12
Non-linear Equations

This chapter explains how to *solve* a non-linear equation using Sage. We first study polynomial equations and show the limitations of the search for exact solutions. We then describe some classical numerical solving methods, while indicating the numerical algorithms implemented in Sage.

12.1 Algebraic Equations

An algebraic equation is an equation of the form $p(x) = 0$, where p is a polynomial in one variable with coefficients in an integral domain A. We say that an element $\alpha \in A$ is a *root* of the polynomial p if $p(\alpha) = 0$.

Let α be an element of A. Polynomial long division of p by $x - \alpha$ yields a constant polynomial r such that

$$p = (x - \alpha)q + r.$$

Upon evaluating this equation at $x = \alpha$, we get $r = p(\alpha)$. So the polynomial $x - \alpha$ divides p if and only if α is a root of p. This remark leads to the notion of *multiplicity* of a root α of the polynomial p: it is the largest integer m such that $(x - \alpha)^m$ divides p. We note that the sum of the multiplicities of the roots of p is less than or equal to the degree of p.

12.1.1 The Method `Polynomial.roots()`

To solve the algebraic equation $p(x) = 0$ means to identify the roots of the polynomial p and their multiplicities. The method `Polynomial.roots()` finds the roots of a polynomial. It takes up to three parameters, all of them optional. The parameter `ring` indicates the ring in which to search for the roots. If we do not

give a value to this parameter, Sage uses the coefficient ring of the polynomial. The boolean parameter `multiplicities` specifies whether the information returned by `Polynomial.roots()` should consist of both roots and multiplicities. The parameter `algorithm` indicates which algorithm to use; the possible values are described below (see §12.2.2).

```
sage: R.<x> = PolynomialRing(RealField(prec=10))
sage: p = 2*x^7 - 21*x^6 + 64*x^5 - 67*x^4 + 90*x^3 \
....: + 265*x^2 - 900*x + 375
sage: p.roots()
[(-1.7, 1), (0.50, 1), (1.7, 1), (5.0, 2)]
sage: p.roots(ring=ComplexField(10), multiplicities=False)
[-1.7, 0.50, 1.7, 5.0, -2.2*I, 2.2*I]
sage: p.roots(ring=RationalField())
[(1/2, 1), (5, 2)]
```

12.1.2 Representation of Numbers

We recall how to denote commonly-used rings with Sage (see §5.2). Integers are represented by objects of the class `Integer`; conversions are performed using the *parent* `ZZ` or the function `IntegerRing()`, which returns the object `ZZ`. Similarly, rational numbers are represented by objects of the class `Rational`; the parent of these objects is the object `QQ`, which is returned by the function `RationalField()`. In both cases Sage uses the arbitrary precision library GMP. Without diving into the inner workings of this library, the integers used by Sage have arbitrary size, limited only by the available memory of the machine on which the software is run.

Several approximate representations of real numbers are available (see chapter 11). There is `RealField()` for floating point numbers with a given precision and, in particular, `RR` for 53-bit precision. Representations using machine double precision are afforded by `RDF` and the function `RealDoubleField()`. There is also the class `RIF` and the function `RealIntervalField()` in which a real number is represented by an interval containing it; the endpoints of this interval are floating point numbers.

The analogous representations for complex numbers are: `CC`, `CDF`, and `CIF`. Again, to which object is associated a function, namely `ComplexField()`, `ComplexDoubleField()`, and `ComplexIntervalField()`.

The computations performed by `Polynomial.roots()` are exact or approximate depending on the representation of the polynomial's coefficients and (if specified) the value of the parameter `ring`: for instance, given `ZZ` or `QQ`, the computation is exact; given `RR` it is approximate. In the second part of this chapter we describe the algorithm used for the computation of the roots and specify the role of the parameters `ring` and `algorithm` (see §12.2).

Solving algebraic equations is intimately related to the notion of number. The *splitting field* of a nonconstant polynomial p is the smallest field extension of the coefficient field in which p is a product of polynomials of degree 1; one can prove that such an extension always exists. In Sage, we can construct the splitting field

of an irreducible polynomial with the method `Polynomial.root_field()`. We can then compute with the roots *implicitly* contained in the splitting field.

```
sage: R.<x> = PolynomialRing(QQ, 'x')
sage: p = x^4 + x^3 + x^2 + x + 1
sage: K.<alpha> = p.root_field()
sage: p.roots(ring=K, multiplicities=None)
[alpha, alpha^2, alpha^3, -alpha^3 - alpha^2 - alpha - 1]
sage: alpha^5
1
```

12.1.3 The Fundamental Theorem of Algebra

The splitting field of the polynomial with real coefficients $x^2 + 1$ is precisely the field of complex numbers. It is remarkable that every nonconstant polynomial with complex coefficients has at least one complex root: this is the content of the *Fundamental Theorem of Algebra*. Therefore every nonconstant complex polynomial is a product of polynomials of degree 1. We noted above that the sum of the multiplicities of the roots of a polynomial p is less than or equal to the degree of p. In other words, every polynomial equation of degree n with complex coefficients has n complex roots, counted with multiplicity.

Let's see how the method `Polynomial.roots()` can be used to illustrate this result. In the following example, we construct the ring of polynomials with real coefficients (represented by floating point numbers with a precision of 53 bits). Then a polynomial of degree at most 15 is picked randomly from this ring. Finally we add the multiplicities of the complex roots computed with the method `Polynomial.roots()` and we compare this sum to the degree of the polynomial.

```
sage: R.<x> = PolynomialRing(RR, 'x')
sage: d = ZZ.random_element(1, 15)
sage: p = R.random_element(d)
sage: p.degree() == sum(r[1] for r in p.roots(CC))
True
```

12.1.4 Distribution of the Roots

We give a curious illustration of the power of the method `Polynomial.roots()`: we plot all the points in the complex plane whose corresponding complex number is a root of a polynomial of degree 12 and with coefficients in the set $\{-1, 1\}$. The choice of degree is rather arbitrary, made in order to obtain a sufficiently detailed plot in a short amount of time. The usage of approximate values for the complex numbers is also motivated by performance reasons. (see §13).

```
sage: def build_complex_roots(degree):
....:     R.<x> = PolynomialRing(CDF, 'x')
....:     v = []
....:     for c in cartesian_product([[-1, 1]] * (degree + 1)):
....:         v.extend(R(c).roots(multiplicities=False))
....:     return v
```

