17

Solving a Cubic Equation

Why Study and Solve Cubic Equations?

- **They model systems with non-linear behaviour**. Cubic relationships arise in many problems, including beam bending, fluid flow, population models, and optimisation. Being able to solve a cubic gives us information about their turning points, thresholds, and stability boundaries.
- **They're more complicated than quadratics**. Quadratics can always be solved with a simple formula; cubics introduce new algebra such as multiple real roots, cusps, and the need for more complex solution methods.
- **Their solution led to major advances in algebra**. Cardano's solution of the cubic in the 1500s drove the development of complex numbers, symbolic manipulation, and the idea of radicals. Cubics are tied to the birth of modern algebra.

17.1 Introduction

The discovery of a general solution to cubic equations can be traced back to the Italian mathematicians Scipione del Ferro (1465–1526), Niccolò Tartaglia (1500–1557), and Gerolamo Cardano (1501–1576). Cardano's publication of Tartaglia's solution in *Ars Magna* in 1545 laid the groundwork for solving higher-degree polynomial, but his approach was geometrical, involving literal cubes and their volumes, which was standard for a time when algebra was in its infancy [**Dun90**]. However, we'll present his ideas algebraically, which clarifies many aspects of his approach, such as his reliance on square roots of negative numbers (i.e. on

complex numbers). For instance, consider the *depressed* cubic $x^3 - 15x - 4 = 0$ (depressed because there's no x^2 term, not because of any emotional malaise). Applying Cardano's formula will lead to:

$$x = \sqrt[3]{2 + \sqrt{-121}} - \sqrt[3]{-2 + \sqrt{-121}}$$

In an historical period when even negative numbers were still viewed suspiciously, utilizing negative square roots were inconceivable, and there was a strong temptation to dismiss equations of this type as unsolvable. Yet it can easily be checked that the equation has a real solution at $x = 4$.

Cardano took a few half-hearted stabs at investigating this problem, but ultimately dismissed the whole enterprise as being "as subtle as it is useless."

It would be another generation before Rafael Bombelli (1526-1573), in his 1572 treatise *Algebra*, took the bold step of regarding complex numbers as necessary. But their full importance didn't really become evident until the work of Euler, Gauss, and Cauchy more than two centuries later. In the meantime, François Viète's (1540-1603) method emerged as a pleasing alternative to solving cubic equations, offering a solution when all three roots were real which used only 'respectable' trigonometry.

We'll implement Cardano's and Viète's work with the help of Python's complex numbers and its `cmath` module. If you need a quick refresher on those topics, please cast an eye over Appendix J. We'll then change track and consider *visual* ways of finding the roots of cubic equations. We'll look at three approaches: ordinary 2D plots, modulus surfaces, and phase maps with domain coloring and contours, all of which are easy to implement using matplotlib.

17.2 Depressing Cubic Equations

To solve $f(x) = ax^3 + bx^2 + cx + d = 0$, we start by rewriting it as a depressed cubic of the form

$$x^3 + px = q.$$

Substitute $y - h$ for x in the full cubic:

$$\begin{aligned} ax^3 + bx^2 + cx + d &= a(y-h)^3 + b(y-h)^2 + c(y-h) + d \\ &= a(y^3 - 3hy^2 + 3h^2y - h^3) + b(y^2 - 2hy + h^2) + c(y-h) + d \\ &= ay^3 - 3ahy^2 + 3ah^2y - ah^3 + by^2 - 2bhy + bh^2 + cy - ch + d \\ &= y^3 + y^2(-3ah + b)/a + y(3ah^2 - 2bh + c)/a + \\ &(-ah^3 + bh^2 - ch + d)/a \end{aligned}$$

If we set $h = \frac{b}{3a}$, then the y^2 term disappears, and the equation takes on the desired form $x^3 + px = q$, where:

$$\begin{aligned} p &= (3ac - b^2)/(3a^2) \\ q &= -(2b^3 - 9abc + 27a^2d)/(27a^3) \end{aligned}$$

The importance of this manipulation is that it shows that any general cubic can be rewritten in depressed form. In addition, p and q can be used to calculate the cubic's *discriminant* Δ:

$$\Delta = \left(\frac{q}{2}\right)^2 + \left(\frac{p}{3}\right)^3$$

Δ provides information about the nature of the cubic's roots:

- If $\Delta > 0$ then the equation has one real root and two complex conjugate roots.
- If $\Delta = 0$ then all the roots are real, and at least two of them are equal.
- If $\Delta < 0$ then the equation has three distinct real roots.

