Solving Equations (Finding Zeros)

We will look at ways of solving equations using computers — either symbolically or numerically.

There are many kinds of equations:

- Equations in one real variable, with one or several solutions.
- Sets of equations in several real variables.
- Equations with variables that are integers, or are complex.
- Differential equations, others...

We will look at equations in one real variable. We can always rewrite such an equation as

$$f(x) = 0$$

So we can also see our problem as that of finding the zeros of a function.

Why Solve Equations?

Solving equations is often necessary, in both practical and theoretical applications. Here are some direct uses:

- Consider a robot arm with two joints, with joint angles θ and ϕ, which we can control. The end-point of the arm will be at

$$
\begin{aligned}
x &= a\cos(\theta) + b\cos(\theta+\phi) \\
y &= a\sin(\theta) + b\sin(\theta+\phi)
\end{aligned}
$$

  To position the arm at some desired (x, y), we must solve these equations for θ and ϕ.

- Suppose you have a model of how world population and food production will change in the future. You could determine when in the future food production will first fall below the amount required by the population by solving an equation.

Minimization & Maximization

Problems of minimizing or maximizing a smooth function can be solved by finding the zeros of its derivative.

To find the x that maximizes $f(x)$, we find the solutions to $f'(x) = 0$, then pick the solutions where $f''(x) < 0$. These are local maxima.

Maximization and minimization problems arise in many applications:

- A firm with a model of its customers, competitors, and production costs can try to find the price for its product that will maximize its profits.

- Statisticians find "maximum likelihood" estimates for unknown quantities by finding the value that maximizes a measure of fit to the observed data.

Solving Equations Symbolically

We can sometimes solve equations symbolically by hand, or by using Maple:

```
> solve(x^2+x-1=0,x);
                                   1/2                 1/2
                  - 1/2 + 1/2 5   , - 1/2 - 1/2 5

> solve(sin(x+1)=1,x);
                                  1/2 Pi - 1

> solve({x+2*y=1,y+1=x^2},{x,y});
                  {x = 1, y = 0}, {y = 5/4, x = -3/2}
```

But it doesn't always work:

```
> solve(x^3+sin(1+x)=0,x);
```

nothing printed

The null response indicates that Maple couldn't find any solutions, but a solution does exist!

Solving Equations Numerically

When symbolic solution fails, we can use a method that gives an approximate numerical answer. Maple does this with `fsolve`:

```
> fsolve(x^3+sin(1+x)=0,x);

                                  -.6800575892
```

Here, numerical solution is used to find a maximum:

```
> y:=-x^4+10*x^2+20*x;
                                   4        2
                         y := - x   + 10 x   + 20 x

> fsolve(diff(y,x)=0,x);

                                  2.627365085

> subs(x=",diff(y,x$2));

                                -62.83656748
```

Since the second derivative is negative, the point where the derivative is zero is a maximum, rather than a minimum.

Methods for Numerical Solution

There are many ways of trying to find numerical solutions. We will look at:

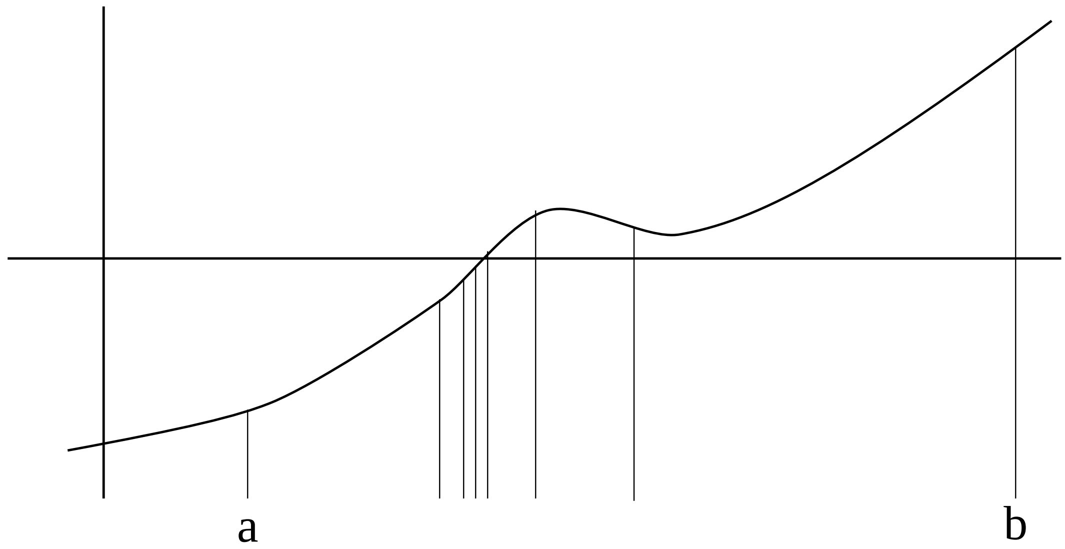
- *Bisection* — A simple and robust method, but not all that fast.
- *Newton-Raphson Iteration* — A much faster method, when it works.
- *The Secant Method* — More robust than Newton-Raphson, but not quite as fast.

None of these methods is perfect. It is very hard to guarantee that a numerical method will always find all solutions.

Finding Zeros by Bisection

Suppose that we know $f(x)$ is continuous, and we have found values a and b such that $f(a) \leq 0$ and $f(b) \geq 0$. Then we can be sure that f has a zero somewhere between a and b.

We can find such a zero by *bisecting* the interval $[a, b]$, until we are close enough:



This is a very robust procedure — nothing much can go wrong.

However, we'll find only one zero this way! There could be more. It's also not very fast.

A Bisection Algorithm in Maple

```
bisection := proc(f,x,rng,tolerance)

    local lx, hx, mx, lf, hf, mf;

    lx := evalf(op(1,rng));
    hx := evalf(op(2,rng));

    lf := evalf(eval(subs(x=lx,f)));
    hf := evalf(eval(subs(x=hx,f)));

    if lf=0 then RETURN(lx) fi;
    if hf=0 then RETURN(hx) fi;

    do # loop until we find zero, or interval is smaller than tolerance

        mx := (lx+hx) / 2;

        if abs(mx-lx)<tolerance or abs(mx-hx)<tolerance then
            RETURN(mx);
        fi;

        mf := evalf(eval(subs(x=mx,f)));

        if mf=0 then
            RETURN(mx);
        fi;

        if mf>0 and lf>0 or mf<0 and lf<0 then
            lx := mx;
        else
            hx := mx;
        fi;
    od;
end;
```

Finding Zeros by Using a Model of the Function

How can we do better than bisection? One approach:

1. Build a “model” of the function, based on points where we have values.

2. Compute a zero of the model function, and take that as a guess for a zero of the real function.

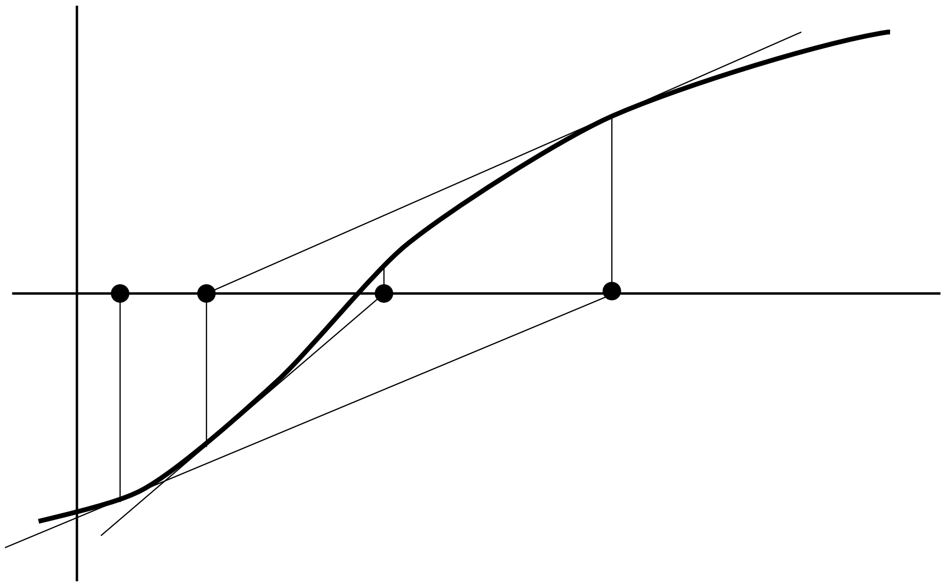
3. Compute the value of the real function at this point, and go back to step (1).

