bmatrix} x \\ y \end{bmatrix}$$

$$B_2 = \begin{bmatrix} x + b \cdot \cos\theta \\ y + b \cdot \sin\theta \end{bmatrix}$$

$$B_3 = \begin{bmatrix} x + b \cdot \cos(60 + \theta) \\ y + b \cdot \sin(60 + \theta) \end{bmatrix} = \begin{bmatrix} x + b \cdot (\cos 60 \cdot \cos\theta - \sin 60 \cdot \sin\theta) \\ y + b \cdot (\cos 60 \cdot \sin\theta + \sin 60 \cdot \cos\theta) \end{bmatrix}$$

In other words, we have taken the last three equations in (1) and found the relationship between them. It is now possible to insert them into the first three equations, to get a system of three variables and three equations. Since the terms become very large very quickly, I have used the program Mathematica to simplify and solve the equations. The Mathematica file will be attached to the report. I will go through the steps here.

First define some variables to keep the equations shorter:

b:= $\frac{1}{4}$ (the side of the triangle)

a_1:=1 (distance on x axis from A_2 to A_1)

a_2:=1 (distance on y axis from A_3 to A_1)

γ:=Pi / 3 (internal angle of the triangle)

Variables that come from the B_i definitions mentioned above:

$$\text{eqs:=}\left\{\begin{array}{l} \text{a2} \to b\text{Cos}[\theta] - a_1 \\ \text{b2} \to b\text{Sin}[\theta] \\ \text{a3} \to b(\text{Cos}[\theta]\text{Cos}[\gamma] - \text{Sin}[\theta]\text{Sin}[\gamma]) \\ \text{b3} \to b(\text{Cos}[\theta]\text{Sin}[\gamma] + \text{Sin}[\theta]\text{Cos}[\gamma]) - a_2 \end{array}\right\}$$

Now the result of inserting said definitions into the first three equations in (1):

$$(2)\left\{\begin{array}{l} \text{p1:=} x^2 + y^2 \to l_1^{\,2} \\ \text{p2:=}(x + \text{a2})^2 + (y + \text{b2})^2 - l_2^{\,2} == 0 \\ \text{p3:=}(x + \text{a3})^2 + (y + \text{b3})^2 - l_3^{\,2} == 0 \end{array}\right.$$

Expanding p2 and p3 yields, and replacing with p1 for the quadratic terms:

$$\text{p2:= a2}^2 + \text{b2}^2 + 2 \cdot \text{a2} \cdot x + 2 \cdot \text{b2} \cdot y + l_1^2 - l_2^2 == 0$$

$$\text{p3:= a3}^2 + \text{b3}^2 + 2 \cdot \text{a3} \cdot x + 2 \cdot \text{b3} \cdot y + l_1^2 - l_3^2 == 0$$

This system of two equations can now be solved to get x and y in terms of θ:

$$(3)\left\{\begin{array}{l} x(\theta)\to -\dfrac{-2\cdot \mathrm{b3}\cdot\left(\mathrm{a2}^2+\mathrm{b2}^2+l_1^2-l_2^2\right)+2\cdot \mathrm{b2}\cdot\left(\mathrm{a3}^2+\mathrm{b3}^2+l_1^2-l_3^2\right)}{4\cdot \mathrm{a3}\cdot \mathrm{b2}-4\cdot \mathrm{a2}\cdot \mathrm{b3}}, \\ y(\theta)\to \dfrac{\mathrm{a2}^2\cdot \mathrm{a3}-\mathrm{a2}\cdot \mathrm{a3}^2+\mathrm{a3}\cdot \mathrm{b2}^2-\mathrm{a2}\cdot \mathrm{b3}^2+(-\mathrm{a2}+\mathrm{a3})\cdot l_1^2-\mathrm{a3}\cdot l_2^2+\mathrm{a2}\cdot l_3^2}{-2\cdot \mathrm{a3}\cdot \mathrm{b2}+2\cdot \mathrm{a2}\cdot \mathrm{b3}} \end{array}\right\}$$

