三角函数的计算是个复杂的主题，有计算机之前，人们通常通过查找三角函数表来计算任意角度的三角函数的值。这种表格在人们刚刚产生三角函数的概念的时候就已经有了，它们通常是通过从已知值（比如sin(π/2)=1）开始并重复应用半角和和差公式而生成。

现在有了计算机，三角函数表便推出了历史的舞台。但是像我这样的喜欢刨根问底的人，不禁要问计算机又是如何计算三角函数值的呢。最容易想到的办法就是利用级数展开，比如泰勒级数来逼近三角函数，只要项数取得足够多就能以任意的精度来逼近函数值。除了泰勒级数逼近之外，还有其他许多的逼近方法，比如切比雪夫逼近、最佳一致逼近和Padé逼近等。

所有这些逼近方法本质上都是用多项式函数来近似我们要计算的三角函数，计算过程中必然要涉及到大量的浮点运算。在缺乏硬件乘法器的简单设备上（比如没有浮点运算单元的单片机），用这些方法来计算三角函数会非常的费时。为了解决这个问题，J. Volder于1959年提出了一种快速算法，称之为CORDIC(COordinate Rotation DIgital Computer) 算法，这个算法只利用移位和加减运算，就能计算常用三角函数值，如Sin，Cos，Sinh，Cosh等函数。 J. Walther在1974年在这种算法的基础上进一步改进，使其可以计算出多种超越函数，更大的扩展了Cordic 算法的应用。因为Cordic 算法只用了移位和加法，很容易用纯硬件来实现，因此我们常能在FPGA运算平台上见到它的身影。不过，大多数的软件程序员们都没有听说过这种算法，也更不会主动的去用这种算法。其实，在嵌入式软件开发，尤其是在没有浮点运算指令的嵌入式平台（比如定点型DSP）上做开发时，还是会遇上可以用到Cordic 算法的情况的，所以掌握基本的Cordic算法还是有用的。

**从二分查找法说起**

先从一个例子说起，知道平面上一点在直角坐标系下的坐标（X,Y）=（100，200），如何求的在极坐标系下的坐标（ρ,θ）。用计算器计算一下可知答案是（223.61，63.435）。

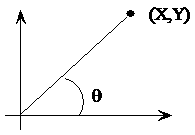
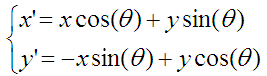
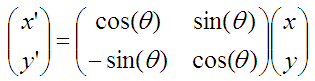


图 1 直角坐标系到极坐标系的转换

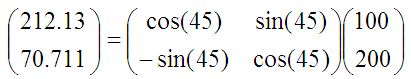
为了突出重点，这里我们只讨论X和Y都为正数的情况。这时θ=atan(y/x)。求θ的过程也就是求atan 函数的过程。Cordic算法采用的想法很直接，将（X，Y）旋转一定的度数，如果旋转完纵坐标变为了0，那么旋转的度数就是θ。坐标旋转的公式可能大家都忘了，这里把公式列出了。设（x,y）是原始坐标点，将其以原点为中心，顺时针旋转θ之后的坐标记为（x’,y’）,则有如下公式：



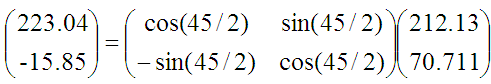
也可以写为矩阵形式：



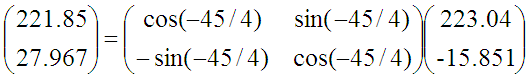
如何旋转呢，可以借鉴二分查找法的思想。我们知道θ的范围是0到90度。那么就先旋转45度试试。



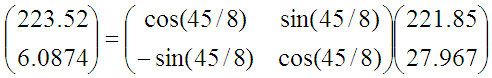
旋转之后纵坐标为70.71，还是大于0，说明旋转的度数不够，接着再旋转22.5度（45度的一半）。



