20 Lecture #14: Wednesday, April 1st, 2026

20.1 The secant method

The previous examples show that, although Newton's method is highly effective under favorable conditions, it also has some important limitations. On the one hand, it requires the computation of the derivative f', which in some applications may be difficult, expensive, or even impossible to obtain explicitly. On the other hand, methods that avoid derivative information, such as the bisection method, are usually more robust, but they converge only linearly and may therefore be too slow when high accuracy is needed. This naturally leads to the following question: can we construct a method that does not require the explicit computation of derivatives and yet converges faster than a linear method?

In Newton's formula (23), we have

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

for every $k \in \mathbb{N}$. As we saw in Section 8.1, the derivative can be approximated by a finite difference using the previous iterate x_{k-1}:

$$f'(x_k) \approx \frac{f(x_k) - f(x_{k-1})}{x_k - x_{k-1}}.$$

Substituting this approximation into Newton's formula, we obtain

$$\begin{aligned} x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} &\approx x_k - \frac{f(x_k)(x_k - x_{k-1})}{f(x_k) - f(x_{k-1})} \\ &= \frac{x_k\Big(f(x_k) - f(x_{k-1})\Big) - f(x_k)(x_k - x_{k-1})}{f(x_k) - f(x_{k-1})} \\ &= \frac{x_k f(x_k) - x_k f(x_{k-1}) - x_k f(x_k) + x_{k-1} f(x_k)}{f(x_k) - f(x_{k-1})} \\ &= \frac{f(x_k)x_{k-1} - f(x_{k-1})x_k}{f(x_k) - f(x_{k-1})}. \end{aligned}$$

This gives rise to the **secant method**, whose iteration is therefore defined by

$$x_{k+1} = \frac{f(x_k)x_{k-1} - f(x_{k-1})x_k}{f(x_k) - f(x_{k-1})}. \tag{24}$$

Formula (24) also has a clear geometric interpretation. Instead of taking the tangent line to the graph of f at the point x_k, as in Newton's method, we consider the secant line passing through the two points $(x_{k-1}, f(x_{k-1}))$ and $(x_k, f(x_k))$ (see Figure 95). The next iterate x_{k+1} is then defined as the intersection of this secant line with the x-axis (see Figure 96). In this sense, the secant method replaces the derivative by the slope of the secant line, and therefore can be viewed as a derivative-free approximation of Newton's method.

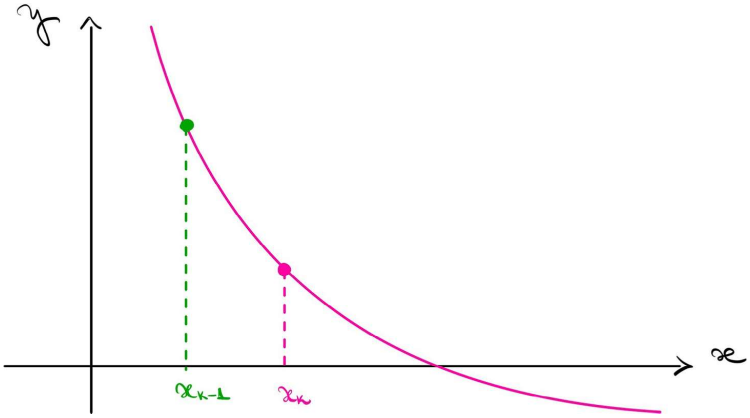


Figure 95: Pick an iteration x_k and the previous one.