These two equations are now reinserted into the first equation in (2). This will give us an equation that is only dependent on θ (albeit a non-linear one since the angles are inside trigonometric functions). Since we are interested in finding the roots of this equation, we can set it to zero and look only at the numerator:

$\mathrm{n1}:=\mathrm{Numerator}[x]$

$\mathrm{n2}:=\mathrm{Numerator}[y]$

$d:=\mathrm{Denominator}[x]$

$$f := \mathrm{n1}^2+\mathrm{n2}^2-l_1^2 d^2 == 0$$

$$f := -4(a3\cdot b2-a2\cdot b3)^2 l_1^2+(-a2^2\cdot a3+a2\cdot a3^2-a3\cdot b2^2+a2\cdot b3^2+a2\cdot l_1^2-a3\cdot l_1^2+a3\cdot l_2^2-a2\cdot l_3^2)^2+$$
$$(-a3^2\cdot b2+a2^2\cdot b3+b2^2\cdot b3-b2\cdot b3^2-b2\cdot l_1^2+b3\cdot l_1^2-b3\cdot l_2^2+b2\cdot l_3^2)^2 == 0$$

To turn this equation into a polynomial equation we need to get rid of the trigonometric functions. We will use Weierstrass' substitution:

$$\mathrm{Cos}[\theta]\to\frac{\left(1-t^2\right)}{\left(1+t^2\right)},\quad \mathrm{Sin}[\theta]\to\frac{2t}{\left(1+t^2\right)},\quad \mathrm{Tan}\left[\frac{\theta}{2}\right]\to t$$

This will yield a sixth degree polynomial on *t*. After some manipulations the coefficients are found. Here they are, ordered from lowest to highest:

$$(4)\left\{\begin{array}{c} 2000.72+5650.38l_1^4+2578.38l_2^4-3332.68l_3^2+2304.l_3^4+l_1^2\left(1059.45-5924.76l_2^2-5376.l_3^2\right)+l_2^2\left(-3384.12+768.l_3^2\right) \\ -2120.03-1499.24l_1^4-2048.l_2^4+1995.32l_3^2+l_1^2\left(475.9+3547.24l_2^2-548.76l_3^2\right)+l_2^2\left(2342.56+548.76l_3^2\right) \\ 14714.4+22546.4l_1^4+11282.4l_2^4-18761.4l_3^2+11008.l_3^4+l_1^2\left(1201.4-22820.8l_2^2-22272.l_3^2\right)+l_2^2\left(-18401.3+256.l_3^2\right) \\ -10313.-2998.48l_1^4-4096.l_2^4+7537.89l_3^2+l_1^2\left(2451.04+7094.48l_2^2-1097.52l_3^2\right)+l_2^2\left(7281.89+1097.52l_3^2\right) \\ 34895.7+28141.6l_1^4+14829.6l_2^4-32571.3l_3^2+15104.l_3^4+l_1^2\left(-3372.31-27867.2l_2^2-28416.l_3^2\right)+l_2^2\left(-32245.5-1792.l_3^2\right) \\ -12289.-1499.24l_1^4-2048.l_2^4+5542.56l_3^2+l_1^2\left(1975.14+3547.24l_2^2-548.76l_3^2\right)+l_2^2\left(4939.32+548.76l_3^2\right) \\ 26278.1+11245.6l_1^4+6125.62l_2^4-17142.6l_3^2+6400.l_3^4+l_1^2\left(-3514.26-10971.2l_2^2-11520.l_3^2\right)+l_2^2\left(-17228.3-1280.l_3^2\right) \end{array}\right\}$$

Since the degree of the polynomial is six, there are at most six distinct solutions. For each *t* root found, we can find the corresponding θ using the relation:

$$\theta = 2\arctan(t)$$

Inserting this angle into the equations of (3) will give the corresponding *x* and *y* values, and having these three values we can find all the B_i coordinates.

Another method considered to solve the problem is to look at the origin of each arm as the origin of a circle, with a radius equal to the length of the arm, and to look at the circle circumscribing the triangle. The set of points containing the intersections of the four circles contains the location of the vertices of the triangle. This method was discarded since it doesn't simplify the problem more than the solution implemented.

