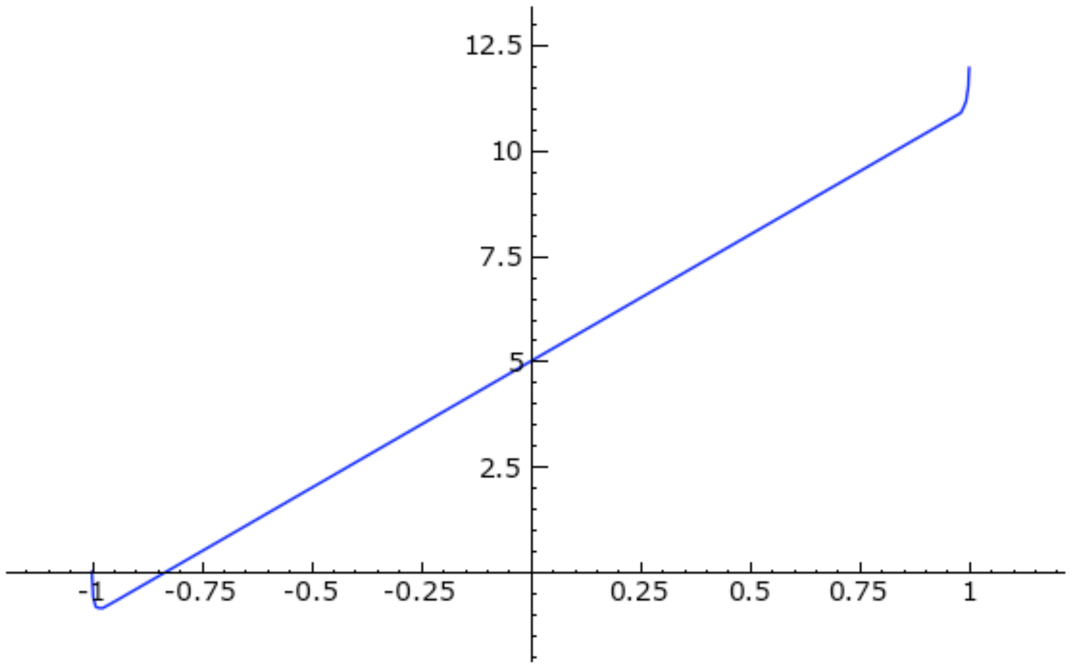
# Computing the Signature $r_1$ and $r_2$ of a Number Field

**February 9, 2009**

Finding all the real roots (e.g., using an iterative algorithm) takes a while.

```
R.<x> = ZZ[]
f = x^198 + 6*x + 5
```

```
plot(f,-1,1)
```



```
time f.roots(RDF)
```

```
    [(-1.0, 1), (-0.833333333333, 1)]
    Time: CPU 0.11 s, Wall: 0.10 s
```

```
time f.roots(RealField(100))
```

```
    [(-1.0000000000000000000000000000, 1),
    (-0.833333333333333336832479080503, 1)]
    Time: CPU 10.45 s, Wall: 10.46 s
```

```
timeit('f.roots(RDF)')
```

```
    5 loops, best of 3: 101 ms per loop
```

In contrast, using ***Sturm sequences*** one can determine the *number* of roots very quickly, with no worries about roundoff error since the algorithm is exact.

```
g = pari('x^198 + 6*x + 5')
```

```
g.polsturm_full()  # number of roots
```

```
    2
```

```
timeit('g.polsturm_full()')
```

```
    625 loops, best of 3: 586 Âµs per loop
```

In this example, on this hardware (MacPro), using Sturm sequences is over 200 times faster.

```
101/.586
```

```
    172.354948805461
```

Our goal is to give a very high-level implementation of an algorithm to compute the signature using Sturm sequences, following Section 4.1.2 of Cohen, GTM 138.

```
def sign(n):
   if n > 0:
       return +1
   elif n < 0:
       return -1
   return 0

def sturm_sig(T):
   """
   INPUT:
       T -- irreducible poly with integer coefficients
   OUTPUT:
       r1, r2 -- signature of T, i.e., number of real
                 and complex-conjugate pairs of roots
   EXAMPLE:
       sage: R.<x> = ZZ['x']
```

```
        sage: f = (x-1)*(x-2)*(x-997)*(x^19+x+1)
        sage: sturm_sig(f)
        (4,9)
        sage: pari(f).polsturm_full()
        4
    """
    Z = ZZ['x']
    T = Z(T)

    # Step 1: Initializations and reductions
    if T.degree() == 0:
        return 0,0
    A = T // T.content()
    Tprime = T.derivative()  # slow because not optimized
    B = Tprime // Tprime.content()
    g = 1
    h = 1
    s = sign(A.leading_coefficient())
    n = A.degree()
    t = (-1)^(n-1) * s
    r1 = 1
    while True:
        # Step 2: Pseudo division
        delta = A.degree() - B.degree()
        Q, R, d = A.pseudo_divrem(B)  # I had to add this
                                      # to Sage -- see trac #5222
        if d != delta+1:
            scale = B.leading_coefficient()^(delta + 1 - d)
            Q, R = scale*Q, scale*R
        if R == 0:
            raise ValueError, "input T is not square free"
        if B.leading_coefficient() > 0 or delta % 2:
            R = -R

        # Step 3: Use Sturm
        sR = sign(R.leading_coefficient())
        if sR != s:
            s  = -s
            r1 = r1 - 1
        if sR != (-1)^R.degree() * t:
            t = -t
            r1 = r1 + 1

        # Step 4: Finished
        if R.degree() == 0:
            return r1, (n-r1)//2
        else:
            A = B
            B = R//(g*h^delta)
            g = abs(A.leading_coefficient())
            h = h^(1-delta) * g^delta
```