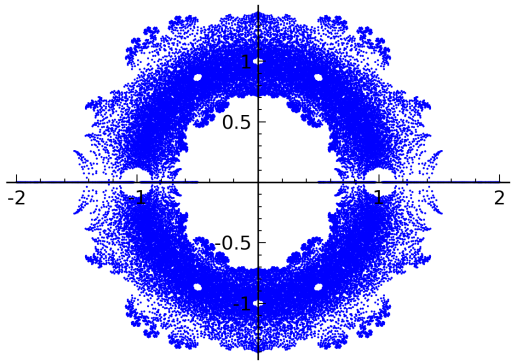

FIGURE 12.1 – Distribution of roots of all polynomials of degree 12 with coefficients -1 or 1.

```
sage: data = build_complex_roots(12)
sage: g = points(data, pointsize=1, aspect_ratio=1)
```

12.1.5 Solvability in Radicals

In certain cases it is possible to compute the exact value of the roots of a polynomial. This occurs for instance when we can express the roots in terms of the coefficients of the polynomial and using radicals (square roots, cubic roots, etc.). We then say that the polynomial is *solvable in radicals*.

To solve in radicals with Sage, we must work with objects of the class `Expression` which represent symbolic expressions. We have seen that integers represented by objects of the class `Integer` have the same *parent*, namely the object `ZZ`. Similarly, the objects of the class `Expression` have the same parent: the object `SR` (short for *Symbolic Ring*); this allows to convert into the class `Expression`.

Quadratic equations.

```
sage: a, b, c, x = var('a, b, c, x')
sage: p = a * x^2 + b * x + c
sage: type(p)
<type 'sage.symbolic.expression.Expression'>
sage: p.parent()
Symbolic Ring
sage: p.roots(x)
[(-1/2*(b + sqrt(-4*a*c + b^2))/a, 1),
(-1/2*(b - sqrt(-4*a*c + b^2))/a, 1)]
```

Degree strictly bigger than 2. Algebraic equations of degree 3 and 4 are solvable in radicals. However, it is generally impossible to solve in radicals

polynomial equations of degree at least 5 (see §7.3.4). This impossibility leads to investigate numerical solution methods (see §12.2).

```
sage: a, b, c, d, e, f, x = var('a, b, c, d, e, f, x')
sage: p = a*x^5+b*x^4+c*x^3+d*x^2+e*x+f
sage: p.roots(x)
Traceback (most recent call last):
...
RuntimeError: no explicit roots found
```

We use Sage to illustrate a method for solving equations of degree 3 over the field of complex numbers. We start by showing that the general equation of degree 3 can be reduced to the form $x^3 + px + q = 0$.

```
sage: x, a, b, c, d = var('x, a, b, c, d')
sage: P = a * x^3 + b * x^2 + c * x + d
sage: alpha = var('alpha')
sage: P.subs(x = x + alpha).expand().coefficient(x, 2)
3*a*alpha + b
sage: P.subs(x = x - b / (3 * a)).expand().collect(x)
a*x^3 - 1/3*(b^2/a - 3*c)*x - 1/3*b*c/a + 2/27*b^3/a^2 + d
```

To obtain the roots of an equation of the form $x^3+px+q = 0$, we set $x = u+v$.

```
sage: p, q, u, v = var('p, q, u, v')
sage: P = x^3 + p * x + q
sage: P.subs(x = u + v).expand()
u^3 + 3*u^2*v + 3*u*v^2 + v^3 + p*u + p*v + q
```

Let's assume the last expression is zero. We then note that $u^3 + v^3 + q = 0$ is equivalent to $3uv + p = 0$; moreover, if these equalities hold, u^3 and v^3 satisfy an equation of degree two: $(X - u^3)(X - v^3) = X^2 - (u^3 + v^3)X + (uv)^3 = X^2 + qX - p^3/27$.

```
sage: P.subs({x: u + v, q: -u^3 - v^3}).factor()
(u + v)*(3*u*v + p)
sage: P.subs({x: u+v, q: -u^3 - v^3, p: -3 * u * v}).expand()
0
sage: X = var('X')
sage: solve([X^2 + q*X - p^3 / 27 == 0], X, solution_dict=True)
[{X: -1/2*q - 1/18*sqrt(12*p^3 + 81*q^2)},
 {X: -1/2*q + 1/18*sqrt(12*p^3 + 81*q^2)}]
```

The solutions of the equation $x^3 + px + q = 0$ are therefore the sums $u + v$ where u and v are the cube roots of

$$-\frac{\sqrt{4p^3 + 27q^2}\sqrt{3}}{18} - \frac{q}{2} \quad \text{et} \quad \frac{\sqrt{4p^3 + 27q^2}\sqrt{3}}{18} - \frac{q}{2}$$

satisfying $3uv + p = 0$.

12.1.6 The Method `Expression.roots()`

The preceding examples use the method `Expression.roots()`. This method returns a list of exact roots, not guaranteed to be complete. Among the optional parameters of this method, we find the parameters `ring` and `multiplicities` already seen for the method `Polynomial.roots()`. It is important to keep in mind that the method `Expression.roots()` is not restricted to polynomial expressions.

```
sage: e = sin(x) * (x^3 + 1) * (x^5 + x^4 + 1)
sage: roots = e.roots(); len(roots)
9
sage: roots
[(0, 1),
 (-1/2*(I*sqrt(3) + 1)*(1/18*sqrt(3)*sqrt(23) - 1/2)^(1/3)
 - 1/6*(-I*sqrt(3) + 1)/(1/18*sqrt(3)*sqrt(23) - 1/2)^(1/3), 1),
 (-1/2*(-I*sqrt(3) + 1)*(1/18*sqrt(3)*sqrt(23) - 1/2)^(1/3)
 - 1/6*(I*sqrt(3) + 1)/(1/18*sqrt(3)*sqrt(23) - 1/2)^(1/3), 1),
 ((1/18*sqrt(3)*sqrt(23) - 1/2)^(1/3) + 1/3/(1/18*sqrt(3)*sqrt(23)
 - 1/2)^(1/3), 1),
 (-1/2*I*sqrt(3) - 1/2, 1), (1/2*I*sqrt(3) - 1/2, 1),
 (1/2*I*(-1)^(1/3)*sqrt(3) - 1/2*(-1)^(1/3), 1),
 (-1/2*I*(-1)^(1/3)*sqrt(3) - 1/2*(-1)^(1/3), 1), ((-1)^(1/3), 1)]
```

If the parameter `ring` is not specified, the method `roots()` of the class `Expression` delegates the computation of the roots to Maxima, which tries to factor the expression, then solves in radicals each factor of degree strictly less than 5. When the parameter `ring` is specified, the expression is converted into an object of the class `Polynomial` whose coefficients have as parent the object identified by the parameter `ring`; then the result of the method `Polynomial.roots()` is returned. We will describe the algorithm used in this case below (see §12.2.2).

