Comp 311
Functional Programming

Eric Allen, Two Sigma Investments
Robert “Corky” Cartwright, Rice University
Sagnak Tasirlar, Two Sigma Investments

Call-By-Value
and
Call-By-Name

Call-By-Value

- Thus far, the evaluation semantics we have studied (both with the substitution and environment models) is known as call-by-value:
  - To evaluate a function application, we first evaluate the arguments and then evaluate the function body

Call-By-Value

- We have seen several “special forms” where this evaluation semantics is not what we want:

`&&`   `||`   `if-else`

Call-By-Value

- We could delay evaluation in these cases by wrapping arguments in function literals that take no parameters

```
def myOr(left: Boolean, right: () => Boolean) =
  if (left) true
  else right()
```

Call-By-Value

- We could delay evaluation in these cases by wrapping arguments in function literals that take no parameters

```
myOr(true, () => 1/0 == 2) ↦ true
```

- Functions that take no arguments are referred to as *thunks*

Call-By-Name

- Scala provides a way that we can pass arguments as thunks without having to wrap them explicitly

```
def myOr(left: Boolean, right: => Boolean) =
  if (left) true
  else right()
```

We simply leave off the parentheses in the parameter's type

Call-By-Name

- Now we can call our function without wrapping the second argument in an explicit thunk:

```
myOr(true, 1/0 == 2) ↦ true
```

- The thunk is applied (to nothing) the first time that the argument is evaluated in a function

Call-By-Name

- We can use by-name parameters to define new *control abstractions:*

```
def myAssert(predicate: => Boolean) =
  if (assertionsEnabled && !predicate)
    throw new AssertionError
```

Syntactic Sugar: Braces for Passing Arguments

- Any function that takes a single argument can be applied by passing the argument enclosed in braces instead of parentheses

```
myAssert {
  2 + 2 == 4
}
```

Syntactic Sugar: Braces for Passing Arguments

- Any function that takes a single argument can be applied by passing the argument enclosed in braces instead of parentheses

```
myAssert {
  def double(n: Int) = 2 * n
  double(2) == 4
}
```

The Environment Model of Type Checking

The Environment Model of Type Checking

- We have used environments in type checking to hold the bounds on type parameters

- They can also be used to record the types of names and function parameters

- Rather than thinking of typing rules as substitutions, we can think of them directly as assertions on expressions that we can reason with according to a logic

The Environment Model of Type Checking

- As a convenient notation, we express subtyping rules in the context of an environment by placing an environment to the left of a “turnstile” and a typing judgement to the right

$$\frac{}{\{T <: \texttt{Any}\} \vdash T <: T}\ \texttt{[S-Refl1]}$$

The Environment Model of Type Checking

- As a convenient notation, we express subtyping rules in the context of an environment by placing an environment to the left of a “turnstile” and a typing judgement to the right

$$\frac{}{\{T <: N\} \vdash T <: T}\ \texttt{[S-Refl2]}$$

The Environment Model of Type Checking

- As a convenient notation, we express subtyping rules in the context of an environment by placing an environment to the left of a “turnstile” and a typing judgement to the right

$$\frac{}{\Delta \vdash T <: T}\ \texttt{[S-Refl]}$$

The Environment Model of Type Checking

- We express typing rules in the context of
  - a type parameter environment and
    - a type environment (mapping names to types)
- We place both environments to the left of the “turnstile” (separated by a semicolon) and a typing judgement to the right:

$$\frac{}{\Delta; \Gamma + \{\mathtt{x:T}\} \vdash \mathtt{x:T}}\ \mathtt{[T\text{-}Var]}$$

The Environment Model of Type Checking

- Some typing judgements require assumptions
- We place assumed judgements above a horizontal bar (above the resulting type judgement)

$$\frac{\Delta ; (\Gamma + \texttt{x:N}) \vdash \texttt{e:M}}{\Delta ; \Gamma \vdash \texttt{((x:N) => e):(N => M)}} \; \texttt{[T-Arrow]}$$

The Environment Model of Type Checking

- Function applications involve checking the function and the arguments:

$$\frac{\Delta ; \Gamma \vdash e_0 : \texttt{R => S};\ \ \Delta ; \Gamma \vdash e_1 : \texttt{T};\ \ \Delta \vdash \texttt{T <: R};}{\Delta ; \Gamma \vdash e_0\ e_1 : \texttt{S}}\ \texttt{[T-App]}$$

The Environment Model of Type Checking

- To type check an expression in a pair of environments:
  - Form a proof tree, where each node is the application of an inference rule
  - The root of the tree is the typing judgement we are trying to prove
  - Each premise in a given rule is the root of a subtree proving that premise

The Environment Model of Type Checking

- For each form of expression there is exactly one inference rule
- Therefore, proving a typing judgement is simply a recursive descent over the structure of an expression

Generative Recursion

Generative vs Structural Recursion

- The functions we have studied to this point have (mostly) followed a common pattern:
  - Break into cases
  - Decompose data into components
  - Process components (usually recursively)
- Functions that follow this pattern are referred to as *structurally recursive functions*

Generative vs Structural Recursion

- Some problems are not amenable to solution by recursive descent
  - Instead, a deeper insight or “eureka” is required
  - Often a result from mathematics or computer science must be applied to discover important structure
  - Consider Euclid’s Algorithm for GCD
- The discovery of these insights and construction of solutions using them is the study of *algorithms*

Generative vs Structural Recursion

- Typically the design of an algorithm distinguishes two kinds of problems:
  - Base cases (or trivially solvable cases)
  - Problems that can be reduced to other problems of the same form
- The design of algorithms using this approach is referred to as *generative recursion*

Square Roots

- We would like to define a function `sqrt` that takes a non-negative value of type `Double` and returns the square root of that value

$$x^2 = 2$$

- There is no obvious way to apply structural recursion to this problem

Square Roots

- We would like to define a function `sqrt` that takes a non-negative value of type `Double` and returns the square root of that value

$$x^2 - 2 = 0$$

- There is no obvious way to apply structural recursion to this problem

Newton’s Method

- We can use derivatives to find successively better approximations to the zeroes of a real-valued function:

$$f(x) = 0$$

Newton’s Method

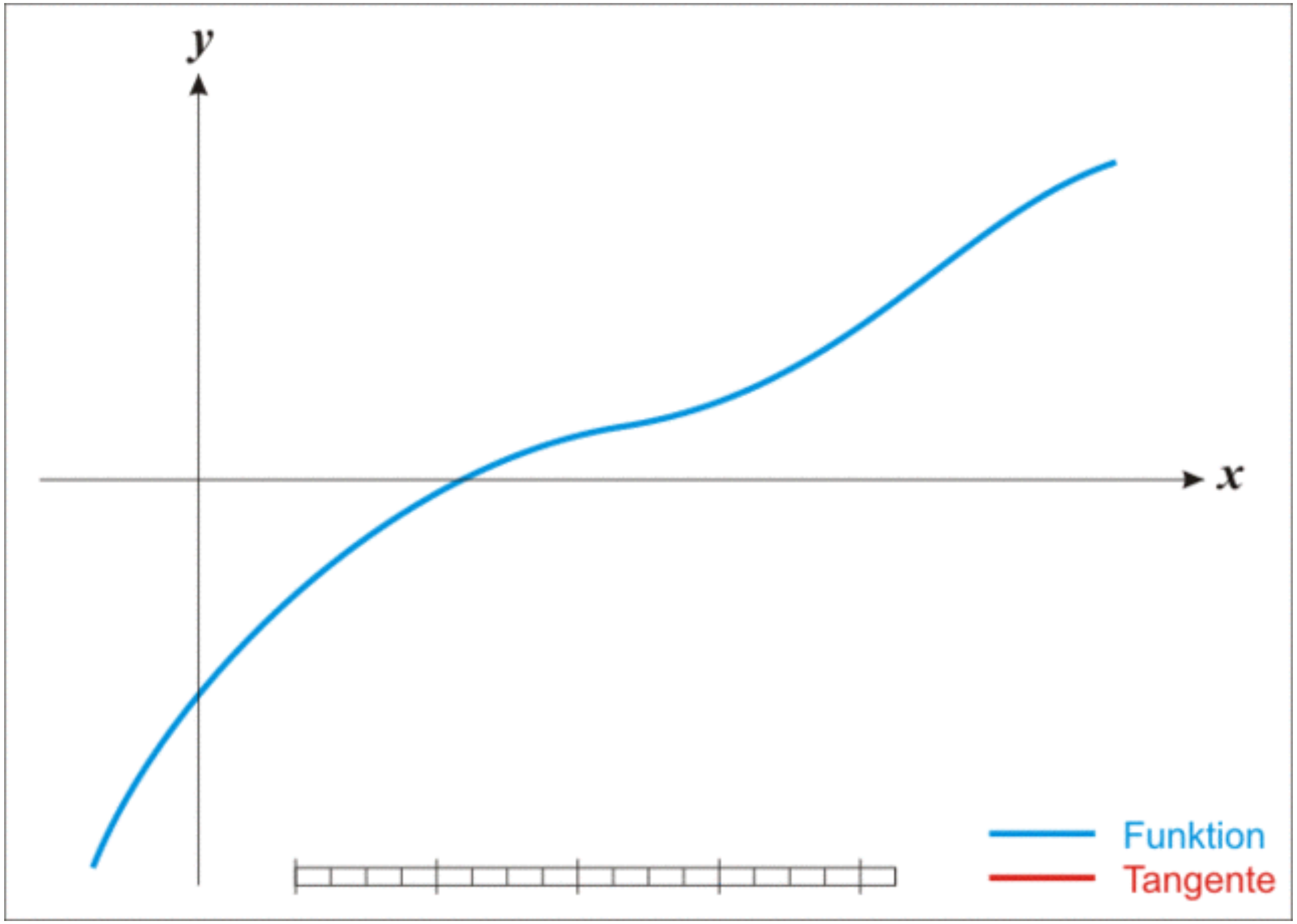
- We start with some guess for a value of `x`