Later on we'll employ the discriminant to decide whether to use Cardano's or Viète's method for solving a cubic.

17.3 The Tartaglia-Cardano's Method

Consider the depressed cubic: $x^3 + px = q$. The key idea is to rewrite x as a difference of two quantities, $x = t - u$, and then find t and u.

By the binomial theorem, we have

$$(t-u)^3 = t^3 - 3t^2u + 3tu^2 - u^3.$$

Thus

$$\begin{aligned} t^3 - u^3 &= (t-u)^3 + 3t^2u - 3tu^2 \\ &= (t-u)^3 + 3tu(t-u). \end{aligned}$$

The equation becomes $x^3 + px = q$ if we set

$$p = 3tu, \qquad q = t^3 - u^3.$$

The first equation gives

$$u = \frac{p}{3t}.$$

Substitute this into the second equation to obtain

$$q = t^3 - \frac{p^3}{27t^3}.$$

Rearrange and multiply through by t^3 to get

$$t^6 - qt^3 - \frac{p^3}{27} = 0.$$

We can apply the standard quadratic formula by assuming that we're solving for t^3:

$$t^3 = \frac{q \pm \sqrt{q^2 + 4p^3/27}}{2} = \frac{q}{2} \pm \sqrt{\frac{q^2}{4} + \frac{p^3}{27}}.$$

Therefore,

$$t = \sqrt[3]{\frac{q}{2} \pm \sqrt{\frac{q^2}{4} + \frac{p^3}{27}}}.$$

$u^3 = t^3 - q$, so

$$u = \sqrt[3]{-\frac{q}{2} \pm \sqrt{\frac{q^2}{4} + \frac{p^3}{27}}}.$$

Also, $x = t - u$, so

$$x = \sqrt[3]{\frac{q}{2} \pm \sqrt{\frac{q^2}{4} + \frac{p^3}{27}}} - \sqrt[3]{-\frac{q}{2} \pm \sqrt{\frac{q^2}{4} + \frac{p^3}{27}}}.$$

The square root term is the discriminant, which we can refer to directly:

$$x = \sqrt[3]{\frac{q}{2} \pm \sqrt{\Delta}} - \sqrt[3]{-\frac{q}{2} \pm \sqrt{\Delta}}.$$

There are two $\pm$ signs in the expression, which may make you think that we've found four solutions, but two of them are the same. Consider the expression to have the form $x = \sqrt[3]{a \pm b} - \sqrt[3]{-a \pm b}$. The solution with two minus signs inside the radicals equals the solution with two positive signs, because of the effects of factoring:

$$\sqrt[3]{a - b} - \sqrt[3]{-a - b} = (-1)\sqrt[3]{-a + b} - (-1)\sqrt[3]{a + b} = \sqrt[3]{a + b} - \sqrt[3]{-a + b}.$$

17.3.1 Cardano's Example. We'll solve $x^3 + 6x = 20$, which was one of Cardano's examples in *Ars Magna*. $p = 6$ and $q = 20$, so that

$$\Delta = (q/2)^2 + (p/3)^3 = 10^2 + 2^3 = 108$$

This means that there's one real root and two complex conjugate roots.

$$\begin{aligned} x &= \sqrt[3]{\tfrac{q}{2} \pm \sqrt{\Delta}} - \sqrt[3]{-\tfrac{q}{2} \pm \sqrt{\Delta}} \\ &= \sqrt[3]{10 \pm \sqrt{108}} - \sqrt[3]{-10 \pm \sqrt{108}} \end{aligned}$$

Cardano ignored any choice of operations that might lead to negative cube roots, which only leaves

$$x = \sqrt[3]{10 + \sqrt{108}} - \sqrt[3]{-10 + \sqrt{108}}$$

This simplifies to $x = 2$, as we can (almost) confirm with Python:

```
>>> import math
>>> d = math.sqrt(108)
>>> a = (10 + d)**(1/3)
>>> b = (-10 + d)**(1/3)
>>> a-b
1.9999999999999996
```

17.3.2 Finding the Other Roots. Once we've used Cardano's method to find a real root, the other two can be found by factorization. For example, we'd factorize $f(x) = x^3 + 6x - 20$ by dividing by $(x - 2)$, producing

$$f(x) = (x-2)(x^2+2x+10) = (x-2)(x-(-1+3i))(x-(-1-3i)).$$

We can solve $x^2+2x+10$ with the quadratic formula, and choose to give up upon discovering that $b^2 - 4ac < 0$ (where $a = 1, b = 2, c = 10$).