It is also possible to compute with implicit roots, which we can access via the objects `QQbar` and `AA` representing the field of algebraic numbers (see §7.3.2).

Elimination of multiple roots Given a polynomial p with multiple roots, it is possible to construct a polynomial with simple roots (that is, of multiplicity 1), identical to those of p. Hence, when computing the roots of a polynomial, we can assume that these roots are simple. We first prove the existence of the polynomial with simple roots and then show how to construct it. This will give a new illustration of the method `Expression.roots()`.

Let α be a root of the polynomial p with multiplicity $m > 1$. It is a root of the derivative p' with multiplicity $m - 1$. In fact, if $p = (x - \alpha)^m q$ then $p' = (x - \alpha)^{m-1}(mq + (x - \alpha)q')$.

```
sage: alpha, m, x = var('alpha, m, x'); q = function('q')(x)
sage: p = (x - alpha)^m * q
sage: p.derivative(x)
m*(-alpha + x)^(m - 1)*q(x) + (-alpha + x)^m*D[0](q)(x)
sage: simplify(p.derivative(x)(x=alpha))
```

```
0
```

Therefore the gcd of p and p' is the product $\prod_{\alpha\in\Gamma}(x-\alpha)^{m_\alpha-1}$, where Γ is the set of roots of p with multiplicity strictly greater than 1, and m_α is the multiplicity of the root α. If d denotes this gcd, then the quotient p/d has the desired properties.

Note that the degree of the quotient of p by d is strictly less than the degree of p. In particular, if this degree is strictly less than 5, it is possible to express the roots in terms of radicals. The following example illustrates this for a polynomial of degree 13 with rational coefficients.

```
sage: R.<x> = PolynomialRing(QQ, 'x')
sage: p = 128 * x^13 - 1344 * x^12 + 6048 * x^11 - 15632 * x^10 \
....: + 28056 * x^9 - 44604 * x^8 + 71198 * x^7 - 98283 * x^6   \
....: + 105840 * x^5 - 101304 * x^4 + 99468 * x^3 - 81648 * x^2 \
....: + 40824 * x - 8748
sage: d = gcd(p, p.derivative())
sage: (p // d).degree()
4
sage: roots = SR(p // d).roots(multiplicities=False)
sage: roots
[1/2*I*2^(1/3)*sqrt(3) - 1/2*2^(1/3),
-1/2*I*2^(1/3)*sqrt(3) - 1/2*2^(1/3), 2^(1/3), 3/2]
sage: [QQbar(p(alpha)).is_zero() for alpha in roots]
[True, True, True, True]
```

12.2 Numerical Solution

There is a traditional dichotomy in mathematics between the *discrete* and the *continuous*. Numerical analysis bridges this gap to some extent: a major aspect of numerical analysis is the study of questions about real numbers (firmly part of the continuous domain) via an experimental point of view, often assisted by a computer, which belongs to the discrete domain.

Regarding the solution of non-linear equations, numerous natural questions arise beyond the computation of approximate values: how many real roots does a given equation have? How many imaginary, positive, or negative roots are there?

In this section we start to answer such questions in the special case of algebraic equations. Then we describe some of the methods for computing approximate solutions of a non-linear equation.

12.2.1 Location of solutions of algebraic equations

Descartes' rule. Descartes' rule of signs is the following: the number of positive roots of a polynomial with real coefficients is less than or equal to the number of sign changes in the sequence of coefficients of the polynomial.

```
sage: R.<x> = PolynomialRing(RR, 'x')
```

```
sage: p = x^7 - 131/3*x^6 + 1070/3*x^5 - 2927/3*x^4 \
....: + 2435/3*x^3 - 806/3*x^2 + 3188/3*x - 680
sage: l = [c for c in p.coefficients(sparse=False) if not c.is_zero()]
sage: sign_changes = [l[i]*l[i + 1] < 0 for i in range(len(l)-1)].count(True)
sage: real_positive_roots = \
....: sum([alpha[1] if alpha[0] > 0 else 0 for alpha in p.roots()])
sage: sign_changes, real_positive_roots
(7, 5)
```

Indeed, let p be a degree d polynomial with real coefficients and let p' be its derivative. We denote by u and u' the sequences of the signs of the coefficients of the polynomials p and p': we have $u_i = \pm 1$ depending on whether the degree i coefficient of p is positive or negative. The sequence u' is a simple truncation of u: we have $u'_i = u_{i+1}$ for $0 \leq i < d$. If follows that the number of sign changes of the sequence u is at most one plus the number of sign changes of the sequence u'.

On the other hand, the number of positive roots of p is at most equal to one plus the number of positive roots of p': any interval whose endpoints are roots of p always contains a root of p'.

As Descartes' rule holds for a degree 1 polynomial, the above observations show that it also holds for a degree 2 polynomial, etc.

Moreover, the difference between the number of positive roots and the number of sign changes in the sequence of coefficients is always even.

Isolating the real roots of a polynomial. Given a polynomial with real coefficients, we have seen that it is possible to bound from above the number of roots contained in the interval $[0, +\infty)$. More generally, there are methods for finding the number of roots contained in a given interval.

One such result is Sturm's Theorem. Let p be a degree d polynomial with real coefficients and let $[a, b]$ be an interval. We construct recursively a sequence of polynomials. We start with $p_0 = p$ and $p_1 = p'$; then p_{i+2} is the negative of the remainder of the polynomial division of p_i by p_{i+1}. By evaluating this sequence of polynomials at the points a and b we get two finite real sequences $(p_0(a), \ldots, p_d(a))$ and $(p_0(b), \ldots, p_d(b))$. Sturm's Theorem is the following: if p has simple roots, $p(a) \neq 0$ and $p(b) \neq 0$, then the number of roots of p contained in the interval $[a, b]$ equals the number of sign changes of the sequence $(p_0(a), \ldots, p_d(a))$ minus the number of sign changes of the sequence $(p_0(b), \ldots, p_d(b))$.