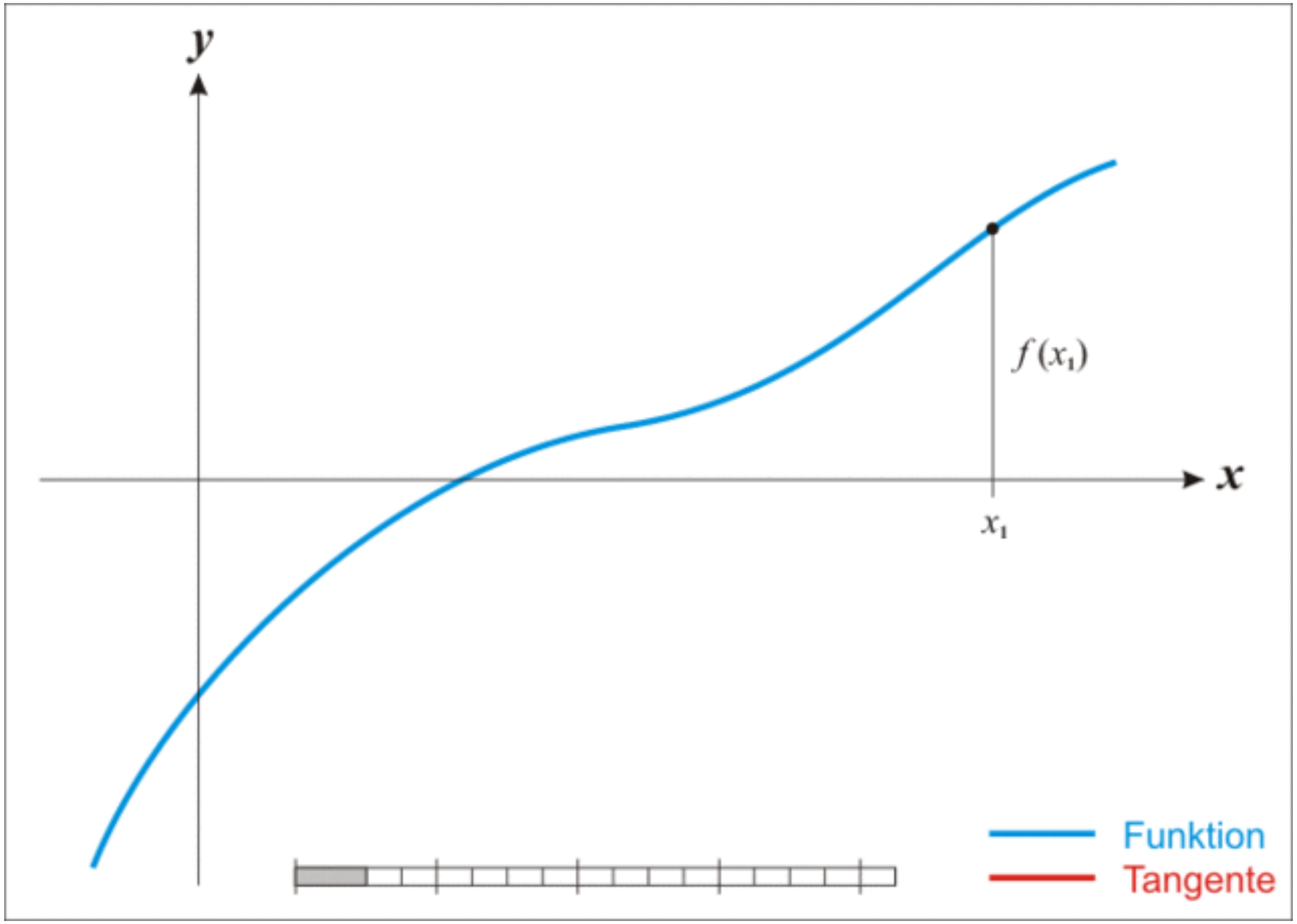
$$x_0 = \texttt{guess}$$

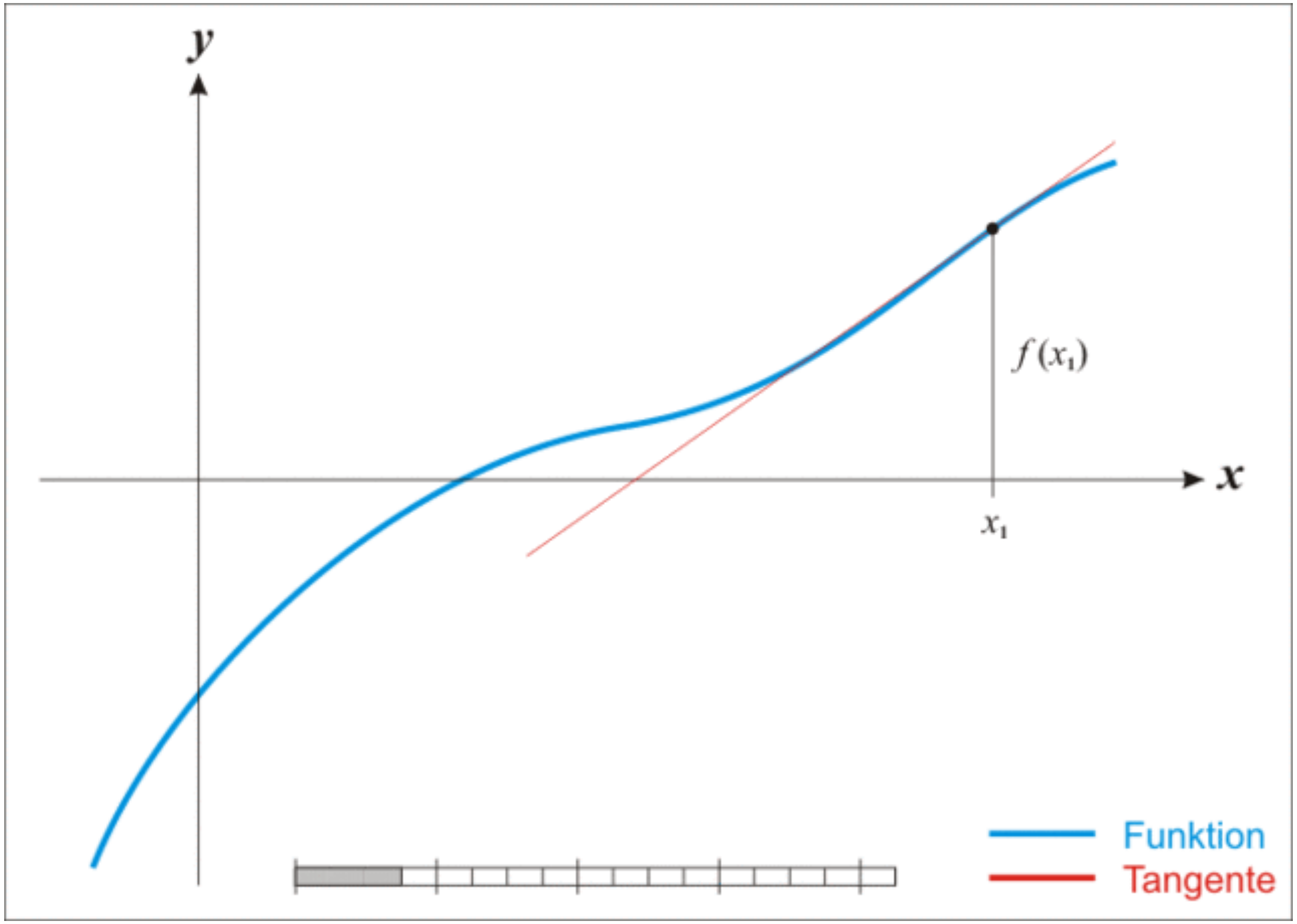
Newton’s Method

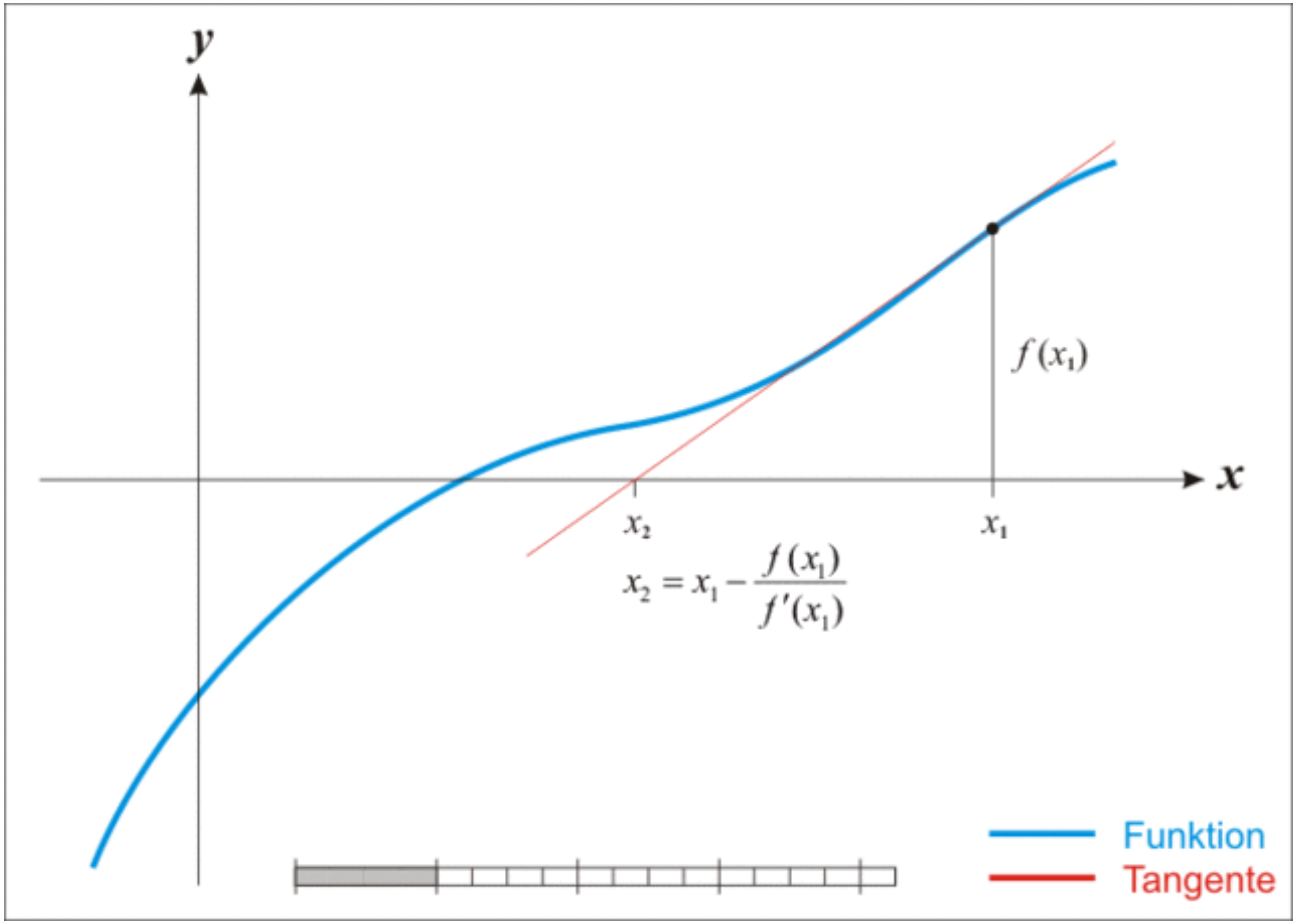
- Then we construct a better approximation with the following formula:

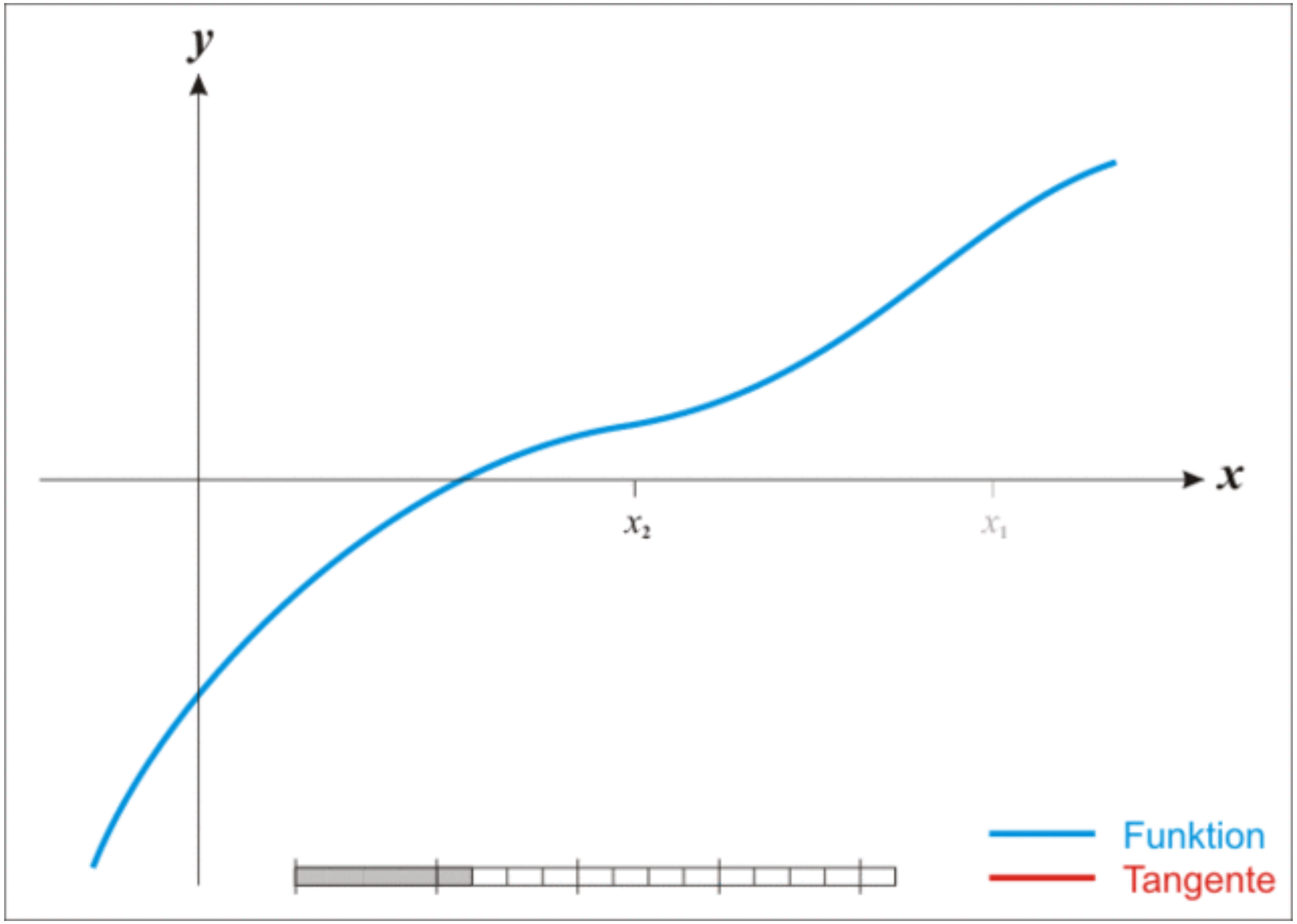
$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