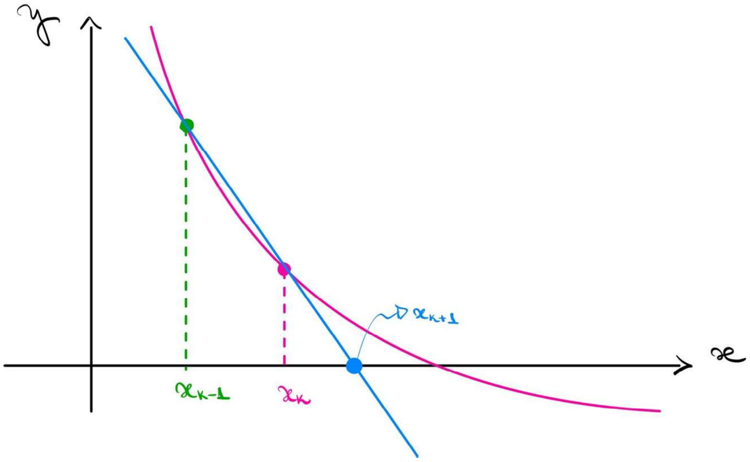


Figure 96: Connect the secant line between them and x_{k+1} is the new approximation.

Algorithm 20.1 (secant method for finding root of a function f)**.** Given a continuous function f and a tolerance ε we follow the steps.

0. Choose x_0, x_1.

1. Let

$$x_{k+1} = \frac{x_{k-1} f(x_k) - x_k f(x_{k-1})}{f(x_k) - f(x_{k-1})}.$$

If $|x_{k+1} - x_k| < \varepsilon$ or $|f(x_{k+1})| < \varepsilon$, then the algorithm stops and the output is x_{k+1}. Otherwise, increase k by one and go back to step 1.

Let us now collect some relevant observations about the secant method. The following items also provide a brief analysis of the main ideas that have appeared so far in our study of the bisection and Newton methods. We choose to present these remarks in itemized form, since this makes the comparison clearer and allows us to see more easily what is going on.

- The secant method **does not provide direct control of the error**. As in Newton's method, the iterates are intended to approach the root, but there is no explicit bound that tells us how far x_k

is from the true solution. This is in clear contrast with the bisection method, where the enclosing interval immediately gives an error estimate.

- The secant method is **not a bracketing method**. Just as in Newton's method, the new approximation is obtained from a local linear construction, and there is no guarantee that the root remains enclosed in an interval where the function changes sign. Hence, both Newton's method and the secant method may leave the region where the root was initially expected, while the bisection method always keeps the root bracketed.
- Because of this lack of direct error control, one must impose once again stopping conditions. These are essentially the same stopping criteria that are used in Newton's method. In both methods, the algorithm is stopped when successive iterates become sufficiently close or when the function value is small enough, since neither method provides a guaranteed error bound of the type available in bracketing methods.
- The secant method is also **unreliable** in much the same way as Newton's method. Its performance depends strongly on the initial approximations, and poor starting values may lead to slow convergence, instability, or divergence (see Figure 97).

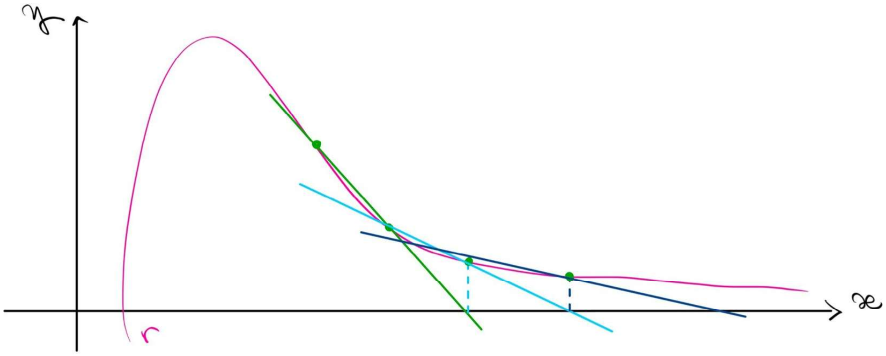


Figure 97: The approximations are going farther and farther away.