Factoring a cubic into a linear term and a quadratic is quite straightforward. If you divide $a_3x^3 + a_2x^2 + a_1x + a_0$ by $b_1x + b_0$ then the coefficients of $q_2x^2 + q_1x + q_0$ satisfy:

$$\begin{aligned} q_2 &= \frac{a_3}{b_1}, \\ q_1 &= \frac{a_2 - q_2 b_0}{b_1}, \\ q_0 &= \frac{a_1 - q_1 b_0}{b_1}, \\ R &= a_0 - q_0 b_0. \end{aligned}$$

where R is a remainder which we expect to be 0 when we factor by a linear equation using a root. These equalities are obtained by multiplying the linear and quadratic terms, and matching their results with the corresponding coefficients in the cubic:

$$(b_1x + b_0) \cdot (q_2x^2 + q_1x + q_0) = a_3x^3 + a_2x^2 + a_1x + a_0$$

Therefore:

$$\begin{aligned} a_3 &= b_1q_2, \\ a_2 &= b_1q_1 + b_0q_2, \\ a_1 &= b_1q_o + b_0q_1, \\ a_0 &= b_0q_0 + R. \end{aligned}$$

`divideCubicByLinear()` in cubicUtils.py is a translation of this into Python. Let's use it on Cardano's cubic:

```
>>> from cubicUtils import *
>>> divideCubicByLinear([1, 0, 6, -20],[1,-2])
([1.0, 2.0, 10.0], 0.0)
```

This is the quadratic $x^2 + 2x + 10$.

cubicUtils.py also contains a quadratic solver, which utilizes `cmath.sqrt()`. We can supply it with the coefficients of the quadratic returned by `divideCubicByLinear()` to get the complex conjugates:

```
>>> from cubicUtils import *
>>> quadraticSolve(1, 2, 10)
[(-1+3j), (-1-3j)]
```

17.3.3 Bombelli's Example. Now we'll consider $x^3-15x = 4$, one of Bombelli's examples, and be brave enough not be put off when we encounter $\sqrt{-1}$. $p = -15$ and $q = 4$, so that

$$\Delta = (q/2)^2 + (p/3)^3 = 2^2 + (-5)^3 = -121$$

This means that there are three distinct roots, but doesn't mean that we'll avoid complex numbers.

$$\begin{aligned} x &= \sqrt[3]{\tfrac{q}{2} \pm \sqrt{\Delta}} - \sqrt[3]{-\tfrac{q}{2} \pm \sqrt{\Delta}} \\ &= \sqrt[3]{2 \pm 11i} - \sqrt[3]{-2 \pm 11i} \end{aligned}$$

We'd dearly like the cube roots to be positive, so limit ourselves to:

$$\begin{aligned} x &= \sqrt[3]{2 + 11i} - \sqrt[3]{-2 + 11i} \\ &= \sqrt[3]{2 + 11i} + \sqrt[3]{2 - 11i} \\ &= (2 + i) + (2 - i) \\ &= 4 \end{aligned}$$

A real solution via the realm of complex numbers.

The calculation can be checked using Python:

```
>>> (2 + 11j) ** (1/3)
(2+1.0000000000000002j)
>>> (2 - 11j) ** (1/3)
(2-1.0000000000000002j)
```

or we can start at the solution and find integer powers:

```
>>> (2 + 1j) ** 3
(2+11j)
>>> (2 - 1j) ** 3
(2-11j)
```

Taking the cube root of a complex number can lead to some confusion, since only one result is given when using the ** operator. For instance if we take the cube root of $(-2+11i)$, which was the original form of the second root in the x equation above, then Python reports:

```
>>> (-2 + 11j) ** (1/3)
(1.8660254037844388+1.2320508075688772j)
```

It's necessary to obtain all three of the roots and select the most suitable one:

```
>>> import complexUtils
>>> rs = complexUtils.getRoots((-2 + 11j), 3)
>>> for r in rs: print(r)
(1.8660254037844388+1.2320508075688772j)
(-2.0000000000000004+0.9999999999999996j)
(0.1339745962155615-2.232050807568877j)
```

The second root, $(-2 + i)$, is the one we need.