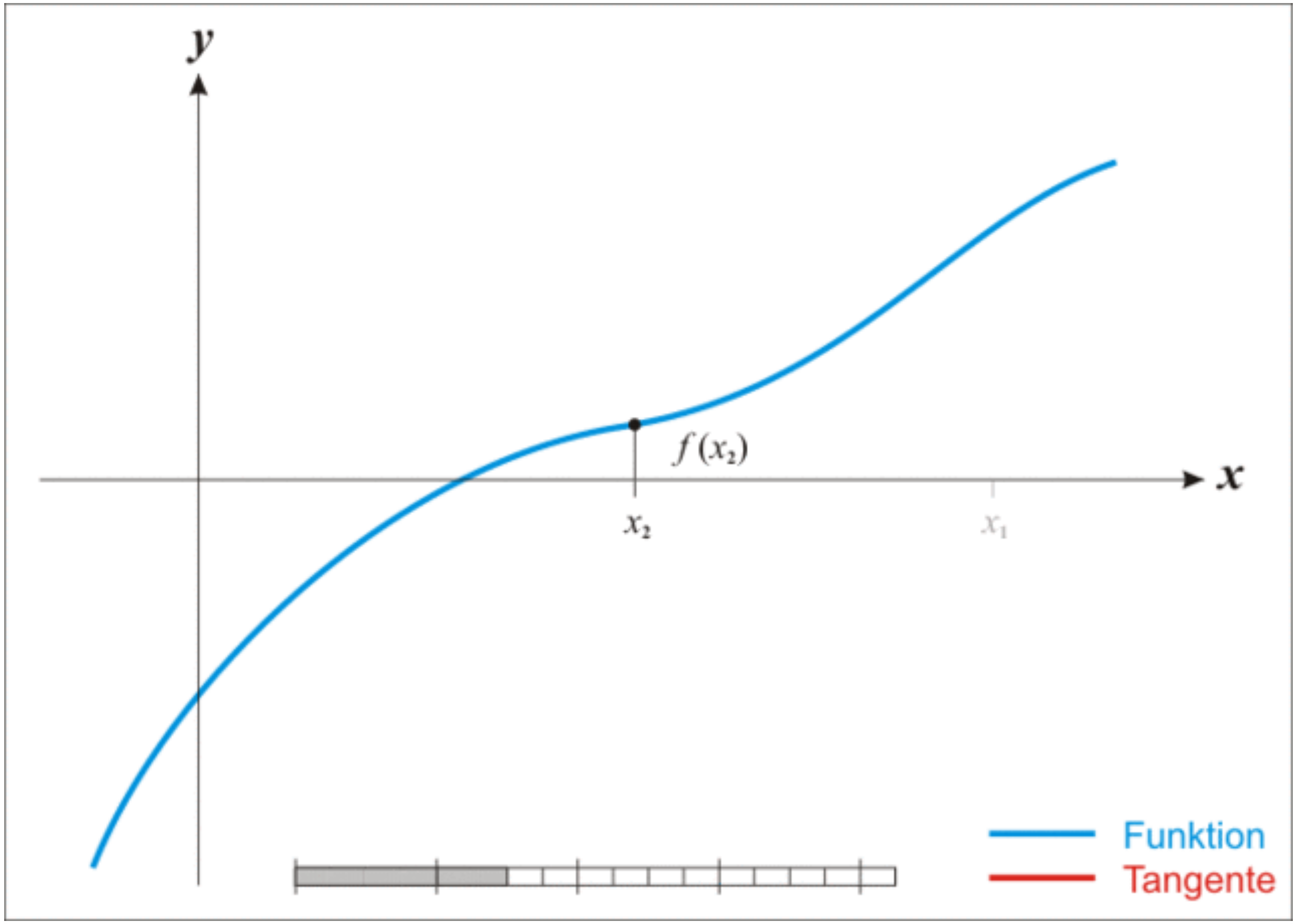

y
x
Funktion
Tangente

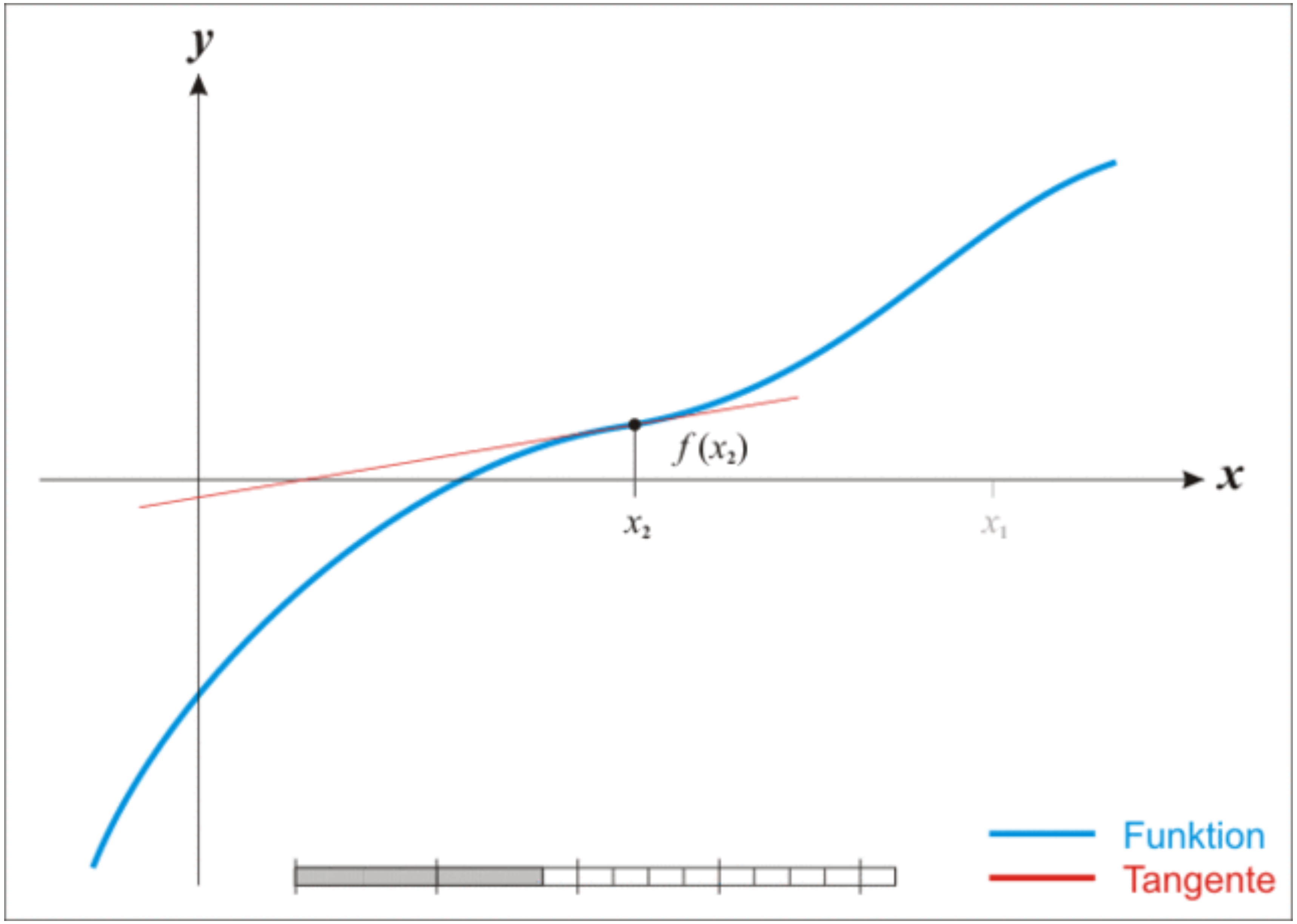

y
x
$f(x_1)$
x_1
Funktion
Tangente

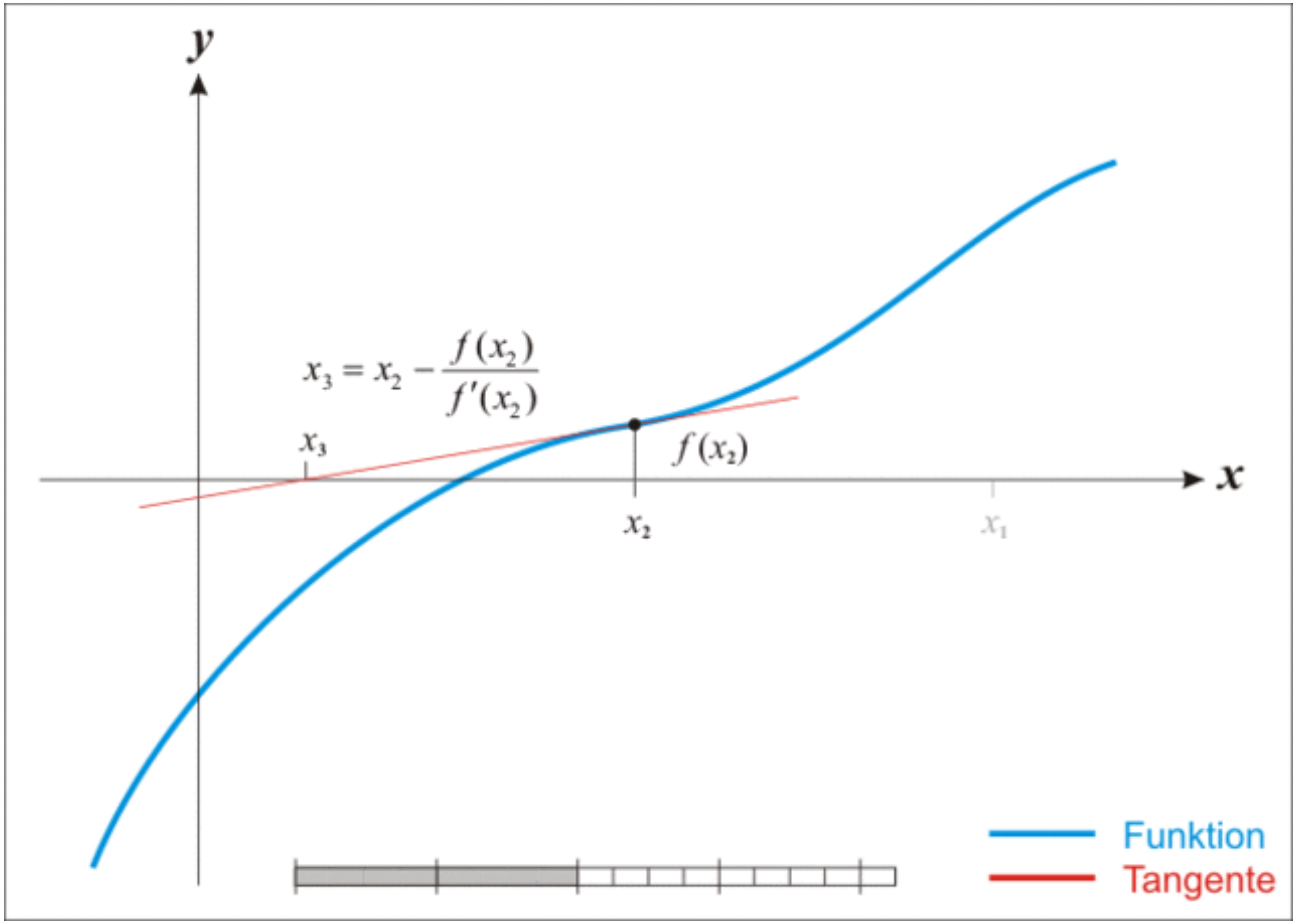