- The following table was obtained from experiments carried out with Maple. It shows that the secant method requires 20 iterations to reach the tolerance 10^{-4} and only 21 iterations to reach the much smaller tolerance 10^{-7}. Thus, even though the requested accuracy is significantly increased, the number of iterations grows only by one.

tolerance ε	10^{-4}	10^{-7}
secant	20 iterations	21 iterations

  This suggests that the rate of improvement of the secant method is quite fast. Indeed, reducing the tolerance from 10^{-4} to 10^{-7} means asking for a substantially more accurate approximation, yet the computational effort increases only very slightly.

The behavior of the last example above is consistent with the fact that the secant method converges faster than a linear method, and therefore provides a substantial gain in efficiency. Indeed, we have the following result.

Theorem 20.2. The secant method is of order $\alpha = \frac{1+\sqrt{5}}{2} \approx 1.6$ for twice continuously differentiable functions. For roots of higher multiplicity it is of order 1.

20.2 A summary of all three methods and comparison between them

This table summarizes the main qualitative differences between the three methods we have studied. The bisection method is the most reliable one, because it is a bracketing method: the root always remains inside an interval where the function changes sign. For this reason, it also provides direct control over the error, since the size of the interval gives explicit information about the accuracy of the approximation. The price to pay for this robustness is that the method is slow, because its convergence is only linear.

Bisection	**Newton**	**Secant**
reliable	unreliable	unreliable
control over error	no error control	no error control
slow	fast	fast

By contrast, Newton's method and the secant method are both faster, but less reliable. Their iterates are not obtained from a bracketing procedure, so they do not guarantee that the root remains enclosed at every step. As a consequence, they do not provide direct control over the error in the same way as bisection does, and one must rely instead on practical stopping criteria. Their advantage, however, is speed: Newton's method is typically very fast near a simple root, and the secant method, although slightly slower than Newton's method, is still much faster than a linear method.

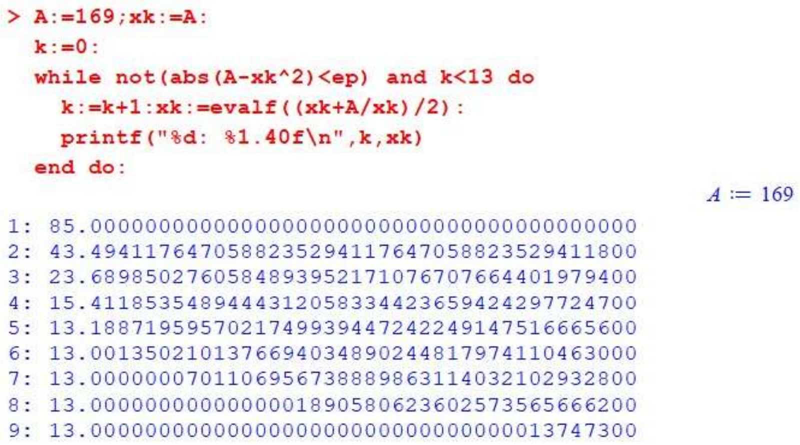
20.3 The Babylonian method

The problem is to approximate $\sqrt{A}$ for a given positive real number A. One classical way to do this is the Babylonian method, which can be viewed as a special case of Newton's method applied to the equation $x^2 - A = 0$. Starting from an initial positive guess, the method produces a sequence of better and better approximations to $\sqrt{A}$. This procedure is very old: methods of this kind were already known in ancient Mesopotamia, which is why it is called the *Babylonian* method. The name reflects the historical fact that Babylonian mathematicians used remarkably effective numerical algorithms for computing square roots long before the modern formulation of Newton's method.

Let us describe the method with more details. Let A be a positive real number. As we want to find an approximation for $\sqrt{A}$, we set $\sqrt{A} = x$, which is the same as $x^2 = A$ for $x > 0$. As we said in the previous paragraph, we then are looking for a root of the the continuous function $f(x) = x^2 - A$. For this, we will apply the Newton Method as follows. By using (23), we have that for $k \in \mathbb{N}$

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} = x_k - \frac{x_k^2 - A}{2x_k} = \frac{x_k}{2} + \frac{A}{2x_k} = \frac{1}{2}\left(x_k + \frac{A}{x_k}\right).$$