Here is how we can implement this theorem in Sage.

```
sage: def count_sign_changes(p):
....:     l = [c for c in p if not c.is_zero()]
....:     changes = [l[i]*l[i + 1] < 0 for i in range(len(l) - 1)]
....:     return changes.count(True)
```

```
sage: def sturm(p, a, b):
....:     assert p.degree() > 2
....:     assert not (p(a) == 0)
....:     assert not (p(b) == 0)
```

```
....:     assert a <= b
....:     remains = [p, p.derivative()]
....:     for i in range(p.degree() - 1):
....:         remains.append(-(remains[i] % remains[i + 1]))
....:     evals = [[], []]
....:     for q in remains:
....:         evals[0].append(q(a))
....:         evals[1].append(q(b))
....:     return count_sign_changes(evals[0]) \
....:            - count_sign_changes(evals[1])
```

Here is an example of usage of this function `sturm()`.

```
sage: R.<x> = PolynomialRing(QQ, 'x')
sage: p = (x - 34) * (x - 5) * (x - 3) * (x - 2) * (x - 2/3)
sage: sturm(p, 1, 4)
2
sage: sturm(p, 1, 10)
3
sage: sturm(p, 1, 200)
4
sage: p.roots(multiplicities=False)
[34, 5, 3, 2, 2/3]
sage: sturm(p, 1/2, 35)
5
```

12.2.2 Iterative approximation methods

Approximation : Gén. au sing. Opération par laquelle on tend à se rapprocher de plus en plus de la valeur réelle d'une quantité ou d'une grandeur sans y parvenir rigoureusement.
Trésor de la Langue Française

In this section we illustrate various ways of approximating the solutions of a non-linear equation $f(x) = 0$. There are essentially two approaches to such approximations. The most efficient algorithm mixes the two approaches.

The first approach constructs a sequence of nested intervals that contain a solution of the equation. We control the precision and convergence is guaranteed, but the convergence speed is not always good.

The second approach starts with a given approximate value of one of the solutions of the equation. If the local behaviour of the function f is sufficiently regular, we can compute a new, more accurate approximation to the solution. By recurrence, we get a sequence of approximate values. This approach assumes that we know a first approximation of the desired number. Moreover, its performance depends on a good local behaviour of the function f: we cannot dictate *a priori* the precision of the answer; worse, the convergence of the sequence of approximations is not necessarily guaranteed.

Throughout this section, we consider a non-linear equation $f(x) = 0$ where f is a numerical function defined on the interval $[a, b]$ and continuous on this interval. We assume that the values of f at the endpoints of the interval $[a, b]$ are non-zero and of opposite sign: in other words, the product $f(a)f(b)$ is strictly negative. The continuity of f guarantees the existence of at least one solution of the equation $f(x) = 0$ in the interval $[a, b]$.

For each of the methods we discuss, we experiment with the following function.

```
sage: f(x) = 4 * sin(x) - exp(x) / 2 + 1
sage: a, b = RR(-pi), RR(pi)
sage: bool(f(a) * f(b) < 0)
True
```

We should note that, for this function, the command `solve` is not useful.

```
sage: solve(f(x) == 0, x)
[sin(x) == 1/8*e^x - 1/4]
```

```
sage: f.roots()
Traceback (most recent call last):
 ...
RuntimeError: no explicit roots found
```

The algorithms for solving non-linear equations can be very time-consuming: it is advisable to take certain precautions before proceeding. In particular, we should ensure that solutions exist by studying the continuity and differentiability of the function in question, as well as its sign changes; plotting the graph of the function can be useful here (see Chapter 4).

The bisection method. This method is based on the first approach: construct a sequence of nested intervals, each of which contains a solution of the equation $f(x) = 0$.

We find the midpoint c of the interval $[a, b]$. Suppose that $f(c) \neq 0$. Either $f(a)f(c)$ is negative, so that the interval $[a, c]$ must contain a solution of the equation; or $f(c)f(b)$ is negative, so that the interval $[c, b]$ contains a solution of the equation. Therefore we can construct an interval containing a solution, and whose length is half the length of the interval $[a, b]$. By iterating this construction we obtain a sequence of intervals with the expected properties. To implement this approach, we define a Python function `intervalgen` as follows.

```
sage: def phi(s, t): return (s + t) / 2
sage: def intervalgen(f, phi, s, t):
....:     msg = 'Wrong arguments: f({0})*f({1})>=0)'.format(s, t)
....:     assert (f(s) * f(t) < 0), msg
....:     yield s
....:     yield t
....:     while True:
....:         u = phi(s, t)
....:         yield u
```

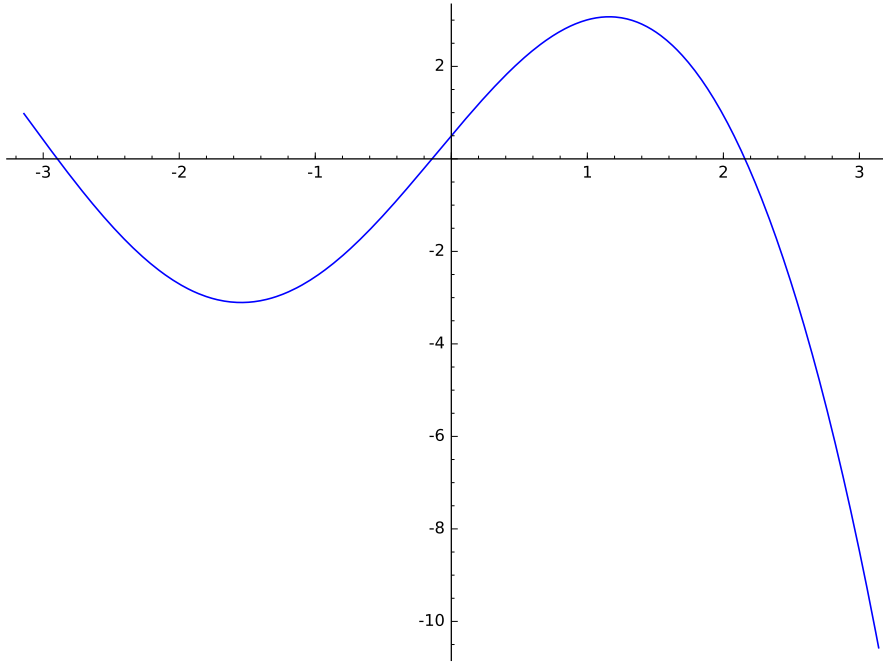

FIGURE 12.2 – Graph of the function f.

```
....:         if f(u) * f(s) < 0:
....:             t = u
....:         else:
....:             s = u
```

The definition of this function deserves explanation. The keyword `yield` in the definition of `intervalgen` turns it into a *generator* (see §15.2.4). When the method `next()` of a generator is called, if the interpreter sees the keyword `yield`, all the local data are saved, the execution is interrupted and the expression immediately at the right of the keyword is returned. The following call of the method `next()` restores the local data that was saved before the interruption and continues from the line following the keyword `yield`. The usage of the keyword `yield` inside an infinite loop (`while True:`) allows the implementation of a recursive sequence via a syntax that resembles closely its mathematical definition. It is possible to stop the execution completely by using the keyword `return`.