Building a model is much like approximation by Taylor polynomials, and also like interolation.

Using a First-Degree Model: Newton-Raphson Iteration

We can build a first-degree model of a function using the function's value and derivative at one point.

Repeating this procedure, always building a model based on the last point, gives the method of *Newton-Raphson iteration*.



We get the next guess, x_{i+1}, from the previous guess, x_i, as follows:

$$x_{i+1} \;=\; x_i \;-\; f(x_i) \;/\; f'(x_i)$$

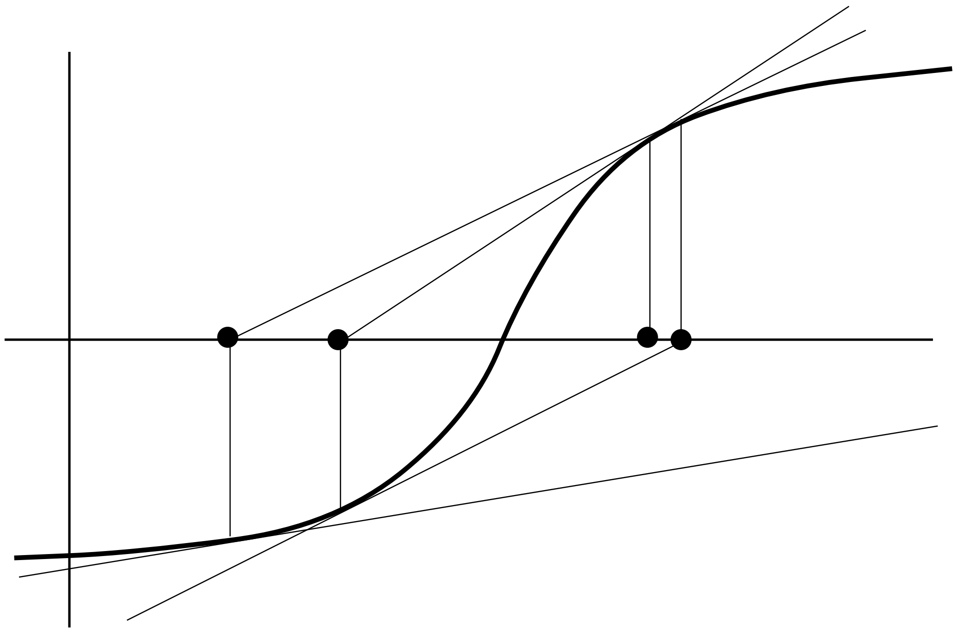
Newton-Raphson Iteration in Maple

```
newton := proc(f,x,start,tolerance,maxi)

    local df, z, zf, zd, oz, i;

    if tolerance<=0 then
        ERROR(`Fourth operand must be a positive tolerance value`);
    fi;

    if not type(maxi,integer) or maxi<=0 then
        ERROR(`Fifth operand must be the maximum number of iterations`);
    fi;

    df := diff(f,x);

    z := evalf(start);

    for i from 1 to maxi do

        oz := z;

        zf := evalf(subs(x=z,f));
        zd := evalf(subs(x=z,df));

        z := z - zf/zd;

        if abs(z-oz)<tolerance then
            RETURN(z);
        fi;
    od;

    ERROR(`Zero not found after maximum number of iterations`);

end;
```

When Does Newton Iteration Work?