In the Maple experiment in Figure 98, one can observe that, after a few iterations, the number of trustworthy digits in the approximation increases very rapidly. Once the iterates are sufficiently close to $\sqrt{169} = 13$, the displayed values show that the correct decimal digits are essentially doubled from one step to the next. This behavior is exactly what one should expect from Newton's method: its error satisfies, locally, a quadratic convergence law, meaning that if the error at step k is small, then the error at step $k+1$ is proportional to the square of the previous one. As a consequence, the approximation improves dramatically at each iteration, which explains the fast stabilization of the digits in the Maple output.



```
> A:=169;xk:=A:
  k:=0:
  while not(abs(A-xk^2)<ep) and k<13 do
    k:=k+1:xk:=evalf((xk+A/xk)/2):
    printf("%d: %1.40f\n",k,xk)
  end do:
                                                            A := 169
1: 85.0000000000000000000000000000000000000000
2: 43.4941176470588235294117647058823529411800
3: 23.6898502760584893952171076707664401979400
4: 15.4118535489444312058334423659424297724700
5: 13.18871959570217499394472422249147516665600
6: 13.0013502101376694034890244817974110463000
7: 13.000000701106956738889863114032102932800
8: 13.0000000000000001890580623602573565666200
9: 13.0000000000000000000000000000000013747300
```

Figure 98: The Babylonian method for calculating an approximation for $\sqrt{13}$.

The stopping condition used in the Maple code is based on the *residual* rather than on the true error (see item (3) in Definition 19.9). In principle, the most natural quantity to monitor would be the error

$$E_k = \sqrt{A} - x_k$$

as this measures directly how far the current iterate x_k is from the exact value $\sqrt{A}$. However, this is not practical here, because the exact root $\sqrt{A}$ is precisely the unknown quantity we are trying to approximate, so we cannot evaluate E_k during the computation. For this reason, we use instead the residual

$$r_k = x_k^2 - A$$

which measures how well the current approximation satisfies the equation $x^2 - A = 0$. In words, the residual tells us how far x_k^2 is from the target value A: if x_k^2 is very close to A, then x_k is a good approximation of $\sqrt{A}$. This is exactly what appears in the Maple stopping test, namely `abs(A-xk^2)<ep`, which requires the absolute value of the residual to be smaller than the prescribed tolerance. Besides this, one should also notice another important feature of the code: the loop is executed only while `k<13`. Therefore, the algorithm stops not only when the residual is sufficiently small, but also when the number of iterations reaches this upper bound.

20.4 The Newton-Raphson method

Suppose that n and d are two real numbers with $d \neq 0$, and we want to compute an approximation of the quotient n/d. Even when n and d are integers, the number n/d need not have a finite decimal expansion. Indeed, it may be irrational. For this reason, division can also be viewed as a problem of numerical approximation.

Notice first that since we can write

$$\frac{n}{d} = n \cdot \frac{1}{d},$$

it is enough to approximate the reciprocal $1/d$. If we write $x = \frac{1}{d}$, then this is equivalent to the equation $xd = 1$. Thus, instead of performing the division directly, we look for the root of an equation whose solution is $x = 1/d$. A convenient choice is $f(x) = \frac{1}{x} - d$. We do not choose instead $f(x) = x - \frac{1}{d}$, because the quantity $1/d$ is precisely the unknown number we are trying to compute, so such a formulation would already assume that the answer is known. Applying Newton's method to $f(x) = \frac{1}{x} - d$, we have $f'(x) = -\frac{1}{x^2}$ and therefore

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} = x_k - \frac{\frac{1}{x_k} - d}{-\frac{1}{x_k^2}} = x_k + x_k^2 \left(\frac{1}{x_k} - d \right) = x_k + x_k(1 - dx_k).$$

This is called the **Newton–Raphson method** for approximating $1/d$ is and it is given by