y
x
f(x₁)
x₁
Funktion
Tangente

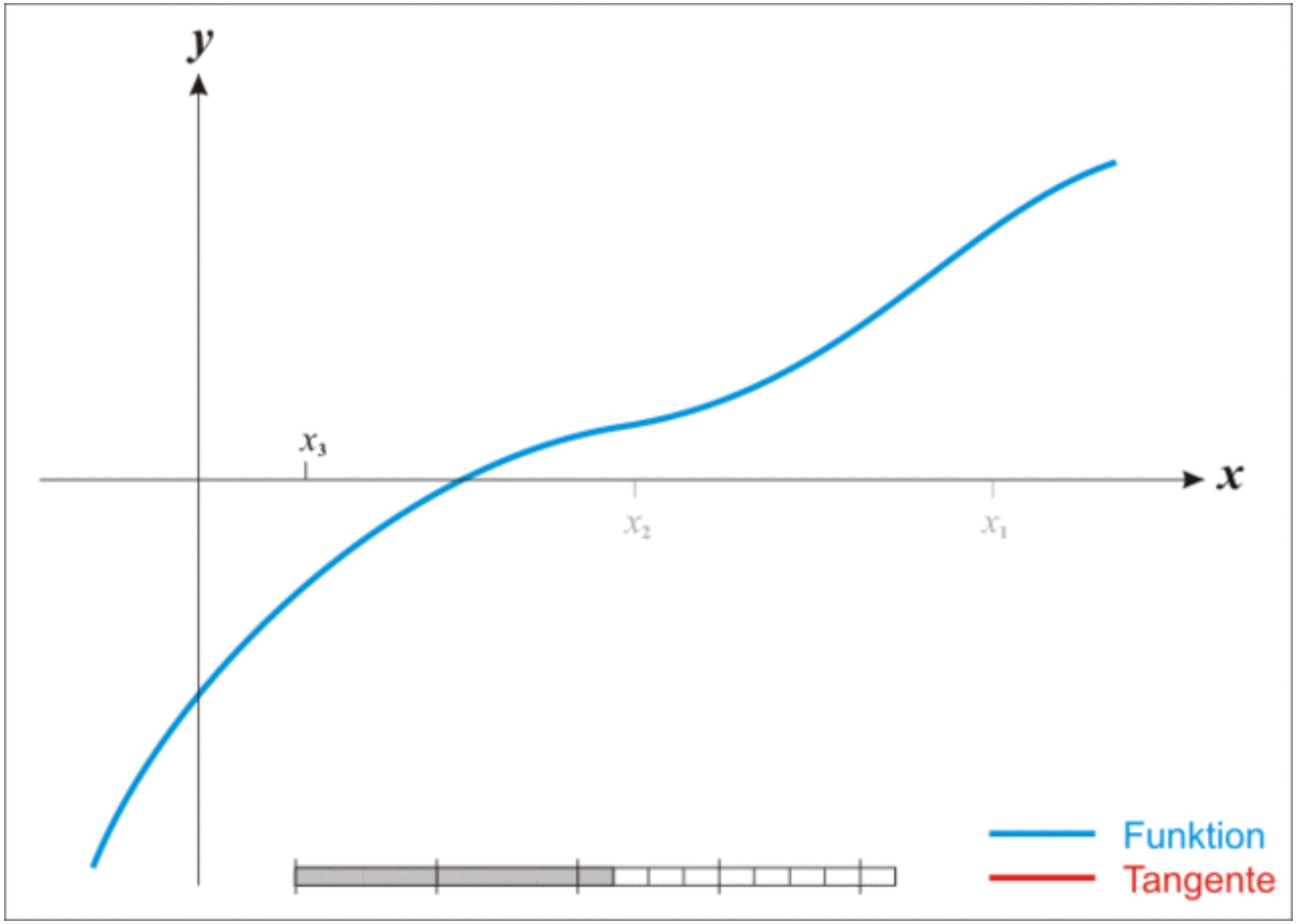

y
x
$f(x_1)$
x_2
x_1
$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)}$
Funktion
Tangente