Newton-Raphson iteration requires that the function be differentiable (and that we can compute the derivatives).

Unfortunately, Newton-Raphson iteration is not guaranteed to converge even if the derivatives do exist. For example:



If we start close enough, however, it does converge for this example.

We might also have problems if the derivative is zero at the solution.

The Secant Method

What if we don't know how to compute derivatives, and so can't use Newton-Raphson iteration?

The *Secant Method* builds a first-degree model of the function using the last two function values.

Given starting values x_0 and x_1, the secant method proceeds iteratively as follows:

$$x_{i+1} \;=\; x_i \;-\; f(x_i)\,\frac{x_i - x_{i-1}}{f(x_i) - f(x_{i-1})}$$

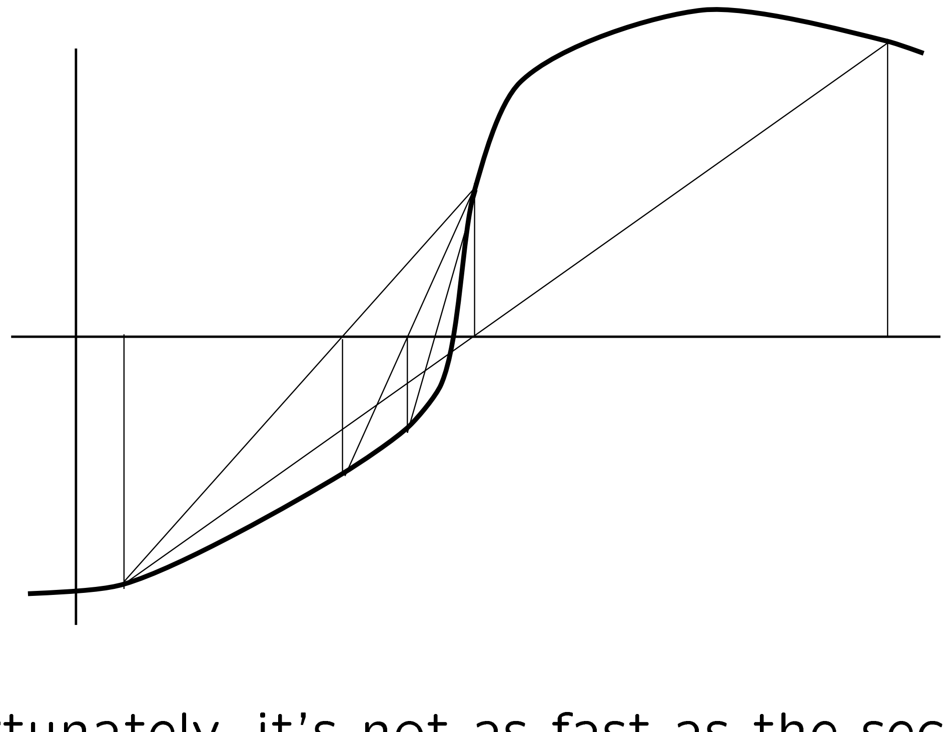
This is not as fast as Newton-Raphson iteration, but it's faster than bisection.

This method can also fail to converge, however, just like Newton-Raphson iteration.

The Secant-Bracket Method

Suppose that we have starting points that bracket a zero, as we assumed for the bisection algorithm. Can we do better than bisection while still guaranteeing convergence?

The *Secant-Bracket Method* builds a first-degree model based on the two bracketing points:



Unfortunately, it's not as fast as the secant method.