17.4 Viète's Method

Viète's trigonometric method solves a cubic equation when all of its roots are real, which we can detect by checking that its discriminant Δ is negative. It utilizes the identity that relates the cosine of a triple angle to a cubic expression:

$$\cos(3\theta) = 4\cos^3(\theta) - 3\cos(\theta)$$

In the depressed cubic $x^3 + px = q$, assume that $x = k\cos\theta$ and $k^2 = -4p/3$ (making $x = 2\sqrt{(-p/3)}\cos\theta$). Substitute this into $x^3 + px = q$:

$$\begin{aligned}
(-8p/3)\sqrt{(-p/3)}\cos^3\theta + 2p\sqrt{(-p/3)}\cos\theta &= q \\
(4\cos^3\theta - 3\cos\theta)\left((-2p/3)\sqrt{(-p/3)}\right) &= q \\
\cos 3\theta &= q/((-2p/3)\sqrt{(-p/3)}) \\
&= \frac{q}{2\sqrt{(-p/3)^3}}
\end{aligned}$$

For this equation to have real solutions,

$$\begin{aligned}
\left|q/((-2p/3)\sqrt{(-p/3)})\right| &\leq 1 \\
q^2/((4p^2/9)(-p/3)) &\leq 1 \quad \text{(when } p < 0\text{)} \\
q^2/(-4p^3/27) &\leq 1 \\
-p^3/27 &\geq q^2/4 \\
p^3/27 + q^2/4 &\leq 0.
\end{aligned}$$

This means that when $p^3/27 + q^2/4 \leq 0$ (or equivalently when $\Delta \leq 0$), then we can indeed find real roots for $x^3 + px = q$ with $x = 2\sqrt{(-p/3)}\cos\theta$. Returning to the cosine,

$$\cos 3\theta = \frac{q}{2\sqrt{(-p/3)^3}},$$

the three angles corresponding to the roots are:

$$\theta_k = \frac{1}{3}\operatorname{acos}\left(\frac{q}{2\sqrt{(-p/3)^3}}\right) + \frac{2\pi k}{3}, \quad k = 0, 1, 2.$$

Substitute each of these into x to obtain

$$x_k = 2\sqrt{(-p/3)}\cos\theta_k.$$

This gives us three real roots without needing to rely on complex numbers.

17.4.1 Why $k^2 = -4p/3$? What's the reason for choosing $k^2 = -4p/3$ at the start of Viète's method? He began with $x^3 + px = q$, and assigned $x = k\cos\theta$, to produce:

$$k^3\cos^3\theta + pk\cos\theta = q$$

Factoring out a k^3 term gives:

$$k^3(\cos^3\theta + \frac{p}{k^2}\cos\theta) = q$$

Rearrange the identity $\cos 3\theta = 4\cos^3\theta - 3\cos\theta$ as:

$$\cos^3\theta = \frac{1}{4}(\cos 3\theta + 3\cos\theta)$$

and use its right-hand side to simplify the conic:

$$k^3\left[\frac{1}{4}(\cos 3\theta + 3\cos\theta)\right] + pk\cos\theta = q$$

or

$$\frac{k^3}{4}\cos 3\theta + \left(\frac{3k^3}{4} + pk\right)\cos\theta = q$$

Viète wanted this equation to reduce to a simple linear relation, ideally eliminating the $\cos\theta$ term completely. To do that, he set:

$$\frac{3k^3}{4} + pk = 0$$

which gives:

$$k^2 = -\frac{4p}{3}$$

17.4.2 Using the Method. We'll consider $x^3 - 2x = -1$, which makes $p = -3$ and $q = -1$. First we check that the discriminant Δ^2 is ≤ 0, making the equation suitable for Viète's method:

$$(p/3)^3 + (q/2)^2 = -1 + 0.25 = -0.75 \leq 0$$

The three angles for the roots:

$$\begin{aligned}\theta_k &= \tfrac{1}{3}\,\text{acos}(\tfrac{q}{2\sqrt{(-p/3)^3}}) + \tfrac{2\pi k}{3}, \quad k = 0, 1, 2.\\ x_k &= 2\sqrt{(-p/3)}\cos\theta_k\end{aligned}$$

Focusing on θ_0 and x_0:

$$\begin{aligned}\theta_0 &= \frac{1}{3}\operatorname{acos}\left(\frac{-1}{2\sqrt{(-3/3)^3}}\right)\\ &= \frac{1}{3}\operatorname{acos}\frac{-1}{2}\\ x_0 &= 2\sqrt{(3/3)}\cos\left(\frac{1}{3}\operatorname{acos}\frac{-1}{2}\right)\\ &= 2\cos\left(\frac{1}{3}\operatorname{acos}\frac{-1}{2}\right)\end{aligned}$$