POLYNOMIAL SOLUTION

The method selected to find the roots was a combination of Sturm sequences to find the intervals where the roots are, and Newton-Raphson method to hone into the roots when an interval with only one root is found.

Sturm's theorem expresses the number of distinct real roots of *p* located in an interval in terms of the number of changes of signs of the values of the Sturm's sequence at the bounds of the interval. Applied to the interval of all the real numbers, it gives the total number of real roots of *p*.[1]

The idea of Newton-Raphson is as follows: one starts with an initial guess which is reasonably close to the true root, then the function is approximated by its tangent line (the negative of the derivative at that point), and one computes the x-intercept of this tangent line. This x-intercept will typically be a better approximation to the function's root than the original guess, and the method can be iterated. [2] That is, the root after *n* steps is:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

The algorithm used then is:

1. Get initial bounds for the roots of the polynom
2. Call RootIsolation with these bounds

RootIsolation(input: interval):

1. Divide the interval in half
2. If number of roots in first half is 1 run NR starting at the middle of this interval
    a. If it's more than one call RootIsolation with this half of the interval
3. If number of roots in second half is 1 run NR starting at the middle of this interval
    a. If it's more than one call RootIsolation with this half of the interval

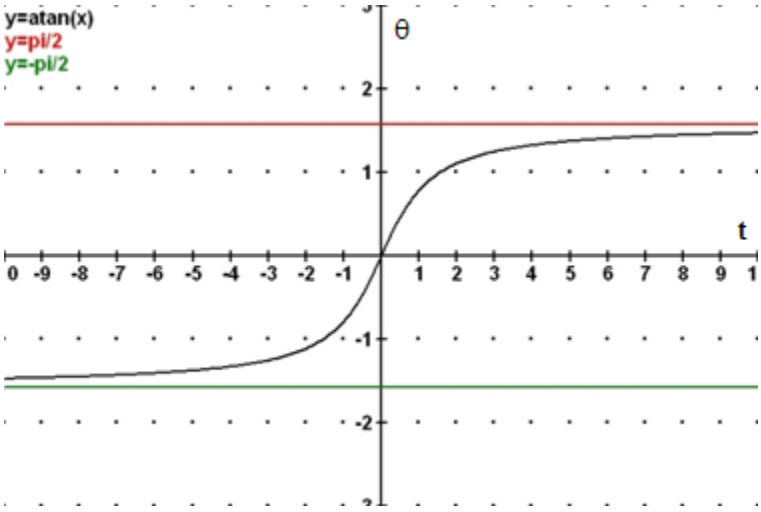
The NR method also looks that the root found is always inside the bounds being looked at. This is to prevent finding the root belonging to other interval, and to prevent the algorithm from diverging. In case the root goes out of bound, it will return control to the RootIsolation function, which will continue halving the interval.

---

[1] http://en.wikipedia.org/wiki/Sturm's_theorem
[2] http://en.wikipedia.org/wiki/Newton's_method

NUMERICAL ISSUES

When selecting the bounds of the roots care must be taken since *t* and the half-angle are related by *arctan*. This means that if the angle is bigger than π/2 degrees (half-angle is π/4, when tan(x)=1), the value of *t* starts growing much faster, to the point where π degrees is at infinity (half-angle of π/2, when tan(x)=∞). This will make the Sturm algorithm much slower, since the initial bounds will be quite large.



To overcome this problem we can use the first polynomial to look only in the area between $-\pi/2 < \theta < \pi/2$. That means the initial bounds to look at will be only $-1 < t < 1$. To find solutions that might be between $\pi/2 < \theta < 3\pi/2$ we can define a different angle of $\tilde{\theta} = \theta - \pi$, that is, just select a different reference axis. Redoing all the calculations yields a second polynomial whose coefficients are the same as the first one, only the order of the coefficients is reversed, and the odd powers have inverted signs (Mathematica file also attached). In this case we also use the boundaries of $-1 < t < 1$, but the answer at the end will be for $\tilde{\theta}$, so to find the angle for θ we just need to add π back.