The parameter `phi` is a function that specifies the approximation method. For the bisection method, this function computes the midpoint of an interval. In order to test other iterative approximation methods that also use nested intervals, we need to give a new definition of the function `phi` and can then use the function `intervalgen` to construct the corresponding generator.

The parameters `s` and `t` of the function specify the endpoints of the first interval. The call to `assert` verifies that the function f changes sign between

the endpoints of this interval; as we have seen, this guarantees the existence of a solution.

The first two values of the generator are the values of the parameters `s` and `t`. The third value is the midpoint of the corresponding interval. The parameters `s` and `t` then represent the endpoints of the last interval computed. After evaluating f at the midpoint of this interval, we change one of the endpoints of the interval in such a way that the new interval still contains a solution. We agree to take as approximation of the desired solution the midpoint of the last interval computed.

We experiment with the chosen example: here are three approximations obtained by the bisection method applied to the interval $[-\pi, \pi]$.

```
sage: a, b
(-3.14159265358979, 3.14159265358979)
sage: bisection = intervalgen(f, phi, a, b)
sage: bisection.next()
-3.14159265358979
sage: bisection.next()
3.14159265358979
sage: bisection.next()
0.000000000000000
```

In order to compare the different approximation methods, it is useful to automate the computation of approximate solutions to the equation $f(x) = 0$ using the generators defined in Sage for each of the methods. This mechanism should allow us to control the precision and the maximum number of iterations. This role is taken by the function `iterate` defined below.

```
sage: from types import GeneratorType, FunctionType
sage: def checklength(u, v, w, prec):
....:     return abs(v - u) < 2 * prec
sage: def iterate(series, check=checklength, prec=10^-5, maxit=100):
....:     assert isinstance(series, GeneratorType)
....:     assert isinstance(check, FunctionType)
....:     niter = 2
....:     v, w = series.next(), series.next()
....:     while niter <= maxit:
....:         niter += 1
....:         u, v, w = v, w, series.next()
....:         if check(u, v, w, prec):
....:             print 'After {0} iterations: {1}'.format(niter, w)
....:             return
....:     print 'Failed after {0} iterations'.format(maxit)
```

The parameter `series` must be a generator. We keep the last three values of this generator to verify convergence. This is the role of the parameter `check`: a function that may or may not stop the iterations. By default the function `iterate` uses the function `checklength` which stops the iterations if the last interval computed has length strictly less than twice the parameter `prec`; this

guarantees that the value computed by bisection approximates the solution with an error strictly less than `prec`.

An exception is raised once the number of iterations is larger than the parameter `maxit`.

```
sage: bisection = intervalgen(f, phi, a, b)
sage: iterate(bisection)
After 22 iterations: 2.15847275559132
```

Exercise 43. Modify the function `intervalgen` so that the generator stops if one of the endpoints of the interval is a solution.

Exercise 44. Use the functions `intervalgen` and `iterate` to compute the approximation to a solution of the equation $f(x) = 0$ using nested intervals, each interval being obtained by dividing the previous one at a randomly chosen point.

The false position method. This method also uses the first approach: construct a sequence of nested intervals that contain a solution of the equation $f(x) = 0$. However, it employs linear interpolation of the function f for dividing each interval.

More precisely, to divide the interval $[a, b]$, we consider the segment joining the two points on the graph of f with x-coordinates a and b. As $f(a)$ and $f(b)$ have opposite signs, this segment intersects the x-axis, thereby dividing the interval $[a, b]$ into two subintervals. As in the bisection method, we identify the subinterval containing a solution by computing the value of f at the common point of the two subintervals.

The line through the points $(a, f(a))$ and $(b, f(b))$ has equation

$$y = \frac{f(b) - f(a)}{b - a}(x - a) + f(a). \tag{12.1}$$

Since $f(b) \neq f(a)$, this line intersects the x-axis in the point with x-coordinate

$$a - f(a)\frac{b - a}{f(b) - f(a)}.$$

We can therefore test this method as follows.

```
sage: phi(s, t) = s - f(s) * (t - s) / (f(t) - f(s))
sage: falsepos = intervalgen(f, phi, a, b)
sage: iterate(falsepos)
After 8 iterations: -2.89603757331027
```

It is important to note that the sequences constructed by the bisection and false position methods do not necessarily converge to the same solutions. By shrinking the interval of study we recover the positive solutions obtained via the bisection method.

```
sage: a, b = RR(pi/2), RR(pi)
sage: phi(s, t) = t - f(t) * (s - t) / (f(s) - f(t))
sage: falsepos = intervalgen(f, phi, a, b)
```

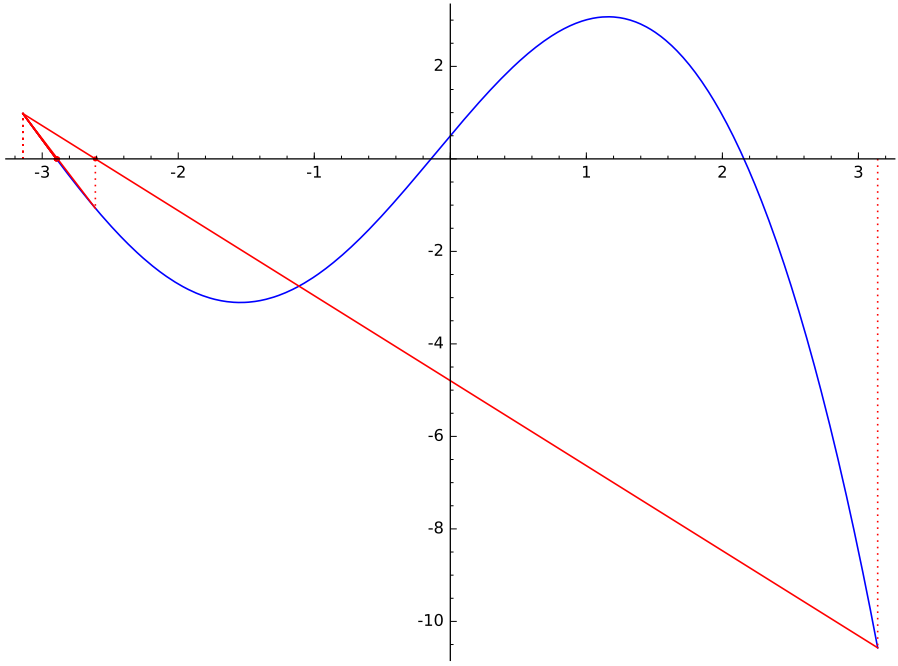

FIGURE 12.3 – The false position method on $[-\pi, \pi]$.

```
sage: phi(s, t) = (s + t) / 2
sage: bisection = intervalgen(f, phi, a, b)
sage: iterate(falsepos)
After 15 iterations: 2.15846441170219
sage: iterate(bisection)
After 20 iterations: 2.15847275559132
```

Newton's method Like the false position method, Newton's method uses a linear approximation of the function f. From a graphical point of view, we consider a tangent to the graph of f as approximating this graph.