Switching to Python:

```
>>> import math
>>> 2 * math.cos ( math.acos(-0.5)/3 )
1.532088886237956
```

The other two values are obtained by adding $2\pi/3$ and $4\pi/3$ (or equivalently $-2\pi/3$) to θ_0:

```
>>> theta0 = math.acos(-0.5)/3
>>> 2 * math.cos( theta0 + (2*math.pi)/3 )
-1.8793852415718166
>>> 2 * math.cos( theta0 - (2*math.pi)/3 )
0.3472963553338613
```

In summary, the three roots (to 4 d.p) are 1.5321, −1.8794, and 0.3473. These can be tested by plugging them back into the original cubic:

```
>>> from cubicUtils import cubic
>>> cubic(1.532088886237956, 1, 0, -3, 1)   # x^3 -3x + 1 = 0
-4.440892098500626e-16
>>> cubic(-1.8793852415718166, 1, 0, -3, 1)
8.881784197001252e-16
>>> cubic(0.3472963553338613, 1, 0, -3, 1)
-1.5543122344752192e-15
```

17.5 Implementing Cardano's and Viète's Methods

The function `cvRoots()` in cubicRoots.py uses either Cardano's or Viète's method to find the roots of a cubic depending on the discriminant of its depressed version.

The following runs of `findCubicRoots()` correspond to the earlier examples:

```
>>> from cubicRoots import *
>>> findCubicRoots((1, 0, 6, -20))
Equation: x^3 + 6x - 20 = 0
Discriminant: 108.0000
One real root and two complex conjugate roots
Roots:
```

```
   2.0000
  -1.0000+3.0000j
  -1.0000-3.0000j

>>> findCubicRoots((1, 0, -15, -4))
Equation: x^3 - 15x - 4 = 0
Discriminant: -121.0000
All three roots are real and distinct
Roots:
   4.0000
  -3.7321
  -0.2679

>>> findCubicRoots((1, 0, -3, 1))
Equation: x^3 - 3x + 1 = 0
Discriminant: -0.7500
All three roots are real and distinct
Roots:
   1.5321
  -1.8794
   0.3473
```

`findCubicRoots()` calls `cvRoots()` to find the roots, and various extra features for printing the discriminant and the roots. The essential structure of `cvRoots()` is shown below:

```
def cvRoots(a, b, c, d):
  if a == 0:
    raise ValueError("Coef 'a' must not be zero")

  # Convert to depressed cubic: x = t - b/(3a)
  p = (3*a*c - b*b)/(3*a*a)
  q = (2*b**3 - 9*a*b*c + 27*a*a*d)/(27*a**3)
  disc = (q/2)**2 + (p/3)**3
  shift = -b/(3*a)

  roots = []
  if abs(disc) < 1e-12:  # disc == 0
    # repeated real roots
    #  :
    roots = ...
  elif disc > 0:
    # Cardano's method
    # One real root and two complex conjugates
    #   :
    roots = ...
  else:  # disc < 0
    # Vieta's method
```

```
    # Three distinct real roots
    #   :
    roots = ...

  return disc, roots
```

p and q are the coefficients of the depressed cubic $x^3 + px = q$, which allows the discriminant (`disc`) to be calculated and used to direct the execution to one of three branches.

17.5.1 Cardano's Method in `cvRoots()`. The code branch that implements Cardano's method:

```
    sqrtD = math.sqrt(disc)
    # t = (q/2 + sqrtD)**(1/3)
    t = complexUtils.getRealRoot((q/2 + sqrtD), 3)
    u = (-q/2 + sqrtD)**(1/3)

    # Ensure t*u = -p/3 relationship;
    # avoids using the wrong cube root
    if t != 0 and isinstance(u, complex):
      vPrev = u
      u = -p/(3*t)

    if a < 0:
      u = -u   # so x1, x2, x3 are additions
    x1 = t - u
    # compute the other two roots using the
    # principal cube root of unity w = e^(2*pi*i/3)
    w = cmath.exp(2*math.pi*1j/3)
    x2 = w*t - w*w*u
    x3 = w*w*t - w*u
    roots = [complex(x1.real+shift, 0), x2+shift, x3+shift]
```

The calculation of t and u are almost as expected, but matters are complicated by having to ensure that the real cube root is returned for t and u. For t this is guaranteed by using `complexUtils.getRoots()` to get all three roots, and then selecting the real one. The correct value for u is determined by making sure that the $tu = -p/3$ relationship is true.