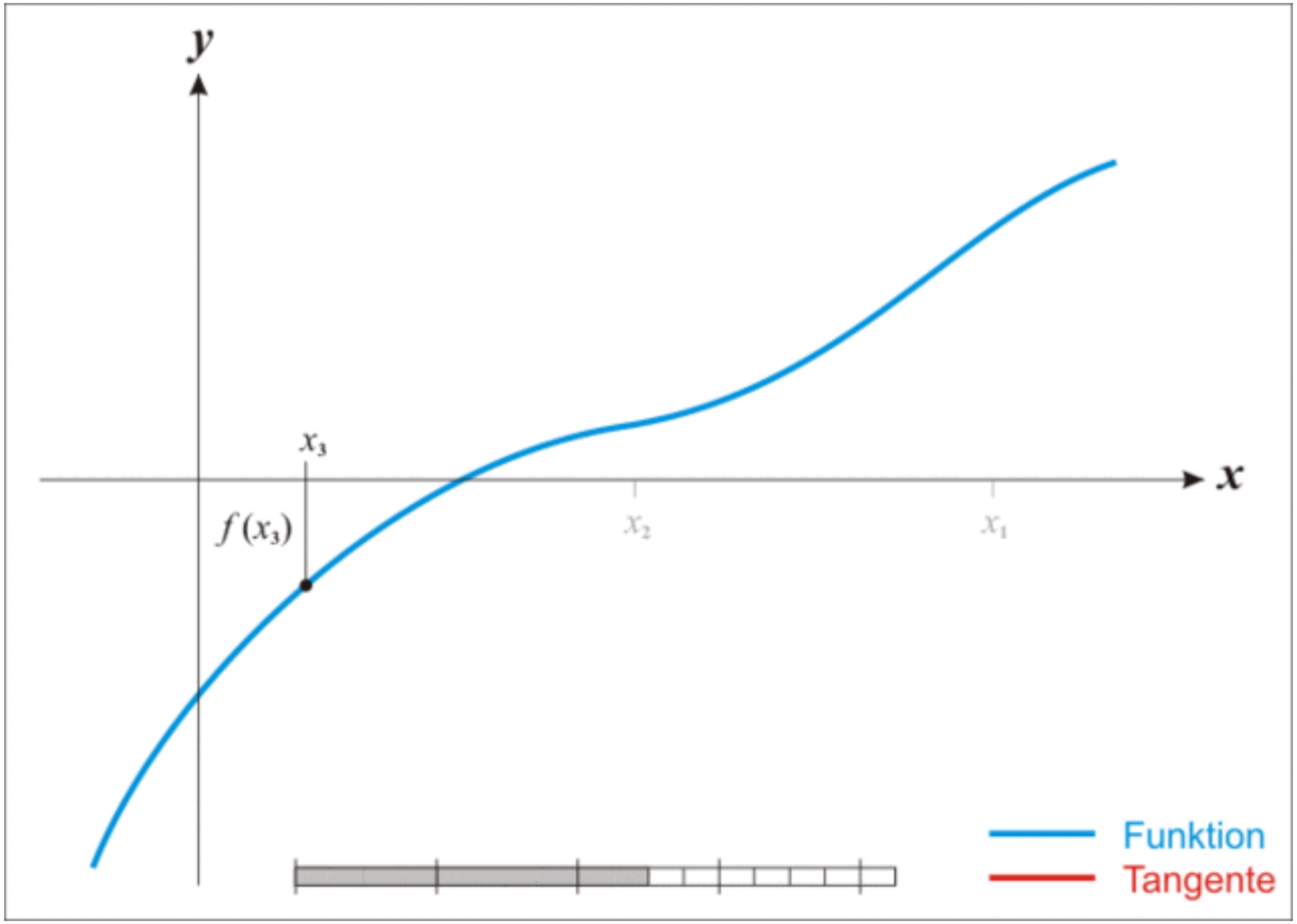

y
x
x_2
x_1
Funktion
Tangente

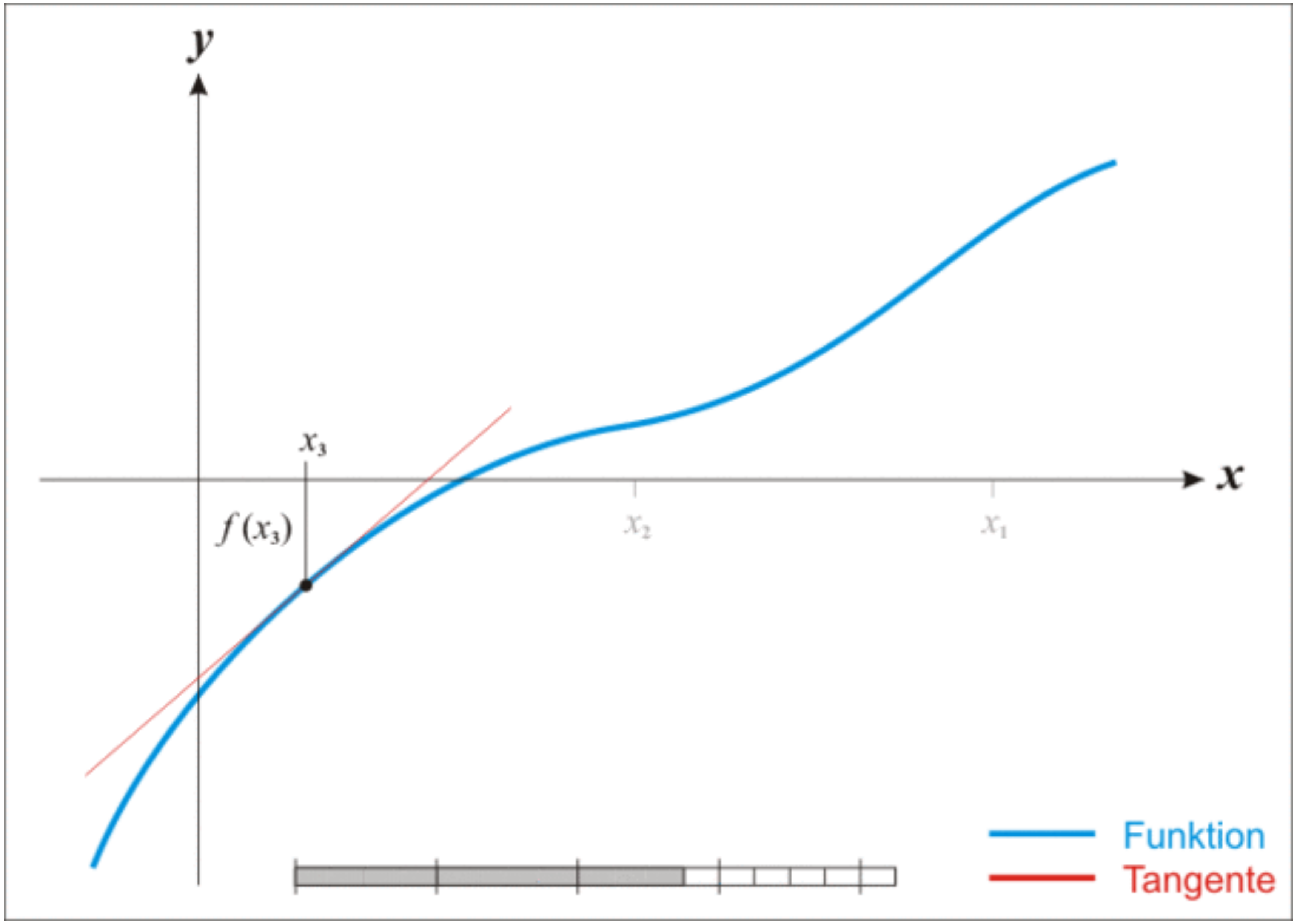

y
x
f(x₂)
x₂
x₁
Funktion
Tangente

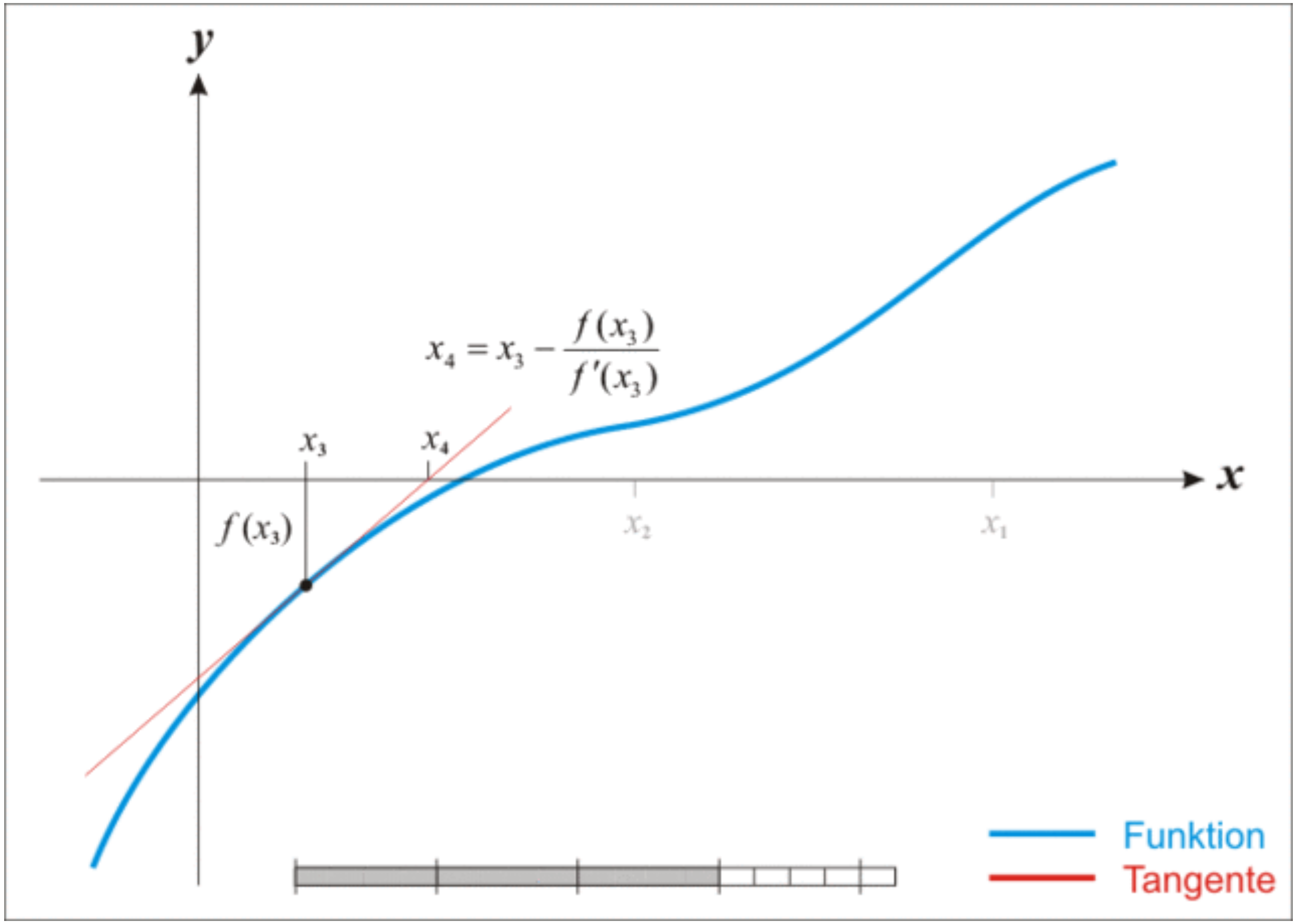

y
x
$f(x_2)$
x_2
x_1
Funktion
Tangente

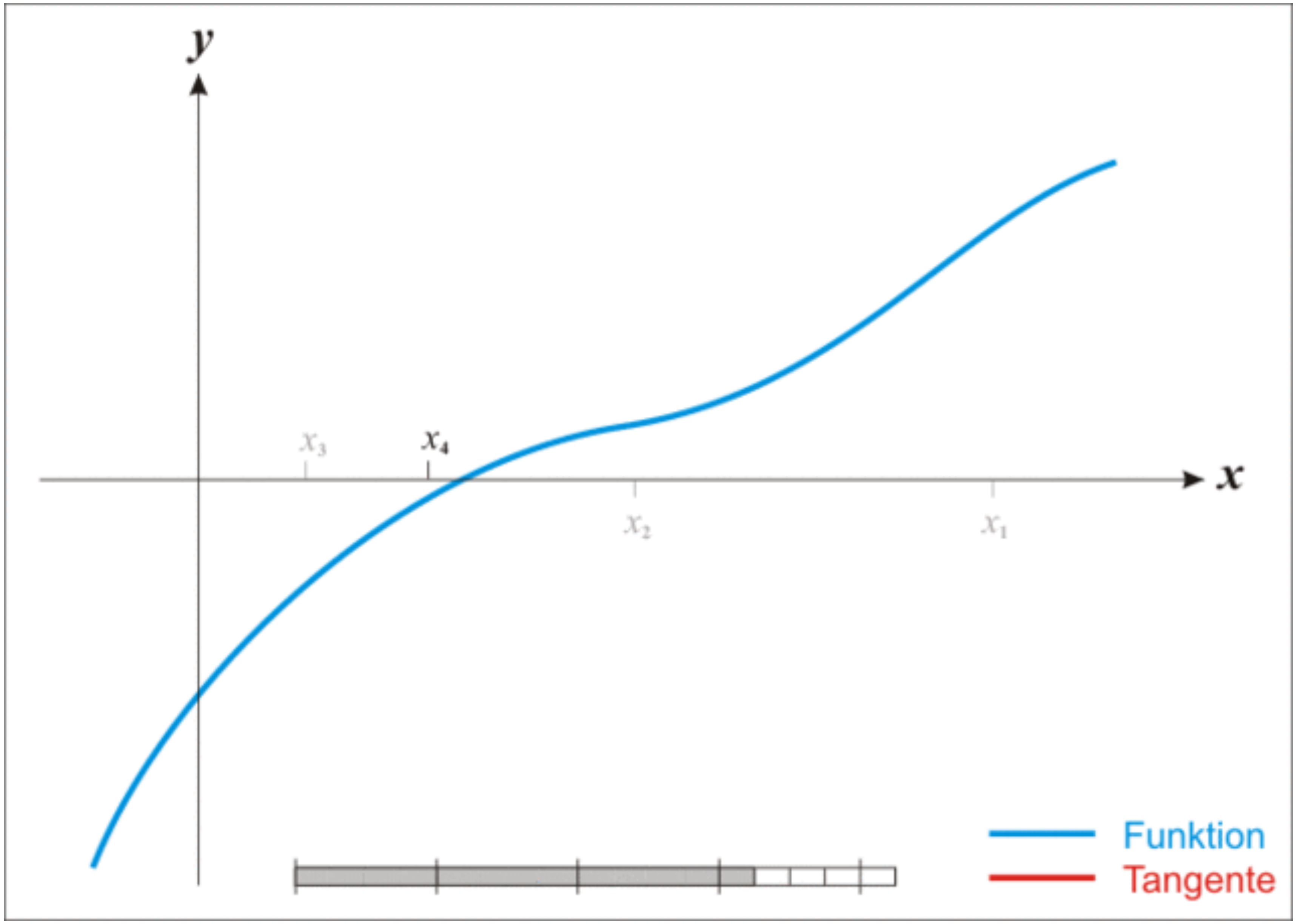

y
$x_3 = x_2 - \frac{f(x_2)}{f'(x_2)}$
x_3
$f(x_2)$
x
x_2
x_1
Funktion
Tangente

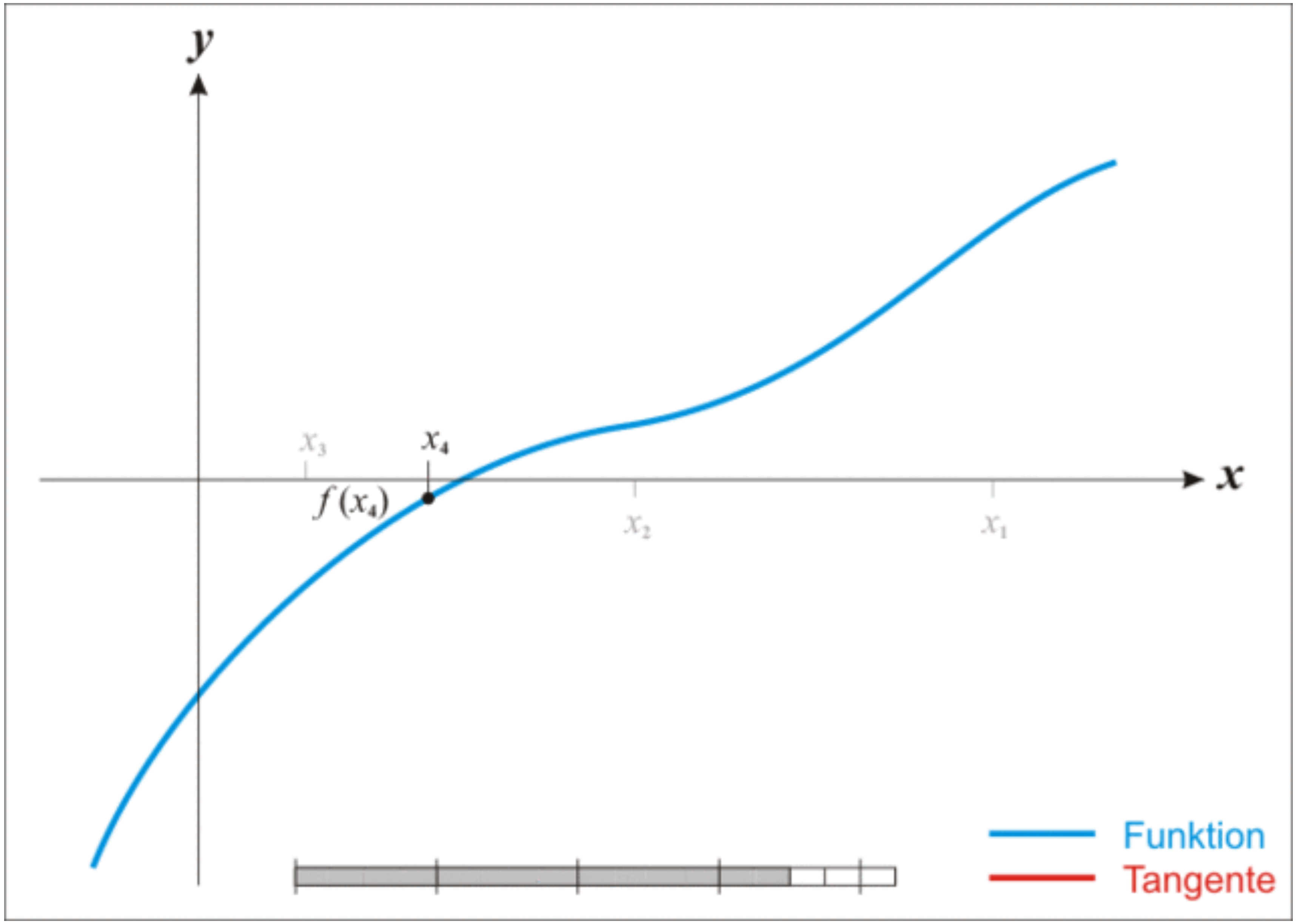

y
x
x3
x2
x1
Funktion
Tangente

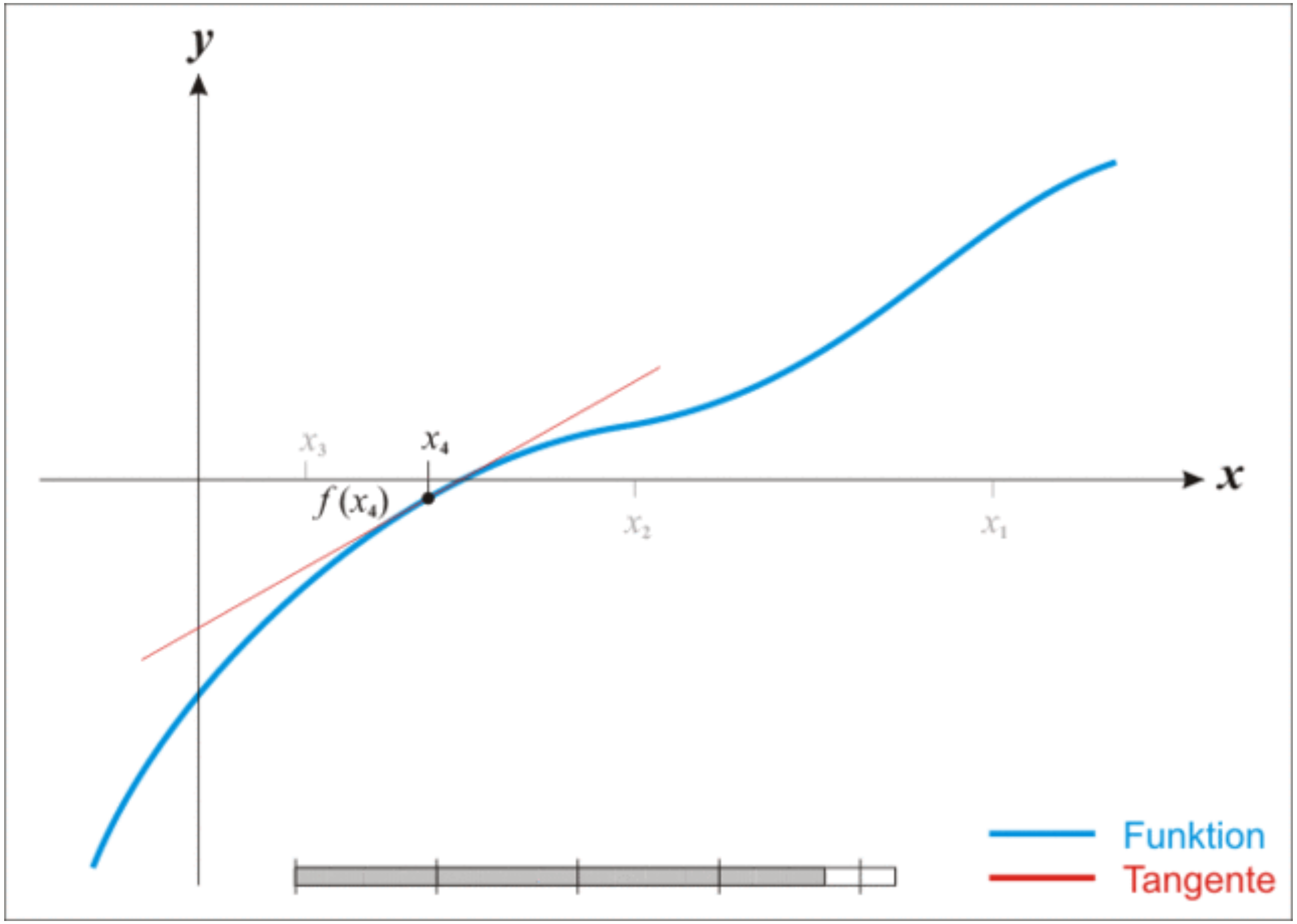

y
x
x_3
$f(x_3)$
x_2
x_1
Funktion
Tangente

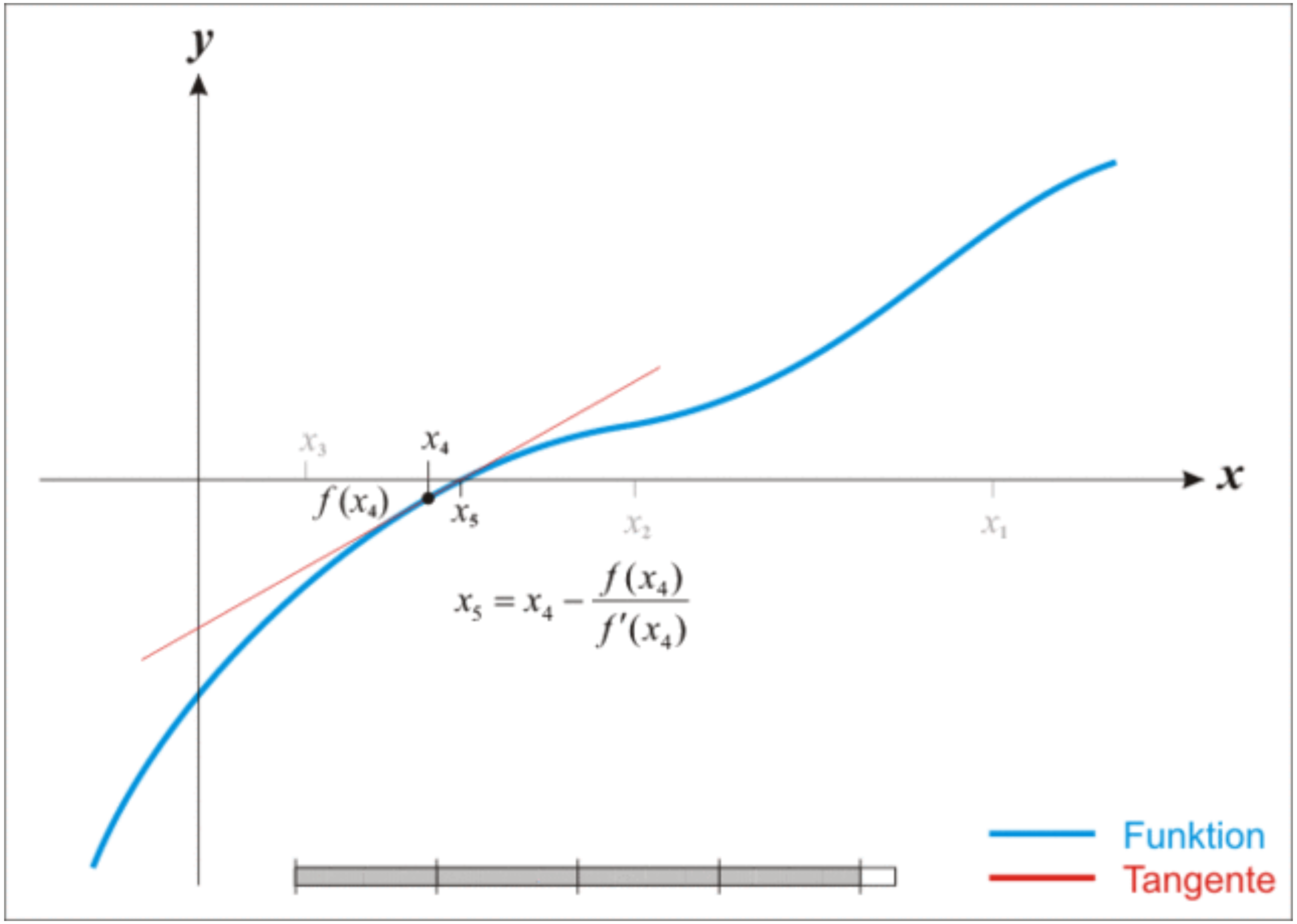

y
x
x_3
$f(x_3)$
x_2
x_1
Funktion
Tangente

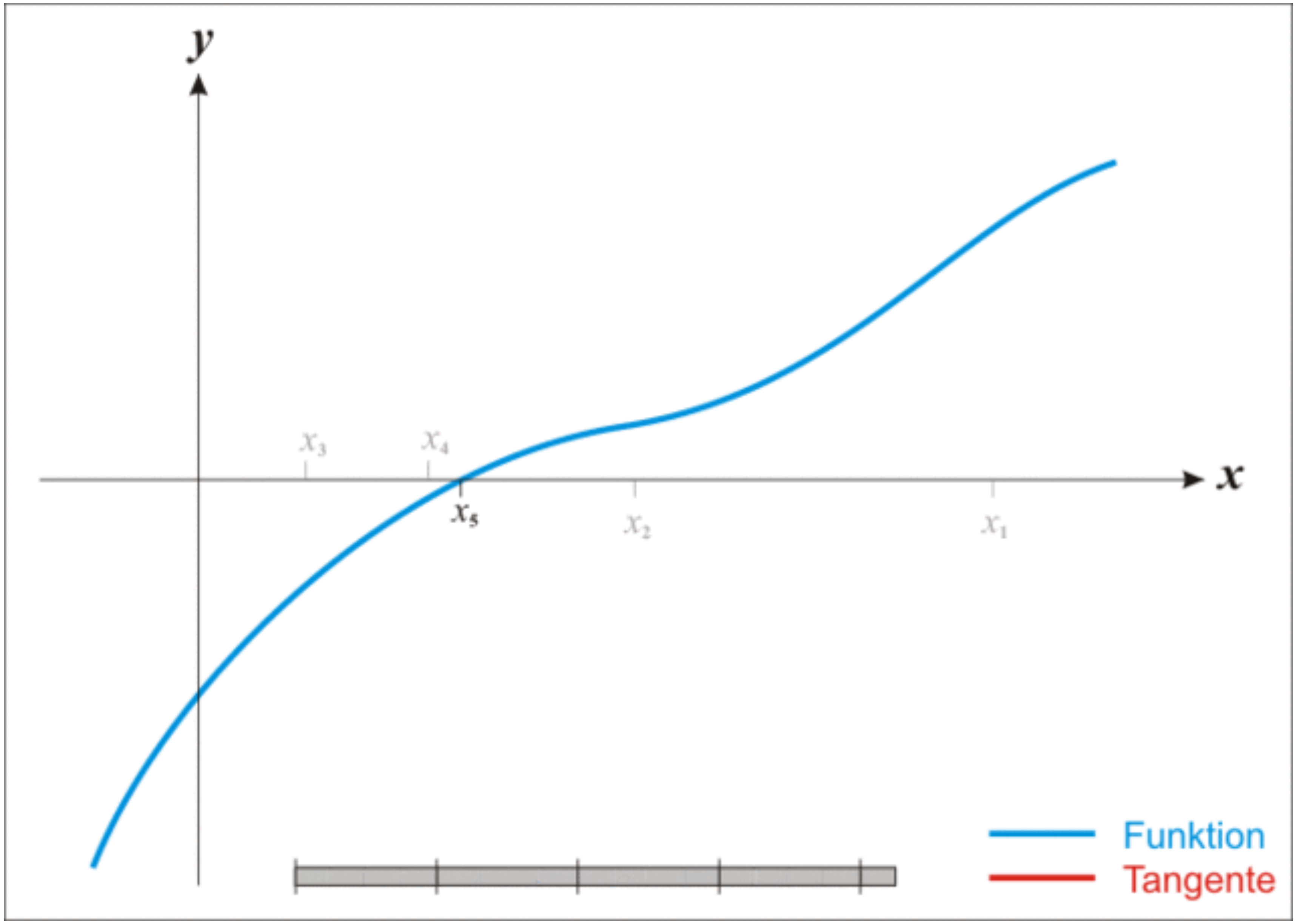

y
$x_4 = x_3 - \frac{f(x_3)}{f'(x_3)}$
x_3
x_4
x
$f(x_3)$
x_2
x_1
Funktion
Tangente


y
x
x_3
x_4
x_2
x_1
Funktion
Tangente


y
x
x_3
x_4
$f(x_4)$
x_2
x_1
Funktion
Tangente


y
x
x_3
x_4
$f(x_4)$
x_2
x_1
Funktion
Tangente


y
x
x_3
x_4
$f(x_4)$
x_5
x_2
x_1
$x_5 = x_4 - \frac{f(x_4)}{f'(x_4)}$
Funktion
Tangente



y
x
x_3
x_4
x_5
x_2
x_1
Funktion
Tangente

Applying Newton's Method to Finding Square Roots

- We can view the process of finding the square root of a number y as finding a solution to the equation:

$$x^2 = y$$

Applying Newton’s Method to Finding Square Roots

- We can view the process of finding the square root of a number y as finding a solution to the equation:

$$x^2 - y = 0$$

Applying Newton's Method to Finding Square Roots

- Equivalently, we want to find a zero to the function:

$$f(x) = x^2 - y$$

Newton’s Method

- Plugging in our function `f`:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Newton's Method

- Plugging in our function `f`:

$$x_{n+1} = x_n - \frac{x_n^2 - y}{2x_n}$$

Newton’s Method

```
def abs(x: Double) = if (x < 0) -x else x
def square(x: Double) = x * x
```

Newton’s Method

- To encode Newton’s Method as an application of generative recursion:
  - We need to choose an initial guess
  - We need to encode computation of the next guess from our current guess
  - We need to determine our base case

Newton’s Method

- For square roots:
    - Our initial guess can be the parameter
    - Our base case is that our current guess falls within some tolerance of the true square root

Newton’s Method

```
def next(guess: Double): Double =
  if (isGoodEnough(guess)) guess
  else next(guess - ((square(guess) - x) /
                    (2 * guess)))
```

Newton’s Method

```scala
val epsilon = 0.000000000000001

def isGoodEnough(guess: Double) =
  abs(square(guess) - x) <= epsilon
```

Newton’s Method

```
def sqrt(x: Double) = {
  val epsilon = 0.000000000000001

  def isGoodEnough(guess: Double) =
    abs(square(guess) - x) <= epsilon

  def next(guess: Double): Double =
    if (isGoodEnough(guess)) guess
    else next(guess - ((square(guess) - x) /
                      (2 * guess)))

  next(x)
}
```

Generalizing to an Arbitrary Function

```
def newtonsMethod(f: Double => Double) = {
  val epsilon = 0.000000000000001
  val delta = 0.000000001

  def isGoodEnough(guess: Double) = abs(f(guess)) <= epsilon

  def fPrime(x: Double) = (f(x + delta) - f(x)) / delta

  def next(guess: Double): Double = {
    if (isGoodEnough(guess)) guess
    else next(guess - f(guess) / fPrime(guess))
  }
  next(2)
}
```

Generalizing to an Arbitrary Function

```
> newtonsMethod((x: Double) => x*x - 2)
res1: Double = 1.414213562373095

> newtonsMethod((x: Double) => x*x*x - 1000)
res0: Double = 10.0
```

Not All Applications of Newton's Method Terminate

- Consider:

$$f(x) = x^2 - x$$

$$f'(x) = 2x - 1$$

- An initial guess of 0.5 leads us to find the root of a tangent with slope zero (which has no root!)

Not All Applications of Newton's Method Terminate

`newtonsMethod((x: Double) => x*x - x)` $\mapsto \bot$

Design Recipe for Generative Recursion

- Data analysis and design
- Contract, purpose, header: Should now include some description of how the function works
- Examples: Include examples that illustrate how the function proceeds (not just input/output)

Design Recipe for Generative Recursion

- Template:
  - What is trivially solvable?
  - We new sub-problems do we generate?
  - How do we combine solutions to the sub-problems?
- Tests
- A termination argument

A Termination Argument

- With structural recursion, the computation follows the structure of the data
- Because immutable data has no cycles, the computation is certain to terminate
- With generative recursion, the sub-problems might be as large as the original problem
- Thus, we should include an explicit argument that the algorithm terminates