Greatest Common Divisor[*]

Graeme Taylor

February 8, 2005

In a computer algebra setting, the greatest common divisor is necessary to make sense of fractions, whether to work with rational numbers or ratios of polynomials. Generally a canonical form will require common factors in the numerator and denominator to be cancelled. For instance, the expressions

$$\frac{-8}{6},\ \frac{4}{-3},\ -(1+(\frac{1}{3})),\ -1\times(\frac{12}{9}),\ \frac{2}{3}-2$$

all mean the same thing, which would be preferable to write as $\frac{-4}{3}$ (that is, a single fraction of the form $\frac{z}{n}$, $z \in \mathbb{Z}$ (an integer), $n \in \mathbb{N}$ (natural), $gcd(z,n) = 1$).

Some useful properties of the gcd

Suppose we are working with a suitable ring R (for instance, the integers, or the rational polynomials). Then the 'greatest' (in the sense of magnitude for numbers, or degree for polynomials) divisor of an element r is clearly itself ($1r = r$). Furthermore, for any element r of R, $0r = 0$ so r is a divisor of 0. Hence a first observation:

$$\text{For any r}, gcd(r,0) = r$$

Next up, it's completely trivial that if c is the gcd of a and b, it is the gcd of b and a. That is,

$$gcd(a,b) = gcd(b,a)$$

Letting c be any divisor, there must be α and β such that $a = \alpha c$ and $b = \beta c$ (otherwise, c isn't a divisor!). So we observe that $a - b = \alpha c - \beta c = (\alpha - \beta)c$. That is,

$$\text{If } c \text{ divides } a \text{ and } c \text{ divides } b\text{, then } c \text{ divides } a - b$$

Then, since division can be thought of as repeated subtraction, we deduce

$$\text{If } c \text{ divides } a \text{ and } c \text{ divides } b\text{, and if } a \geq b\text{, then } c \text{ divides the remainder of } \frac{a}{b}.\ \text{(for non-zero } b)$$

[*]First appeared on Everything2, at *http://www.everything2.com/index.pl?node_id=1698077*

Integer GCDs

Equipped with just these four simple facts, you might be able to determine a process for finding the greatest divisor of two integers (note that this is defined to be a natural number.) If you can't, the work has conveniently already been done by Euclid, some 2 millenia ago. Despite its age, Euclid's algorithm remains the best method for this task, and the so called extended euclid algorithm even allows you to express the gcd in terms of the original integers.

Polynomial GCDs

To prove that expertise in this area has in fact progressed in the past two thousand years, the rest of this writeup considers the problem of finding the gcd of two polynomials (in a single variable). This is equivalent to finding their common roots, meaning that gcd calculations can be applied to solving systems of equations.

Euclidean Techniques

A natural first approach is to refine Euclid's algorithm as devised for numbers to work on polynomials. It is indeed possible to create such an algorithm. However, one of two problems arises. Either you are forced to work with fractional coefficients (awkward) or a fraction-free approach is employed by rescaling- which causes the size of coefficients in intermediate expressions to skyrocket (again awkward). Non-euclidean techniques can be devised which call for a euclidean gcd algorithm ony in circumstances where these two pitfalls can be avoided. But before discussing these, there is an approach along Euclidean lines which works somewhat better than a naive re-scaling version of Euclid's algorithm.

GCDs, Resultants, and the Sylvester Matrix

First, some definitions. For polynomials $P = a_nX^n + \ldots + a_0$ and $Q = b_mX^m + \ldots + b_0$, whose roots are the sets α_i and β_j respectively, we define the resultant of P and Q as

$$r(P,Q) = \prod_{i=1..n} \prod_{j=1..m} (\beta_j - \alpha_i)$$

That is, the product of all possible differences of roots.

Then the Sylvester Matrix of P and Q is an $(m+n) \times (m+n)$ square matrix generated by m copies of the coefficients of P and n copies of those of Q, shifted right each row and padded by zeros:

$$\begin{array}{ccccccc}
a_n & a_{n-1} & \ldots & a_0 & 0 & \ldots & 0 \\
0 & a_n & \ldots & a_1 & a_0 & \ldots & 0 \\
. & & . & & & & . \\
. & & & . & & & . \\
. & & & & . & & . \\
0 & 0 & \ldots & 0 & a_n & \ldots & a_0 \\
b_m & b_{m-1} & \ldots & b_0 & 0 & \ldots & 0 \\
0 & b_m & \ldots & b_1 & b_0 & \ldots & 0 \\
. & & . & & & & . \\
. & & & . & & & . \\
. & & & & . & & . \\
0 & 0 & \ldots & 0 & b_m & \ldots & b_0
\end{array}$$

It is a remarkable result that the determinant of this matrix is precisely the resultant of P and Q. Note that the resultant will be zero iff P and Q have a common root- in which case, they have a non-trivial greatest common divisor.

Hence, gaussian elimination can be applied to the Sylvester Matrix, analagous to Euclid's algorithm. If the result is non-zero, the polynomials are co-prime. If a result of zero is obtained, then there is a gcd of interest- with a bit more work, an algorithm can keep track of the cancellations that occur during elimination and in this way the common factors determined. Finally, a system of rescalings exists which allows for fraction-free calculation whilst generating coefficients with predictable common factors, which may therefore be efficiently cancelled. This entire technique is encapsulated in the Dodgson-Bareiss algorithm, and it represents the most efficient way to find the gcd along euclidean lines.