Another adjustment is to check the sign of a to decide if x should be calculated using an addition of t and u.

The last few lines are not part of Cardano's method since they utilize the cube roots of unity to 'rotate' the t and u values to obtain the other two x roots.

17.5.2 Viète's Method in `cvRoots()`. The code branch that implements Viète's method:

```
arg = q/(2*math.sqrt(-(p/3)**3))
theta = math.acos(arg)/3
m = 2*math.sqrt(-p/3)
roots = [ m*math.cos(theta) + shift,
          m*math.cos(theta + 2*math.pi/3) + shift,
          m*math.cos(theta - 2*math.pi/3) + shift ]
```

This is almost unchanged from the maths.

17.6 Visually Finding Roots in a 2D Plot

In the last section we found the roots of $x^3 + 6x - 20 = 0$. One way to check the results is to plot the cubic using `plot2D.py`. For example:

```
> python plot2D.py
Enter cubic coefs (a b c d): 1 0 6 -20
Range (or 3): 4
Range: [-4, 4]
```

This displays the graph in Fig. 17.1.

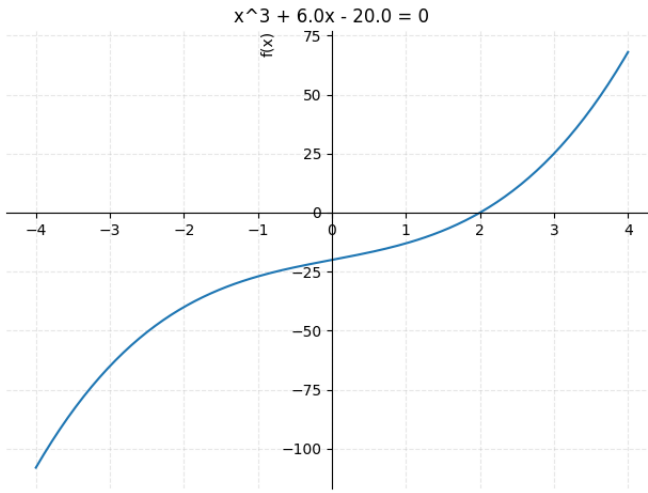

Figure 17.1. A 2D Plot of a Cubic

Fig. 17.1 illustrates the main drawback of visualizing cubics in 2D – only the real roots (in this case $x = 2$) are shown. However, there are ways to locate the complex conjugate pair $(-1 \pm 3i)$ once the real root has been identified. Let the roots of $y = px^3 + qx^2 + rx + s = 0$ be $x = c$ and $x = a \pm ib$, where a, b, c are real numbers. The factorized cubic is

$$y = (x - c)(x^2 - [a + ib])(x - [a - ib]) = 0,$$

or

$$y = (x - c)(x^2 - 2ax + a^2 + b^2) = 0.$$

Our aim is to find a and b by employing the geometric properties shown in Fig. 17.2.

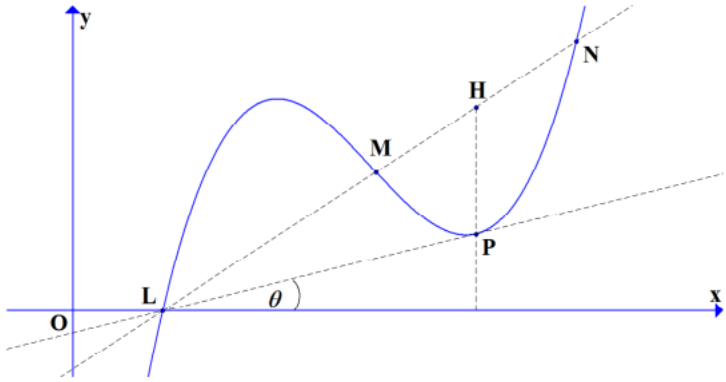

Figure 17.2. Find the Complex Conjugate Roots

We draw a line LMN through the cubic, where L is the real root at $x = c$. The general equation of a line is given by $y - y_1 = m(x - x_1)$, where m is the slope. So for LMN we have $y = m(x - c)$. When it intersects the cubic at M and N we get

$$m(x - c) = (x - c)(x^2 - 2ax + a^2 + b^2),$$

which simplifies to

$$x^2 - 2ax + a^2 + b^2 - m = 0.$$

Solving this gives us

$$x = a \pm \sqrt{m - b^2}.$$

M and N will have the same x value when their x equations are equal, implying that $m = b^2$, and that $x = a$. The line becomes LP, a tangent to the cubic, with LP = the slope = $m = b^2 = \tan\theta$. The real parts of the complex roots $x = a \pm ib$ can now be located:

- a is the x coordinate of the tangent point P. This can be read off by drawing a vertical line from P down to the x-axis;
- $b = \sqrt{\tan\theta}$, where θ is measured from the graph, or alternatively $b = \sqrt{|m|}$ since we're only interested in the magnitude of the slope.