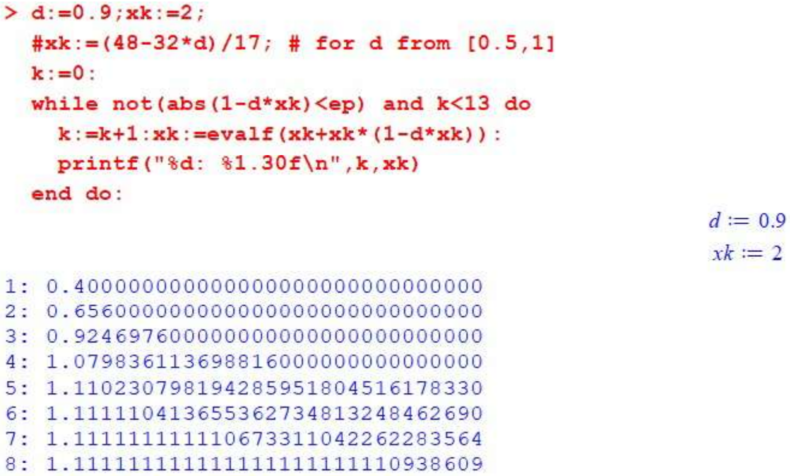
$$x_{k+1} = x_k + x_k(1 - dx_k). \tag{25}$$

Once an approximation of $1/d$ has been obtained, an approximation of n/d is obtained simply by multiplying by n. Let us use the iterations in (25) on Maple.

Remark 20.3. In this procedure, it is enough to consider the case in which $d \in [0.5, 1]$. Indeed, for a general positive number d, we can always rescale it by a suitable power of 10 and reduce the computation of $1/d$ to the reciprocal of a number in that interval. For example,

$$\frac{1}{13} = \frac{1}{10} \cdot \frac{1}{2} \cdot \frac{1}{0.65}.$$

Now the number 0.65 belongs to $[0.5, 1]$, so it is enough to approximate $1/0.65$ and then multiply the result by $1/20 = 0.05$.



```
> d:=0.9;xk:=2;
  #xk:=(48-32*d)/17; # for d from [0.5,1]
  k:=0:
  while not(abs(1-d*xk)<ep) and k<13 do
    k:=k+1:xk:=evalf(xk+xk*(1-d*xk)):
    printf("%d: %1.30f\n",k,xk)
  end do:
```

```
                                                        d := 0.9
                                                        xk := 2
1: 0.400000000000000000000000000000
2: 0.656000000000000000000000000000
3: 0.924697600000000000000000000000
4: 1.079836113698816000000000000000
5: 1.110230798194285951804516178330
6: 1.111104136553627348132484626690
7: 1.111111111110673311042262283564
8: 1.111111111111111111111110938609
```

Figure 99: The Newton-Raphson division to find an approximation for 1/0.9.

This Maple code in Figure 99 implements the Newton-Raphson iteration for approximating the reciprocal 1/0.9. The variable `xk` stores the current iterate, that is, the current approximation of $1/d$, and the code starts with the initial guess `xk:=2`. After initializing the iteration counter `k:=0`, the loop applies Newton-Raphson method with formula (25). At each step, the new value of `xk` is computed and printed, so that one can observe the convergence of the iterates toward

$$\frac{1}{0.9} = 1.111....$$

As in the Babylonian method, the stopping condition is not written in terms of the true error, because the exact quantity $1/d$ is the unknown we are trying to compute. Instead, the code uses the residual. Since the defining equation is $dx = 1$, the residual is $r_k = 1 - dx_k$, which measures how well the current approximation satisfies the equation. In words, it tells us how far the product dx_k is from 1: if dx_k is very close to 1, then x_k is a good approximation of $1/d$. This is exactly what appears in the Maple stopping test

```
abs(1-d*xk)<ep
```

which requires the residual to be smaller than the prescribed tolerance. One should also notice, exactly as before, that the loop includes the additional condition `k<13`. Therefore, the algorithm stops either when the residual is sufficiently small or when the maximum allowed number of iterations is reached.