We assume that f is differentiable and the derivative f' has the same sign in the interval $[a, b]$; hence f is monotone. We also assume that f changes sign in the interval $[a, b]$. The equation $f(x) = 0$ then has a unique solution in this interval; we denote it by α.

If the sequence u converges[1], then its limit ℓ satisfies $\ell = \ell - f(\ell)/f'(\ell)$, hence $f(\ell) = 0$; the limit is therefore equal to α, the solution of the equation $f(x) = 0$.

So that our example satisfies the monotonicity hypotheses, we have to shrink the interval of study.

[1] A theorem of L. Kantorovich gives a sufficient condition for the convergence of Newton's method.

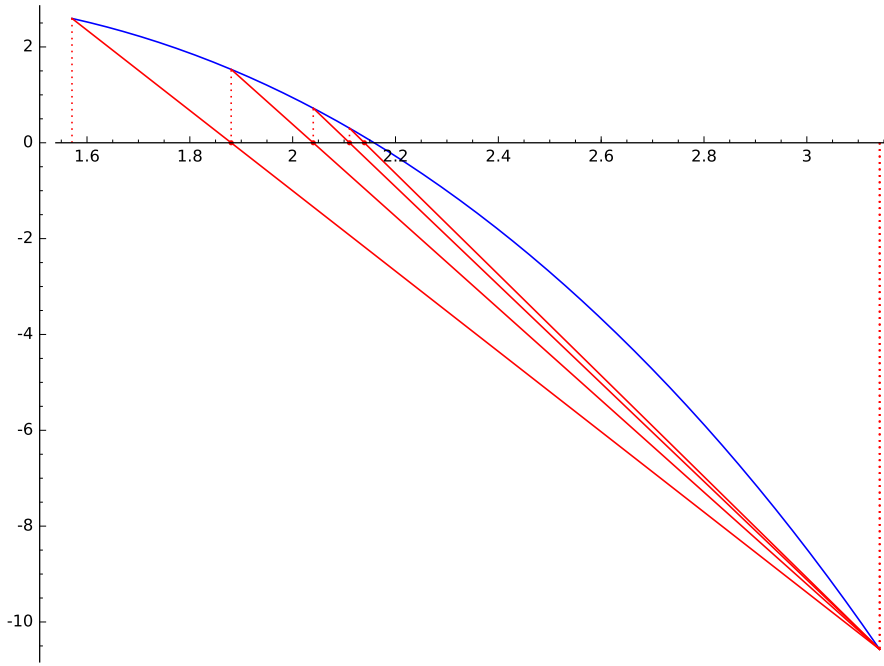

FIGURE 12.4 – The false position method on $[\pi/2, \pi]$.

```
sage: f.derivative()
x |--> -1/2*e^x + 4*cos(x)
sage: a, b = RR(pi/2), RR(pi)
```

We define a Python generator `newtongen` representing the recursive sequence that we defined above. Then we define a new convergence test `checkconv` that stops the iterations if the last two computed terms are sufficiently close; of course this test does not guarantee the convergence of the sequence of approximations.

```
sage: def newtongen(f, u):
....:     while True:
....:         yield u
....:         u -= f(u) / f.derivative()(u)
sage: def checkconv(u, v, w, prec):
....:     return abs(w - v) / abs(w) <= prec
```

We can now test Newton's method on our example.

```
sage: iterate(newtongen(f, a), check=checkconv)
After 6 iterations: 2.15846852566756
```

The secant method The computation of the derivative in Newton's method can be expensive. It is possible to replace it by a linear interpolation: given two

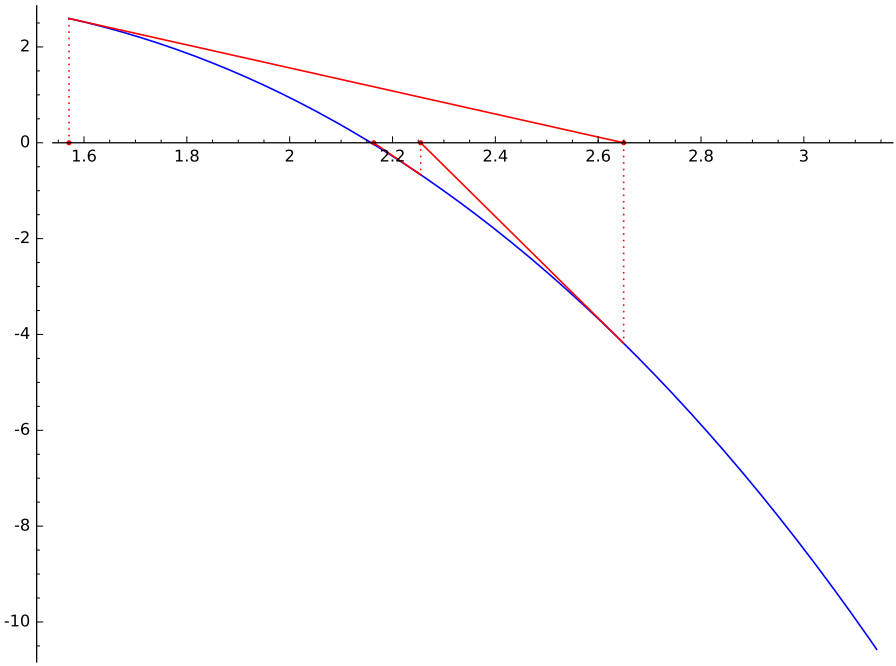

FIGURE 12.5 – Newton's method.

approximations to the solution, hence two points on the graph of f, if the line through these two points intersects the x-axis, we take the x-coordinate of the intersection point as the new approximation. To start the construction, and when the two lines are parallel, we use Newton's method.