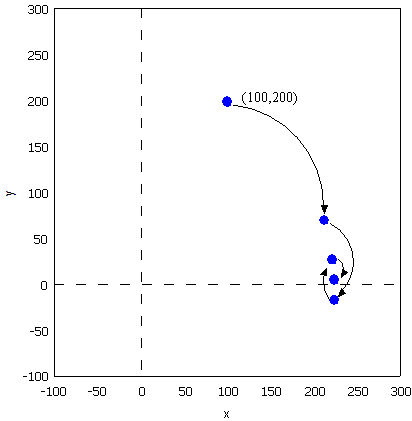
这时总共旋转了45+22.5=67.5度。结果纵坐标变为了负数，说明θ<67.5度，这时就要往回转，还是二分查找法的思想，这次转11.25度。



这时总共旋转了45+22.5-11.25=56.25度。又转过头了，接着旋转，这次顺时针转5.625度。



这时总共旋转了45+22.5-11.25+5.625=61.875度。这时纵坐标已经很接近0了。我们只是说明算法的思想，因此就不接着往下计算了。计算到这里我们给的答案是 61.875±5.625。二分查找法本质上查找的是一个区间，因此我们给出的是θ值的一个范围。同时，坐标到原点的距离ρ也求出来了，ρ=223.52。与标准答案比较一下计算的结果还是可以的。旋转的过程图示如下。



可能有读者会问，计算中用到了 sin 函数和 cos 函数，这些值又是怎么计算呢。很简单，我们只用到很少的几个特殊点的sin 函数和 cos 函数的值，提前计算好存起来，用时查表。

下面给出上面方法的C 语言实现。

**[cpp]** [view plaincopy](http://blog.csdn.net/liyuanbhu/article/details/8458769)

1. #include <stdio.h>
2. #include <stdlib.h>
4. **double** my\_atan2(**double** x, **double** y);
5. **int** main(**void**)
6. {
7. **double** z = my\_atan2(100.0, 200.0);
8. printf("\n z = %f \n", z);
10. **return** 0;
11. }

14. **double** my\_atan2(**double** x, **double** y)
15. {
16. **const** **double** sine[] = {0.7071067811865,0.3826834323651,0.1950903220161,0.09801714032956,
17. 0.04906767432742,0.02454122852291,0.01227153828572,0.006135884649154,0.003067956762966
18. ,0.001533980186285,7.669903187427045e-4,3.834951875713956e-4,1.917475973107033e-4,
19. 9.587379909597735e-5,4.793689960306688e-5,2.396844980841822e-5
20. };
22. **const** **double** cosine[] = {0.7071067811865,0.9238795325113,0.9807852804032,0.9951847266722,
23. 0.9987954562052,0.9996988186962,0.9999247018391,0.9999811752826,0.9999952938096,
24. 0.9999988234517,0.9999997058629,0.9999999264657,0.9999999816164,0.9999999954041,
25. 0.999999998851,0.9999999997128
26. };

29. **int** i = 0;
30. **double** x\_new, y\_new;
31. **double** angleSum = 0.0;
32. **double** angle = 45.0;
34. **for**(i = 0; i < 15; i++)
35. {
36. **if**(y > 0)
37. {
38. x\_new = x \* cosine[i] + y \* sine[i];
39. y\_new = y \* cosine[i] - x \* sine[i];
40. x = x\_new;
41. y = y\_new;
42. angleSum += angle;
43. }
44. **else**
45. {
46. x\_new = x \* cosine[i] - y \* sine[i];
47. y\_new = y \* cosine[i] + x \* sine[i];
48. x = x\_new;
49. y = y\_new;
50. angleSum -= angle;
51. }
52. printf("Debug: i = %d angleSum = %f, angle = %f\n", i, angleSum, angle);
53. angle /= 2;
54. }
55. **return** angleSum;
56. }