PROGRAM DESCRIPTION

The program consists of a main function that can receive input either from the command line or user input. It then calls **GetForwardKinematics** which is the main solver function.

I made some changes to the given envelope to allow easier testing of various cases. I added to the input a number to be added to the solution filename. This is used in a batch file to make several runs at once and keep all the solutions files. The batch file is also attached to the program.

Furthermore because the running times vary so much depending on varying factors, I ran the solution for 1000 iterations and found the average running time. I assume this is more accurate and statistically important.

GetForwardKinematics is the main function which receives the input and returns the solutions found. Since a maximum of six solutions can be found, all arrays are statically initialized to this number. This is much better than using *malloc*, trading memory for speed (and not that much memory for the sizes we are dealing with).

A structure is used to keep the polynomial and its degree

```
typedef struct
{
  short deg;   // Actual degree of polynom (could be less than 6)
  double c[MAXDEG+1];// Coefficients of polynom of degree 6 (ordered from low to high)
} poly;
```

First the function **Build2Polynoms** builds the coefficient list for both polynomials, according to the given actuator lengths. Then for each polynomial the function **FindRootsInterval** finds the roots according to the algorithm described above. It uses the function **Sturm** to find the Sequence for each half of the interval. If one root only is found it calls **NewtonRaphson**, otherwise it keeps subdividing the interval. The same is done for the second half of the interval. This way intervals that don't have any roots are immediately discarded.

The **NewtonRaphson** function uses the NR algorithm, but returns FALSE if it goes out of bounds or exceeds the maximum allowed number of iterations. If **FindRootsInterval** receives FALSE it continues subdividing, as explained above.

Finally when the roots are found, the function **GetXYCoords** calculates the final B_i coordinates.

All helper functions that are small enough and not used recursively are declared inline in AuxFunctions.h for a faster runtime.

The trigonometric functions are some of the heaviest functions. Care was taken to minimize their use. Only one call was used in the end to find cosθ when finding the final coordinates. The sin of the same angle was found using the property $\cos^2\theta+\sin^2\theta=1\rightarrow\sin\theta=\pm\sqrt{1-\cos^2\theta}$. Finding the root is computationally cheaper.

A fast arc tangent approximation function was also tried, but its precision was only good for ε=0.01 and so the gains were too low to use.

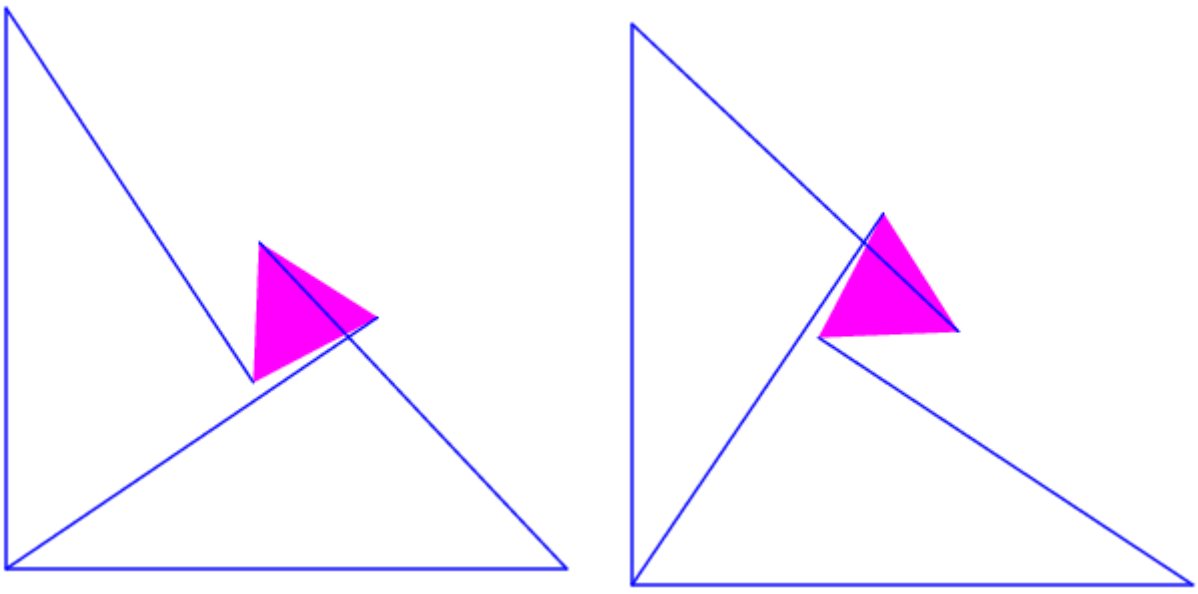
There is also the issue of the required tolerance (eps). Since the tolerance required is checked on the actuator lengths, it must be somehow translated to the realms of *t*, where the roots are found (and the error lies). The following heuristic was used:

Since between -1 and 1 the arctan is almost linear, we can approximate and say $\arctan\theta=t\Rightarrow\theta\cong t$ and so the tolerance for t is the same as the tolerance for the angle. Now the tolerance for the angle must be found. Looking at Figure 2, the error on the position of the B's can be approximated by $\varepsilon=d\theta\cdot b$ (approximation by arc length) and so the maximum angle deviation desired would be $d\theta=\varepsilon/b$.

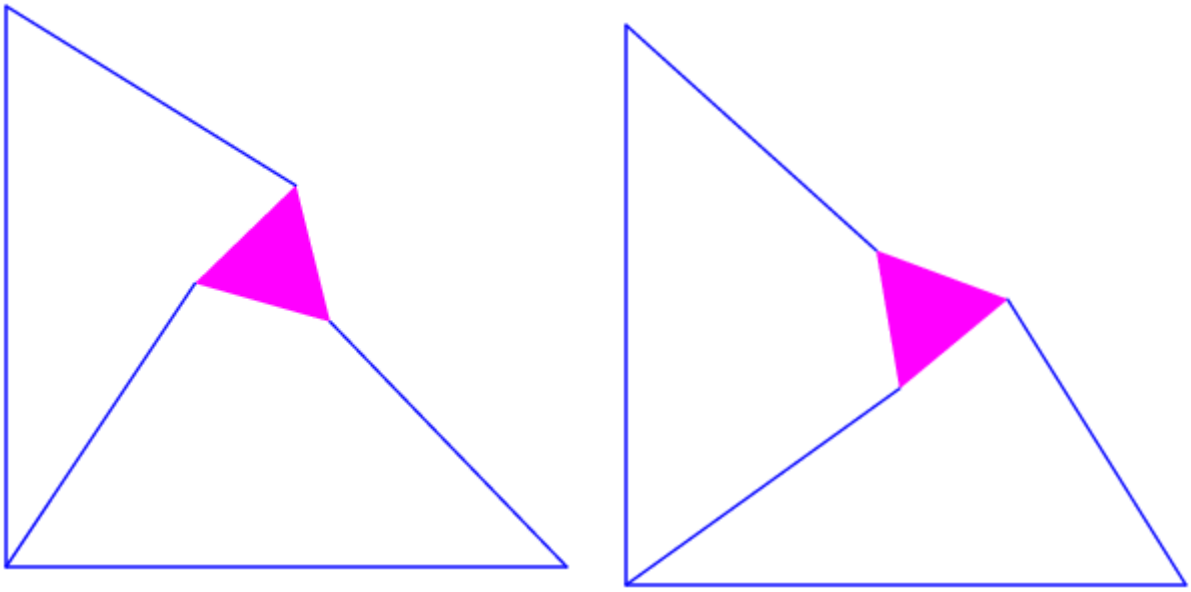
SAMPLES