Non-Euclidean techniques- modular GCD calculation

We have seen that polynomial gcd calculation is possible in a fraction-free manner, at the price of intermediate expression swell, which, although reduced by the Bareiss algorithm, can still be horrific. A useful trick would be to bound the size of those intermediate expressions, and this is generally accomplished by the use of modular methods- working modulo a small value such that all expressions fall in a range $0 \ldots n$ or $-p \ldots p$ say.

Two (de)motivating Examples

Sadly, the simplification power of modular mathematics can often erase interesting or important information. There is a further complication with gcds, which is that coefficients of the answer may be greater than any and all of the coefficients present in your original polynomials. Here are some examples that demonstrate these concerns directly.

Problem 1 Consider $P = x^3+x^2-x-1$ and $Q = x^4+x^3+x+1$. Here all the coefficients are 0 or 1. However, when you factorise P and Q you observe $P = (x+1)^2(x-1)$ and $Q = (x+1)^2(x^2-x-1)$. So their gcd is $(x+1)^2$ which expands as x^2+2x+1- we have obtained a larger coefficient. Examples of this form can be created to generate an arbitrarily large coefficient.

Problem 2 Let $P = x-3$ and $Q = x+2$. Then these are clearly coprime- so their gcd is 1. Yet working modulo 5, $Q = P$ so we have a non-trivial gcd of $x+2$, of greater degree than the 'true' gcd.

Solutions to these problems

The first issue, of unexpectedly large coefficients, can be fairly easily resolved- there is an upper bound, the Landau-Mignotte bound, on the absolute value of coefficients in the gcd- generated by a fairly ugly formula involving the degrees of the polynomials and their coefficients. For a precise description, see the literature referenced at the end of this writeup; its existance is sufficient for the discussion that follows.

The second problem needs more work to resolve, but is easy to describe. Given polynomials P and Q, we observe (without proof) that

- degree(gcd((P mod n),(Q mod n))) $\geq$ degree(gcd(P,Q))
- Equality holds in the above only if gcd((P mod n),(Q mod n)) = (gcd(P,Q)) mod n. That is, if the modular image of the gcd is the gcd of the modular images.
- The above will fail only if n divides the resultant of $\frac{P}{G}$, $\frac{Q}{G}$, where G is the true gcd of P and Q.
- The resultant is finite, so has only finitely many prime factors. Hence, if we chose n to be prime, then only finitely many choices of n will be 'bad'.

A 'bad' gcd will be immediately apparent- it won't actually divide both P and Q! On the other hand, a gcd found by modular means has degree at least that of the true gcd, so if it turns out to be a divisor, it is the gcd. So if we just keep generating gcds working modulo a prime, eventually we'll exhaust the bad primes and uncover the true gcd.

Note that in the two algorithms offered below, a gcd calculation is required! This seems circular- but we already have a working yet potentially unwieldy gcd algorithm from the Dodgson-Bareiss approach, which, modulo a prime, won't be so unwieldy after all.

Large prime modular GCD algorithm

Pick p a prime greater than twice the LM (Landau-Mignotte) bound.
Calculate $Qp = gcd(Ap, Bp)$ (where Xp denotes X mod p)
Rewrite Qp over the integers such that its coefficients fall in the range $\frac{-p}{2}$ to $\frac{p}{2}$ (that is, add or subtract p from each coefficient that falls outside the LM bound)
If Qp divides A and divides B, Qp is the true gcd.
If not, repeat from start with the next largest prime.

Many small primes modular GCD algorithm

The above single-prime technique could still yield large coefficients if the LM bound is high; and we only determine if it was a 'good' prime at the very end. Using the Chinese remainder theorem, it is possible to build up to the LM bound in stages, discarding any unlucky primes along the way. By using successive small primes, no modular gcd calculation will be very difficult, and there need not be many such iterations- knowing the answer mod m and n, the Chinese remainder theorem yields an answer mod mn.

```
GCD(A,B)
Define LM= Landau-Mignotte bound of A,B
Define p=firstPrime(A,B)
// p should be chosen so as not to divide the leading coefficients of A and B.

Define C=gcd(Ap,Bp)
*  If degree C = 0, return 1
//NB this assumes monic polynomials; the point is we have a trivial gcd

Define Known=p
Define Result=C

While Known <= 2LM do
  p=nextPrime(A,B,p)
  C=gcd(Ap,Bp)

  If degree C < degree Result, goto *  //all previous primes bad!
  If degree C > degree Result, do nothing
  If degree C = degree Result
      Result=ChineseRemainder(C,Result,p,Known)
      Known=Known*p

end while
```

```
Check Result divides A and B. If not, redo from start, avoiding all primes used so far.
```

Note that an early-exit strategy is possible- if the result after applying the chinese remainder theorem is unchanged, the odds are very good that you have found the true gcd. So the algorithm can be refined by testing for Result dividing A and B whenever this occurs, and halting the loop if successful.

References

CM30070 Computer Algebra, University of Bath- lecture notes, revision notes, and the lecturer's book of the same title. More in-depth information on the specifics of algorithms and bounds can be found in the book, although it is currently out of print. Details at *http://www.bath.ac.uk/ masjhd/DSTeng.html*