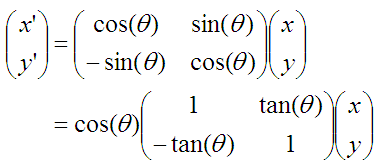
程序运行的输出结果如下：

**[plain]** [view plaincopy](http://blog.csdn.net/liyuanbhu/article/details/8458769)

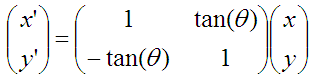
1. Debug: i = 0 angleSum = 45.000000, angle = 45.000000
2. Debug: i = 1 angleSum = 67.500000, angle = 22.500000
3. Debug: i = 2 angleSum = 56.250000, angle = 11.250000
4. Debug: i = 3 angleSum = 61.875000, angle = 5.625000
5. Debug: i = 4 angleSum = 64.687500, angle = 2.812500
6. Debug: i = 5 angleSum = 63.281250, angle = 1.406250
7. Debug: i = 6 angleSum = 63.984375, angle = 0.703125
8. Debug: i = 7 angleSum = 63.632813, angle = 0.351563
9. Debug: i = 8 angleSum = 63.457031, angle = 0.175781
10. Debug: i = 9 angleSum = 63.369141, angle = 0.087891
11. Debug: i = 10 angleSum = 63.413086, angle = 0.043945
12. Debug: i = 11 angleSum = 63.435059, angle = 0.021973
13. Debug: i = 12 angleSum = 63.424072, angle = 0.010986
14. Debug: i = 13 angleSum = 63.429565, angle = 0.005493
15. Debug: i = 14 angleSum = 63.432312, angle = 0.002747
17. z = 63.432312

**减少乘法运算**

现在已经有点 Cordic 算法的样子了，但是我们看到没次循环都要计算 4 次浮点数的乘法运算，运算量还是太大了。还需要进一步的改进。改进的切入点当然还是坐标变换的过程。我们将坐标变换公式变一下形。



可以看出 cos(θ)可以从矩阵运算中提出来。我们可以做的再彻底些，直接把 cos(θ) 给省略掉。



省略cos(θ)后发生了什么呢，每次旋转后的新坐标点到原点的距离都变长了，放缩的系数是1/cos(θ)。不过没有关系，我们求的是θ，不关心ρ的改变。这样的变形非常的简单，但是每次循环的运算量一下就从4次乘法降到了2次乘法了。

还是给出 C 语言的实现：

**[cpp]** [view plaincopy](http://blog.csdn.net/liyuanbhu/article/details/8458769)

1. **double** my\_atan3(**double** x, **double** y)
2. {
3. **const** **double** tangent[] = {1.0,0.4142135623731,0.1989123673797,0.09849140335716,0.04912684976947,
4. 0.02454862210893,0.01227246237957,0.006136000157623,0.003067971201423,
5. 0.001533981991089,7.669905443430926e-4,3.83495215771441e-4,1.917476008357089e-4,
6. 9.587379953660303e-5,4.79368996581451e-5,2.3968449815303e-5
7. };

10. **int** i = 0;
11. **double** x\_new, y\_new;
12. **double** angleSum = 0.0;
13. **double** angle = 45.0;
15. **for**(i = 0; i < 15; i++)
16. {
17. **if**(y > 0)
18. {
19. x\_new = x + y \* tangent[i];
20. y\_new = y - x \* tangent[i];
21. x = x\_new;
22. y = y\_new;
23. angleSum += angle;
24. }
25. **else**
26. {
27. x\_new = x - y \* tangent[i];
28. y\_new = y + x \* tangent[i];
29. x = x\_new;
30. y = y\_new;
31. angleSum -= angle;
32. }
33. printf("Debug: i = %d angleSum = %f, angle = %f, ρ = %f\n", i, angleSum, angle, hypot(x,y));
34. angle /= 2;
35. }
36. **return** angleSum;
37. }