Running times are on a Core 2 Duo 2.93 Ghz with 3 GB of RAM

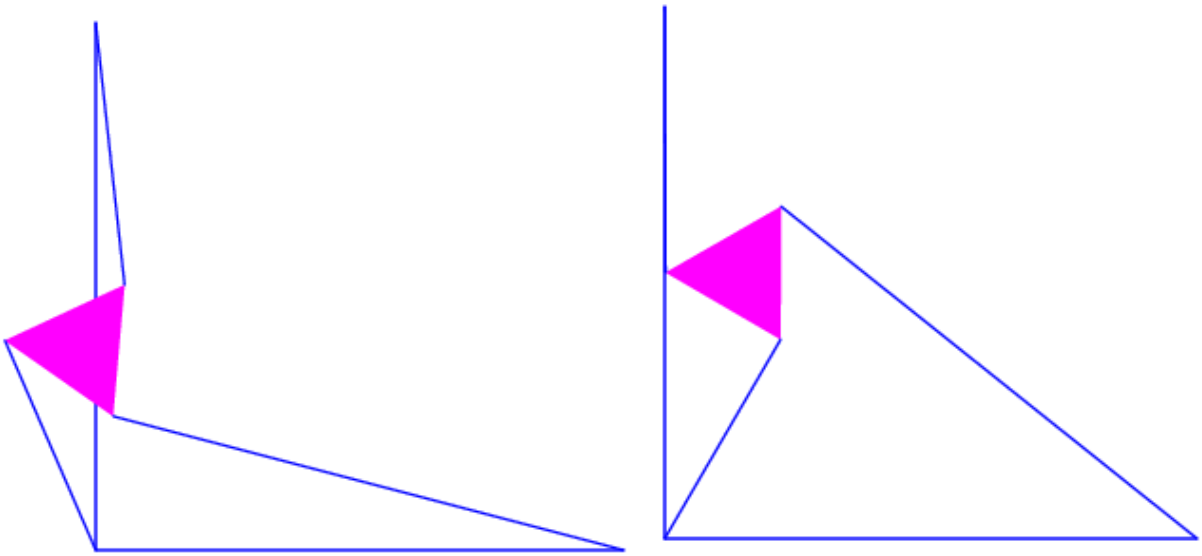
```
Elapsed milli seconds for 1000 run(s): 3.788301
Average per run milli seconds: 0.003788
Ran for L1=0.800, L2=0.800, L3=0.800. Eps=0.01000
||B1-A1||-L1 = -0.000001
||B2-A2||-L2 = -0.000001
||B3-A3||-L3 = -0.000001
||B1-B2||-0.25 = -0.000000
||B2-B3||-0.25 = -0.000000
||B3-B1||-0.25 = -0.000000
||B1-A1||-L1 = -0.000001
||B2-A2||-L2 = -0.000001
||B3-A3||-L3 = -0.000001
||B1-B2||-0.25 = 0.000000
||B2-B3||-0.25 = -0.000000
||B3-B1||-0.25 = -0.000000
```