As an example, let's find the roots for $x^3 - 3x^2 + x + 5$. Of course, we can just call `findCubicRoots()`:

```
>>> from cubicRoots import *
>>> findCubicRoots((1, -3, 1, 5))
Equation: x^3 - 3x^2 + x + 5 = 0
Discriminant: 3.7037
One real root and two complex conjugate roots
Roots:
  -1.0000
   2.0000+1.0000j
   2.0000-1.0000j
```

However, let's use the geometric approach on Fig. 17.3, which uses a graph drawn by `plot2D.py`:

```
> python plot2D.py
Enter cubic coefs (a b c d): 1 -3 1 5
Range (or 3):
Range: [-3, 3]
```

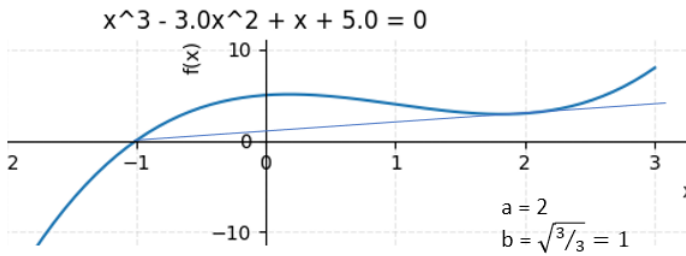

Figure 17.3. Find the Complex Roots for $x^3 - 3x^2 + x + 5$

To be clear, `plot2D.py` only draws the curve; we manually drew the tangent line and calculated a and b in Fig. 17.3. The graph suggests that $a = 2$ and $b \approx 1$, which is in agreement with the actual values for the complex roots.

17.7 Visually Finding Roots with a Modulus Surface

Long and Hern [**LH89**] popularized the use of a *modulus surface* to find the roots of complex functions. This approach can easily be applied to a cubic function f by supplying it with complex numbers rather than reals. Each complex result is converted to a modulus, $|f(z)|$ so the plot can be 3D – the $(x + iy)$ inputs occupy the xy plane and the moduli are plotted up the z-axis. The roots occur when the modulus surface touches the plane.

modulusPlot.py implements this using matplotlib and a small use of numpy so the x and y values can be converted into a mesh. Fig. 17.4 shows the graph generated for $x^3 - 3x^2 + x + 5$:

```
> python modulusPlot.py
Enter cubic coefs (a b c d): 1 -3 1 5
Range (or 3):
Real range: [-3, 3]
Imaginary range: [-3, 3]
```