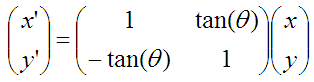
计算的结果是：

**[plain]** [view plaincopy](http://blog.csdn.net/liyuanbhu/article/details/8458769)

1. Debug: i = 0 angleSum = 45.000000, angle = 45.000000, ρ = 316.227766
2. Debug: i = 1 angleSum = 67.500000, angle = 22.500000, ρ = 342.282467
3. Debug: i = 2 angleSum = 56.250000, angle = 11.250000, ρ = 348.988177
4. Debug: i = 3 angleSum = 61.875000, angle = 5.625000, ρ = 350.676782
5. Debug: i = 4 angleSum = 64.687500, angle = 2.812500, ρ = 351.099697
6. Debug: i = 5 angleSum = 63.281250, angle = 1.406250, ρ = 351.205473
7. Debug: i = 6 angleSum = 63.984375, angle = 0.703125, ρ = 351.231921
8. Debug: i = 7 angleSum = 63.632813, angle = 0.351563, ρ = 351.238533
9. Debug: i = 8 angleSum = 63.457031, angle = 0.175781, ρ = 351.240186
10. Debug: i = 9 angleSum = 63.369141, angle = 0.087891, ρ = 351.240599
11. Debug: i = 10 angleSum = 63.413086, angle = 0.043945, ρ = 351.240702
12. Debug: i = 11 angleSum = 63.435059, angle = 0.021973, ρ = 351.240728
13. Debug: i = 12 angleSum = 63.424072, angle = 0.010986, ρ = 351.240734
14. Debug: i = 13 angleSum = 63.429565, angle = 0.005493, ρ = 351.240736
15. Debug: i = 14 angleSum = 63.432312, angle = 0.002747, ρ = 351.240736
17. z = 63.432312

**消除乘法运算**

我们已经成功的将乘法的次数减少了一半，还有没有可能进一步降低运算量呢？还要从计算式入手。



第一次循环时，tan(45)=1，所以第一次循环实际上是不需要乘法运算的。第二次运算呢？

Tan(22.5)=0.4142135623731,很不幸，第二次循环乘数是个很不整的小数。是否能对其改造一下呢？答案是肯定的。第二次选择22.5度是因为二分查找法的查找效率最高。如果选用个在22.5到45度之间的值，查找的效率会降低一些。如果稍微降低一点查找的效率能让我们有效的减少乘法的次数，使最终的计算速度提高了，那么这种改进就是值得的。

我们发现tan(26.565051177078)=0.5，如果我们第二次旋转采用26.565051177078度，那么乘数变为0.5，如果我们采用定点数运算的话（没有浮点协处理器时为了加速计算我们会大量的采用定点数算法）乘以0.5就相当于将乘数右移一位。右移运算是很快的，这样第二次循环中的乘法运算也被消除了。

类似的方法，第三次循环中不用11.25度，而采用 14.0362434679265 度。

Tan(14.0362434679265)= 1/4

乘数右移两位就可以了。剩下的都以此类推。

**[plain]** [view plaincopy](http://blog.csdn.net/liyuanbhu/article/details/8458769)

1. tan(45)= 1
2. tan(26.565051177078)= 1/2
3. tan(14.0362434679265)= 1/4
4. tan(7.1250163489018)= 1/8
5. tan(3.57633437499735)= 1/16
6. tan(1.78991060824607)= 1/32
7. tan(0.8951737102111)= 1/64
8. tan(0.4476141708606)= 1/128
9. tan(0.2238105003685)= 1/256

还是给出C语言的实现代码，我们采用循序渐进的方法，先给出浮点数的实现（因为用到了浮点数，所以并没有减少乘法运算量，查找的效率也比二分查找法要低，理论上说这个算法实现很低效。不过这个代码的目的在于给出算法实现的示意性说明，还是有意义的）。