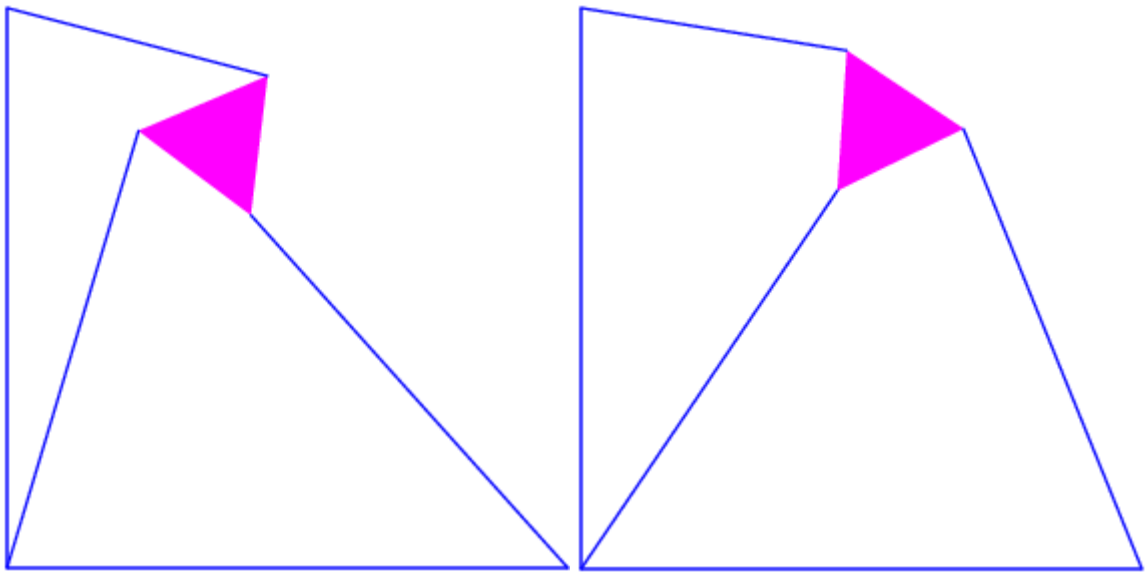
```
Elapsed milli seconds for 1000 run(s): 5.269480
Average per run milli seconds: 0.005269
Ran for L1=0.600, L2=0.600, L3=0.600. Eps=0.01000
||B1-A1||-L1 = 0.007993
||B2-A2||-L2 = 0.007993
||B3-A3||-L3 = 0.007993
||B1-B2||-0.25 = 0.000000
||B2-B3||-0.25 = -0.000000
||B3-B1||-0.25 = -0.000000
||B1-A1||-L1 = 0.001594
||B2-A2||-L2 = 0.001594
||B3-A3||-L3 = 0.001594
||B1-B2||-0.25 = -0.000000
||B2-B3||-0.25 = -0.000000
||B3-B1||-0.25 = -0.000000
```



```
Elapsed milli seconds for 1000 run(s): 12.181069
Average per run milli seconds: 0.012181
Ran for L1=0.433, L2=1.000, L3=0.500. Eps=0.00100
||B1-A1||-L1 = 0.000001
||B2-A2||-L2 = 0.000001
||B3-A3||-L3 = 0.000001
||B1-B2||-0.25 = 0.000000
||B2-B3||-0.25 = -0.000000
||B3-B1||-0.25 = -0.000000
||B1-A1||-L1 = 0.000000
||B2-A2||-L2 = 0.000000
||B3-A3||-L3 = 0.000000
||B1-B2||-0.25 = 0.000000
||B2-B3||-0.25 = -0.000000
||B3-B1||-0.25 = -0.000000
```



```
Elapsed milli seconds for 1000 run(s): 4.500789
Average per run milli seconds: 0.004501
Ran for L1=0.815, L2=0.847, L3=0.480. Eps=0.00010
||B1-A1||-L1 = -0.000000
||B2-A2||-L2 = -0.000000
||B3-A3||-L3 = -0.000000
||B1-B2||-0.25 = 0.000000
||B2-B3||-0.25 = -0.000000
||B3-B1||-0.25 = -0.000000
||B1-A1||-L1 = -0.000000
||B2-A2||-L2 = -0.000000
||B3-A3||-L3 = -0.000000
||B1-B2||-0.25 = -0.000000
||B2-B3||-0.25 = -0.000000
||B3-B1||-0.25 = -0.000000
```



CONCLUSIONS

The method chosen seems to converge very fast to a solution. The running time was on average under 0.01 milliseconds. Improvements can still be done by removing altogether the more expensive trigonometric functions. The trig functions in math.h are known to be more precise than needed in some cases, but the approximations found (based on the fourier series, or based on the local shape of the sin for example) didn't provide the necessary accuracy for all cases.

Another change that can be done is to use Vincent's theorem, which is similar to Sturm but said to be more efficient. In most cases only one iteration of Sturm was necessary to reach NR method, which also needed usually between 2 to 4 iterations to reach the target, so this change might not yield much better results.

At the beginning I tried to use only one polynomial, and just choose the one that has the lowest bound on the roots. This had the same running time at best, and twice the running time at the worst, so the method was discarded.

I could not find any case where more than 2 solutions were found. It is possible that because of the data of the problem, only 2 real solutions always exist, but this was harder to prove without actually finding the roots. But if this is really the case, the solution array can be initialized to a smaller value.