This gives rise to the same iterative formula as the false position method, but applied at different points. Contrary to the false position method, the secant method does not provide an interval that contains a solution.

We define a Python generator that implements this method.

```
sage: def secantgen(f, a):
....:     yield a
....:     estimate = f.derivative()(a)
....:     b = a - f(a) / estimate
....:     yield b
....:     while True:
....:         fa, fb = f(a), f(b)
....:         if fa == fb:
....:             estimate = f.derivative()(a)
....:         else:
....:             estimate = (fb - fa) / (b - a)
....:         a = b
....:         b -= fb / estimate
```

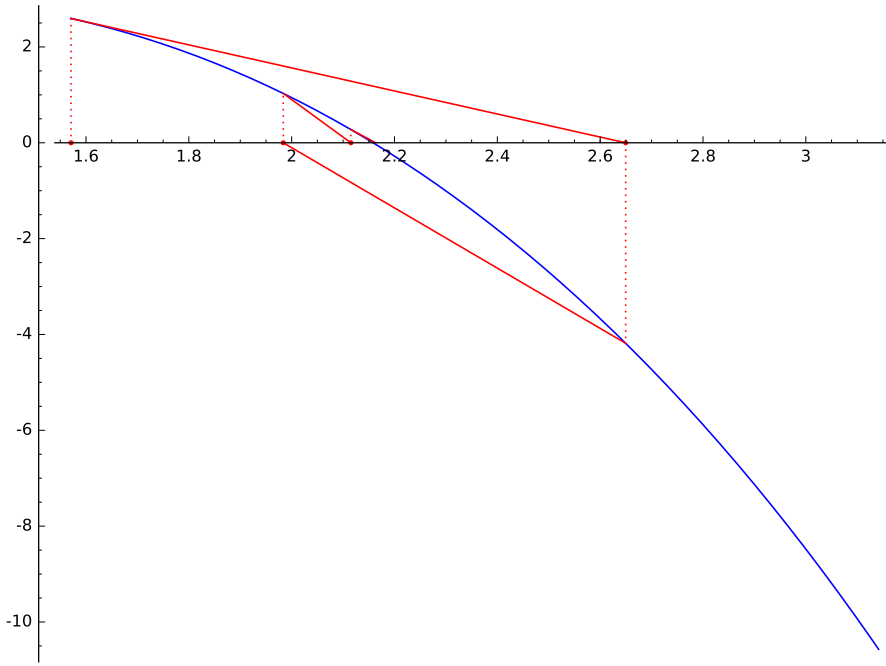

FIGURE 12.6 – The secant method.

```
....:         yield b
```

We can now test the secant method on our example.

```
sage: iterate(secantgen(f, a), check=checkconv)
After 8 iterations: 2.15846852557553
```

Muller's method It is possible to extend the secant method by replacing f by polynomial approximations of any degree. For instance, Muller's[2] method uses quadratic approximations.

Suppose we have constructed three approximations r, s, and t of the solution to the equation $f(x) = 0$. We consider the Lagrange interpolation polynomial defined by the three points on the graph of f with x-coordinates r, s, and t. It is a second-degree polynomial. We take as new approximation the root of this polynomial that is closest to t. The first three terms of the sequence are just taken to be a, b, and $(a + b)/2$.

We should note that the roots of the polynomials—and hence the computed approximations—can be complex numbers.

[2]This is David E. Muller, also known for inventing Reed-Muller codes, and not Jean-Michel Muller mentioned in Chapter 11.

The implementation of this method in Sage is not difficult; it can be done similarly to the secant method. However, our implementation uses a data structure that is better suited to enumerating the terms of a recursive sequence.

```
sage: from collections import deque
sage: basering = PolynomialRing(CC, 'x')
sage: def quadraticgen(f, r, s):
....:     t = (r + s) / 2
....:     yield t
....:     points = deque([(r,f(r)), (s,f(s)), (t,f(t))], maxlen=3)
....:     while True:
....:         pol = basering.lagrange_polynomial(points)
....:         roots = pol.roots(ring=CC, multiplicities=False)
....:         u = min(roots, key=lambda x: abs(x - points[2][0]))
....:         points.append((u, f(u)))
....:         yield points[2][0]
```

The module `collections` of the Python standard library provides several data structures. In `quadraticgen`, the class `deque` is used to store the last approximations computer. A `deque` object stores data up to the limit `maxlen` specified at its creation; here the maximum number of stored data is the recurrence order of the sequence of approximations. Once a `deque` object reaches its maximum storage capacity, the method `deque.append()` inserts new data according to the rule "first in, first out".

Note that the iterations of this method do not require the computation of derivatives. Moreover, each iteration only requires one evaluation of the function f.

```
sage: generator = quadraticgen(f, a, b)
sage: iterate(generator, check=checkconv)
After 5 iterations: 2.15846852554764
```

Back to polynomials. We return to the situation studied at the beginning of the chapter: compute the roots of a polynomial P with real coefficients. We assume that P is monic:

$$P = a_0 + a_1 x + \ldots + a_{d-1} x^{d-1} + x^d.$$

It is easy to check that P is the characteristic polynomial of the *companion* matrix (see §8.2.3):

$$A = \begin{pmatrix} 0 & 0 & 0 & \ldots & 0 & -a_0 \\ 1 & 0 & 0 & \ldots & 0 & -a_1 \\ 0 & 1 & 0 & \ldots & 0 & -a_2 \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & 0 & 0 & \ldots & 1 & -a_{d-1} \end{pmatrix}.$$

Therefore the roots of the polynomial P are the eigenvalues of the matrix A. We can therefore apply the methods of Chapter 13.

We have seen that the method `Polynomial.roots()` takes up to three parameters, all optional: `ring`, `multiplicities`, and `algorithm`. Assume that a Sage object of the `Polynomial` class has name `p` (so that `isinstance(p, 'Polynomial')` returns `True`). The algorithm used by the command `p.roots()` then depends on the parameters `ring` and `algorithm`, as well as on the coefficient ring of the polynomial, that is `p.base_ring()`.

The algorithm checks whether arithmetic operations in `ring` and `p.base_ring()` are exact. If this is not the case, the approximations to the roots are computed via the library NumPy if `p.base_ring()` is `RDF` or `CDF`, or via the library PARI otherwise (the parameter `algorithm` allows the user to override this and specify which library to use). By looking at the source code of NumPy we see that the root approximation method used by this library computes the eigenvalues of the companion matrix.