**[cpp]** [view plaincopy](http://blog.csdn.net/liyuanbhu/article/details/8458769)

1. **double** my\_atan4(**double** x, **double** y)
2. {
3. **const** **double** tangent[] = {1.0, 1 / 2.0, 1 / 4.0, 1 / 8.0, 1 / 16.0,
4. 1 / 32.0, 1 / 64.0, 1 / 128.0, 1 / 256.0, 1 / 512.0,
5. 1 / 1024.0, 1 / 2048.0, 1 / 4096.0, 1 / 8192.0, 1 / 16384.0
6. };
7. **const** **double** angle[] = {45.0, 26.565051177078, 14.0362434679265, 7.1250163489018, 3.57633437499735,
8. 1.78991060824607, 0.8951737102111, 0.4476141708606, 0.2238105003685, 0.1119056770662,
9. 0.0559528918938, 0.027976452617, 0.01398822714227, 0.006994113675353, 0.003497056850704
10. };
12. **int** i = 0;
13. **double** x\_new, y\_new;
14. **double** angleSum = 0.0;
16. **for**(i = 0; i < 15; i++)
17. {
18. **if**(y > 0)
19. {
20. x\_new = x + y \* tangent[i];
21. y\_new = y - x \* tangent[i];
22. x = x\_new;
23. y = y\_new;
24. angleSum += angle[i];
25. }
26. **else**
27. {
28. x\_new = x - y \* tangent[i];
29. y\_new = y + x \* tangent[i];
30. x = x\_new;
31. y = y\_new;
32. angleSum -= angle[i];
33. }
34. printf("Debug: i = %d angleSum = %f, angle = %f, ρ = %f\n", i, angleSum, angle[i], hypot(x, y));
35. }
36. **return** angleSum;
37. }

程序运行的输出结果如下：

**[plain]** [view plaincopy](http://blog.csdn.net/liyuanbhu/article/details/8458769)

1. Debug: i = 0 angleSum = 45.000000, angle = 45.000000, ρ = 316.227766
2. Debug: i = 1 angleSum = 71.565051, angle = 26.565051, ρ = 353.553391
3. Debug: i = 2 angleSum = 57.528808, angle = 14.036243, ρ = 364.434493
4. Debug: i = 3 angleSum = 64.653824, angle = 7.125016, ρ = 367.270602
5. Debug: i = 4 angleSum = 61.077490, angle = 3.576334, ρ = 367.987229
6. Debug: i = 5 angleSum = 62.867400, angle = 1.789911, ρ = 368.166866
7. Debug: i = 6 angleSum = 63.762574, angle = 0.895174, ρ = 368.211805
8. Debug: i = 7 angleSum = 63.314960, angle = 0.447614, ρ = 368.223042
9. Debug: i = 8 angleSum = 63.538770, angle = 0.223811, ρ = 368.225852
10. Debug: i = 9 angleSum = 63.426865, angle = 0.111906, ρ = 368.226554
11. Debug: i = 10 angleSum = 63.482818, angle = 0.055953, ρ = 368.226729
12. Debug: i = 11 angleSum = 63.454841, angle = 0.027976, ρ = 368.226773
13. Debug: i = 12 angleSum = 63.440853, angle = 0.013988, ρ = 368.226784
14. Debug: i = 13 angleSum = 63.433859, angle = 0.006994, ρ = 368.226787
15. Debug: i = 14 angleSum = 63.437356, angle = 0.003497, ρ = 368.226788
17. z = 63.437356

有了上面的准备，我们可以来讨论定点数算法了。所谓定点数运算，其实就是整数运算。我们用256 表示1度。这样的话我们就可以精确到1/256=0.00390625 度了，这对于大多数的情况都是足够精确的了。256 表示1度，那么45度就是 45\*256 = 115200。其他的度数以此类推。