Other Methods

A huge number of other methods are possible, using higher-degree models of the function based on various pieces of information. For example:

- We could build a second-degree model based on the last three function values (Muller's method).

- We could build a second-degree model based on the function value, the first derivative, and the second derivative at the last point.

- We could build a third-degree model based on the function values and derivatives at the last two points.

How to Choose a Method

In choosing what method to use, we need to look at three things:

1. The information needed to use the method. Just function values, or also derivatives? Is one starting point needed, or two (perhaps bracketing a zero)?

2. How robust the method is. Ie, does it sometimes fail to find a zero that exists?

3. How fast the method is at finding a zero.

Hybrid methods are often used to try to get the advantages of several methods.

The Convergence Rate of Bisection

How accurate is the result of using bisection for n iterations?

Suppose we start with an interval of size I that contains a zero. If we did no iterations, just guessed the mid-point of this interval, our error would be no more than

$$\epsilon_0 = \frac{I}{2}$$

Each iteration cuts the size of the interval in half, so

$$\epsilon_{i+1} = \frac{1}{2}\,\epsilon_i$$

After the n iterations, guessing the mid-point would give an error of no more than

$$\epsilon_n = \frac{I}{2}\,2^{-n}$$

Convergence Rates in Terms of Number of Digits of Accuracy

We can translate a bound on the error after n iterations to the number of digits of accuracy after n iterations.

We just take logs (to base 10): An error of no more than ϵ_n means the result is accurate to about $-\log_{10}(\epsilon_n)$ digits after the decimal point. (Accuracy in terms of total digits will depend on the magnitude of the answer.)

Eg: If the error is no more than 0.0001, the result will be accurate to $-\log_{10}(0.0001) = 4$ digits after the decimal point.

For bisection:

$$-\log_{10}(\epsilon_n) \ = \ -\log_{10}(I/2) \ + \ n\log_{10}(2)$$

The number of digits of accuracy goes up linearly with n.

Convergence Rate of Newton Iteration

Suppose after i iterations the Newton-Raphson method has found a point, x_i, that is fairly close to the true zero, x_*. Let $\epsilon_i = |x_i - x_*|$ be the error. What will the error be after one more iteration?

We can use Taylor's theorem to conclude that

$$f(x_*) \;=\; f(x_i) + f'(x_i)\,(x_* - x_i) + \frac{f''(c)}{2}\,(x_* - x_i)^2$$

for some c between x_* and x_i.

Since $f(x_*) = 0$, this can be rearranged into

$$\left[x_i \;-\; \frac{f(x_i)}{f'(x_i)}\right] \;-\; x_* \;=\; \frac{f''(c)}{2f'(x_i)}\,(x_* - x_i)^2$$

The left side is equal to $x_{i+1} - x_*$. From this, we get

$$\epsilon_{i+1} \;=\; \left|\frac{f''(c)}{2f'(x_i)}\right|\,\epsilon_i^2$$

If f has a continuous second derivative, the constant above will not vary much once x_i is close to x_*. The error decreases quadratically.

The Benefits Obtained from "Super-Linear" Convergence

For bisection: $\epsilon_{i+1} = C\,\epsilon_i$

For Newton iteration: $\epsilon_{i+1} = C\,\epsilon_i^2$

For some other methods: $\epsilon_{i+1} = C\,\epsilon_i^p$
with p between 1 and 2.

How many digits accuracy after n iterations?

If $\epsilon_{i+1} = C\,\epsilon_i^p$, then

$$[-\log_{10}(\epsilon_{i+1})] = -\log_{10}(C) + p\,[-\log_{10}(\epsilon_i)]$$

which we can write as $D_{i+1} = K + pD_i$.

For $p = 1$, the number of digits of accuracy grows linearly.
For $p = 2$, the number of digits of accuracy approximately *doubles* each iteration (for large i).

For any $p > 1$, the number of iterations to get D digits of accuracy is proportional to $\log(D)$, when D is large.