The following command identifies which objects have exact arithmetic operations (the method `Ring.is_exact()` returning `True` in this case).

```
sage: for ring in [ZZ, QQ, QQbar, RDF, RIF, RR, AA, CDF, CIF, CC]:
....:     print("{0:50} {1}".format(ring, ring.is_exact()))
Integer Ring                                       True
Rational Field                                     True
Algebraic Field                                    True
Real Double Field                                  False
Real Interval Field with 53 bits of precision      False
Real Field with 53 bits of precision               False
Algebraic Real Field                               True
Complex Double Field                               False
Complex Interval Field with 53 bits of precision   False
Complex Field with 53 bits of precision            False
```

When the parameter `ring` is `AA` or `RIF`, while `p.base_ring()` is `ZZ`, `QQ` or `AA`, the algorithms calls the function `real_roots()` of the module `sage.rings.polynomial.real_roots`. This function converts the polynomial into the Bernstein basis then uses Casteljau's algorithm (for evaluating polynomials in Berstein basis) and Descartes' rule of signs (see §sec:regle-de-descartes) to isolate the roots.

When the parameter `ring` is `QQbar` or `CIF` and `p.base_ring()` is `ZZ`, `QQ`, `AA` or the Gaussian numbers $\mathbb{Q}[\sqrt{-1}]$, the algorithm passes the computation to NumPy and PARI, whose results are then converted into the appropriate rings.

For a comprehensive view of all the situations covered by the method `Polynomial.ro` we refer the reader to the method's documentation.

Rate of convergence Consider a convergent sequence of numbers u and let ℓ be its limit. We say that the rate of convergence of the sequence u is *linear* if there exists $K \in (0, 1)$ such that

$$\lim_{n\to\infty} \frac{|u_{n+1} - \ell|}{|u_n - \ell|} = K.$$

The rate of convergence of the sequence u is said to be *quadratic* if there exists $K > 0$ such that

$$\lim_{n\to\infty} \frac{|u_{n+1} - \ell|}{|u_n - \ell|^2} = K.$$

Recall that Newton's method constructs a recursive sequence u defined by $u_{n+1} = \varphi(u_n)$, where φ is the function $x \mapsto x - f(x)/f'(x)$. If f is twice differentiable, Taylor's formula for φ with x in a neighbourhood of the root α is

$$\varphi(x) = \varphi(\alpha) + (x-\alpha)\varphi'(\alpha) + \frac{(x-\alpha)^2}{2}\varphi''(\alpha) + O_\alpha((x-\alpha)^3).$$

But $\varphi(\alpha) = \alpha$, $\varphi'(\alpha) = 0$ and $\varphi''(\alpha) = f''(\alpha)/f'(\alpha)$. By substituting in the previous formula and using the definition of the sequence u, we get

$$u_{n+1} - \alpha = \frac{(u_n-\alpha)^2}{2}\frac{f''(\alpha)}{f'(\alpha)} + O_\infty((u_n-\alpha)^3).$$

Therefore, when Newton's method converges, the convergence rate of the sequence is quadratic.

Acceleration of convergence Given a convergent sequence whose rate of convergence is linear, it is possible to construct a sequence whose rate of convergence is quadratic. The same technique, applied to Newton's method, is known as Steffensen's method.

```
sage: def steffensen(sequence):
....:     assert isinstance(sequence, GeneratorType)
....:     values = deque(maxlen=3)
....:     for i in range(3):
....:         values.append(sequence.next())
....:         yield values[i]
....:     while True:
....:         values.append(sequence.next())
....:         u, v, w = values
....:         yield u - (v - u)^2 / (w - 2 * v + u)
```

```
sage: g(x) = sin(x^2 - 2) * (x^2 - 2)
sage: sequence = newtongen(g, RR(0.7))
sage: accelseq = steffensen(newtongen(g, RR(0.7)))
sage: iterate(sequence, check=checkconv)
After 17 iterations: 1.41422192763287
sage: iterate(accelseq, check=checkconv)
After 10 iterations: 1.41421041980166
```

Note that the rate of convergence is an asymptotic notion: it says nothing about the error $|u_n - \ell|$ for a given n.

```
sage: sequence = newtongen(f, RR(a))
sage: accelseq = steffensen(newtongen(f, RR(a)))
```

Solution of non-linear equations	
Approximate roots of a polynomial	`Polynomial.roots()`
Exact roots (not guaranteed to give all of them)	`Expression.roots()`
Approximate real roots	`real_roots()`
Approximate roots via Brent's method	`Expression.find_root()`

TABLE 12.1 – Summary of commands described in this chapter.

```
sage: iterate(sequence, check=checkconv)
After 6 iterations: 2.15846852566756
sage: iterate(accelseq, check=checkconv)
After 7 iterations: 2.15846852554764
```

The method `Expression.find_root()`. We now consider the most general situation: the computation of an approximate solution of the equation $f(x) = 0$. In Sage, this is done using the method `Expression.find_root()`.

The parameters of the method `Expression.find_root()` allow us to specify an interval where the root should be found, the precision of the computation or the number of iterations. The parameter `full_output` gives access to additional information about the computation, in particular the number of iterations and the number of evaluations of the function.

```
sage: result = (f == 0).find_root(a, b, full_output=True)
sage: result[0], result[1].iterations
(2.1584685255476415, 9)
```

In fact, the method `Expression.find_root()` does not implement an algorithm for finding the solutions of equations: the computation is delegated to the module SciPy. The SciPy functionality used by Sage for solving equations implements Brent's method, which combines three of the methods we discussed above: the bisection method, the secant method, and quadratic interpolation. The two first approximate values are the endpoints of the interval where we are looking for a solution of the equation. The next approximation is obtained by linear interpolation, as in the secant method. In the following iterations, the function is approximated by quadratic interpolation; the x-coordinate of the intersection point of the interpolating curve and the x-axis is the new approximation, unless this x-coordinate is sandwiched between the last two computed approximations, in which case we continue with the bisection method.

The SciPy library does not offer arbitrary precision computations (unless we are satisfied to compute with integers); in fact, the source code of the method `Expression.find_root()` starts by converting the bounds into double precision numbers. On the other hand, all the illustrations of solution methods we have given in this chapter work in arbitrary precision, and even symbolically.

```
sage: a, b = pi/2, pi
sage: generator = newtongen(f, a)
```

```
sage: generator.next(); generator.next()
1/2*pi
1/2*pi - (e^(1/2*pi) - 10)*e^(-1/2*pi)
```

Exercise 45. Write a generator for Brent's method that works with arbitrary precision.