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 度数 | ×256 | 取整 |
| 1 | 45.0 | 11520 | 11520 |
| 2 | 26.565051177078 | 6800.65310133196 | 6801 |
| 3 | 14.0362434679265 | 3593.27832778918 | 3593 |
| 4 | 7.1250163489018 | 1824.00418531886 | 1824 |
| 5 | 3.57633437499735 | 915.541599999322 | 916 |
| 6 | 1.78991060824607 | 458.217115710994 | 458 |
| 7 | 0.8951737102111 | 229.164469814035 | 229 |
| 8 | 0.4476141708606 | 114.589227740302 | 115 |
| 9 | 0.2238105003685 | 57.2954880943458 | 57 |
| 10 | 0.1119056770662 | 28.647853328949 | 29 |
| 11 | 0.0559528918938 | 14.3239403248137 | 14 |
| 12 | 0.027976452617 | 7.16197186995294 | 7 |
| 13 | 0.01398822714227 | 3.58098614841984 | 4 |
| 14 | 0.006994113675353 | 1.79049310089035 | 2 |
| 15 | 0.003497056850704 | 0.8952465537802 | 1 |

C 代码如下：

**[cpp]** [view plaincopy](http://blog.csdn.net/liyuanbhu/article/details/8458769)

1. **int** my\_atan5(**int** x, **int** y)
2. {
3. **const** **int** angle[] = {11520, 6801, 3593, 1824, 916, 458, 229, 115, 57, 29, 14, 7, 4, 2, 1};
5. **int** i = 0;
6. **int** x\_new, y\_new;
7. **int** angleSum = 0;
9. x \*= 1024;// 将 X Y 放大一些，结果会更准确
10. y \*= 1024;
12. **for**(i = 0; i < 15; i++)
13. {
14. **if**(y > 0)
15. {
16. x\_new = x + (y >> i);
17. y\_new = y - (x >> i);
18. x = x\_new;
19. y = y\_new;
20. angleSum += angle[i];
21. }
22. **else**
23. {
24. x\_new = x - (y >> i);
25. y\_new = y + (x >> i);
26. x = x\_new;
27. y = y\_new;
28. angleSum -= angle[i];
29. }
30. printf("Debug: i = %d angleSum = %d, angle = %d\n", i, angleSum, angle[i]);
31. }
32. **return** angleSum;
33. }

计算结果如下：

**[plain]** [view plaincopy](http://blog.csdn.net/liyuanbhu/article/details/8458769)

1. Debug: i = 0 angleSum = 11520, angle = 11520
2. Debug: i = 1 angleSum = 18321, angle = 6801
3. Debug: i = 2 angleSum = 14728, angle = 3593
4. Debug: i = 3 angleSum = 16552, angle = 1824
5. Debug: i = 4 angleSum = 15636, angle = 916
6. Debug: i = 5 angleSum = 16094, angle = 458
7. Debug: i = 6 angleSum = 16323, angle = 229
8. Debug: i = 7 angleSum = 16208, angle = 115
9. Debug: i = 8 angleSum = 16265, angle = 57
10. Debug: i = 9 angleSum = 16236, angle = 29
11. Debug: i = 10 angleSum = 16250, angle = 14
12. Debug: i = 11 angleSum = 16243, angle = 7
13. Debug: i = 12 angleSum = 16239, angle = 4
14. Debug: i = 13 angleSum = 16237, angle = 2
15. Debug: i = 14 angleSum = 16238, angle = 1
17. z = 16238

16238/256=63.4296875度，精确的结果是63.4349499度，两个结果的差为0.00526，还是很精确的。

到这里 CORDIC 算法的最核心的思想就介绍完了。当然，这里介绍的只是CORDIC算法最基本的内容，实际上，利用CORDIC 算法不光可以计算 atan 函数，其他的像 Sin，Cos，Sinh，Cosh 等一系列的函数都可以计算，不过那些都不在本文的讨论范围内了。另外，每次旋转时到原点的距离都会发生变化，而这个变化是确定的，因此可以在循环运算结束后以此补偿回来，这样的话我们就同时将（ρ,θ）都计算出来了。

想进一步深入学习的可以阅读 John Stephen Walther 于2000年发表在 Journal of VLSI signal processing systems for signal, image and video technology上的综述性文章“The Story of Unified Cordic”。