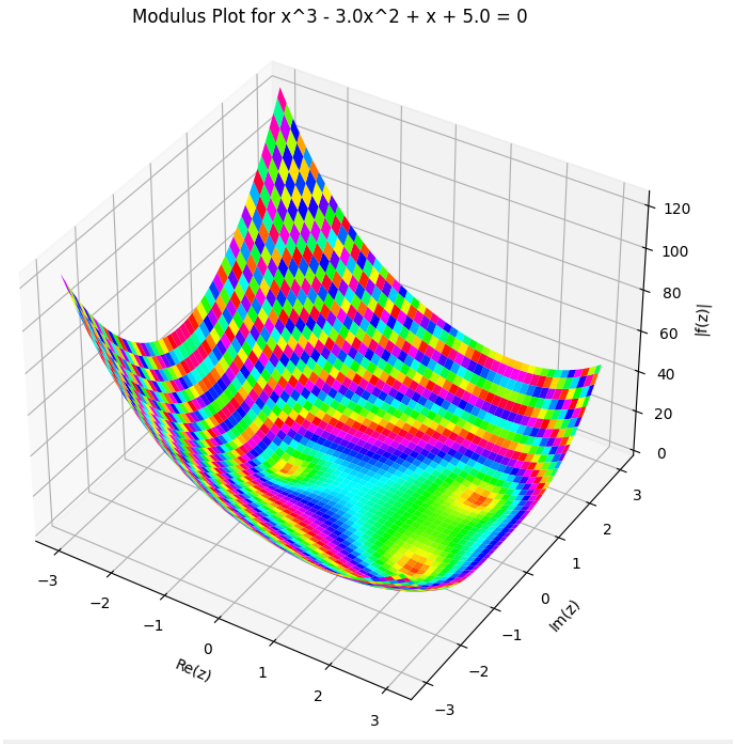

Figure 17.4. Modulus Plot for $x^3 - 3x^2 + x + 5$

The hsv colormap used by `matplotlib.plot_surface()` has been made to cycle through all its colors in steps of 10, starting from $z = 0$. This makes it much easier to see the location of the three roots (the red spots at $(-1, 0)$, and $(2 \pm i)$). The modulus data is generated by:

```
def getMagnitudes(realVs, imagVs):
  mags = []
  for im in imagVs:
```

```
    rowMag = []
    for re in realVs:
      z = complex(re, im)
      try:
        w = cubicUtils.cubic(z, a, b, c, d)
      except (ValueError, ZeroDivisionError):
        w = complex(float('nan'), float('nan'))
      rowMag.append( abs(w))
    mags.append(rowMag)
  return mags
```

a, b, c, and d are the coefficients supplied by the user.

Fig. 17.4 also hints at the weakness of this approach: we needed to rotate the graph so that the roots became easier to see, and 'view improvement' may not always be possible, especially if the surface is very mountainous. The other issue, common to all of these visualizations, is that the roots can only be estimated approximately.

17.8 Visually Finding Roots with Enhanced Phase Maps

By 'enhanced' we mean phase maps with domain coloring and contours, an approach that's explained in Appendix J on complex numbers. Nevertheless, we'll briefly describe the meaning of its color scheme here as well. The cubic is displayed as a phase map by calling phasePlot.py:

```
> python phasePlot.py
Enter cubic coefs (a b c d): 1 -3 1 5
Range (or 3):
Real range: [-3, 3]
Imaginary range: [-3, 3]
```

with the graph in Fig. 17.5 being drawn.

Positive numbers are colored red, negative numbers are colored blue. This is easiest to understand by looking along the real axis from right to left which shows values switching from red to light blue at $(-1, 0)$ which correspond to the way that the real part of the curve behaves in Fig. 17.3.

Zeros and poles occur at the points where all the colors meet. A zero is distinguished by the change from red, yellow, green, to blue in a counter-clockwise order around the point. A pole, which is when the output extends off to infinity, has its colors change in a clockwise order. This means that there are three zeros in Fig. 17.5, at $(-1, 0)$ and $(2 \pm i)$, as expected.

There are two types of contour line drawn over the surface – the black lines are contours of equal phase, and are labeled in degrees. The white lines are contours of equal magnitude (modulus), which pleasingly match the shape of the curve in Fig. 17.4.

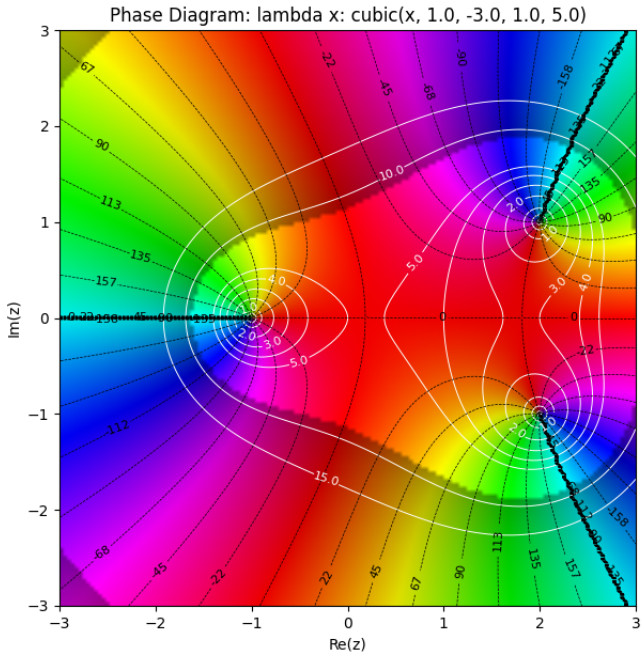

Figure 17.5. Phase Plot for $x^3 - 3x^2 + x + 5$

phasePlot.py is actually very simple, little more than a wrapper around a call to `showPhase()` in complexUtils.py.