LECTURE 10: UNDIMENSIONAL ROOT FINDING

Rootfinding and optimisation

Introduction

Polynomial

$$f(x) = ax + b$$

Solution

$$x = -\frac{b}{a}$$

Polynomial

$$f(x) = ax^2 + bx + c$$

Solution

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Polynomial

$$f(x) = ax^3 + bx^2 + cx + d$$

Solution

$$x = \sqrt[3]{\left(\frac{-b^3}{27a^3} + \frac{bc}{6a^2} - \frac{d}{2a}\right) + \sqrt{\left(\frac{-b^3}{27a^3} + \frac{bc}{6a^2} - \frac{d}{2a}\right)^2 + \left(\frac{c}{3a} - \frac{b^2}{9a^2}\right)^3}} + \sqrt[3]{\left(\frac{-b^3}{27a^3} + \frac{bc}{6a^2} - \frac{d}{2a}\right) - \sqrt{\left(\frac{-b^3}{27a^3} + \frac{bc}{6a^2} - \frac{d}{2a}\right)^2 + \left(\frac{c}{3a} - \frac{b^2}{9a^2}\right)^3}} - \frac{b}{3a}$$

Polynomial

$$f(x) = ax^4 + bx^3 + cx^2 + dx + e$$

Solution

$$x = \ldots$$

Polynomial

$$f(x) = ax^5 + bx^4 + cx^3 + dx^2 + ex + f$$

Solution

does not exist (Abel, 1826)

Polynomial

$$f(x) = ax^5 + bx^4 + cx^3 + dx^2 + ex + f$$

Solution

has closed-form solution iff its *Galois group* is *solvable* (Galois, 1846)

Rootfinding and optimisation

Introduction

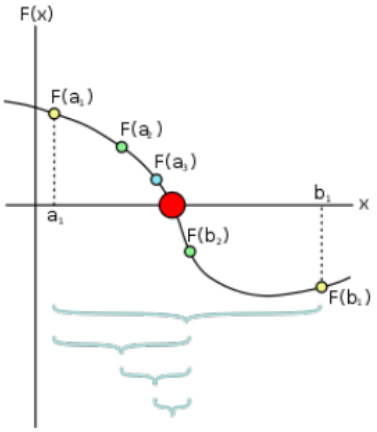
- Motivation
  - $\cos(x) = x$ has no "closed form" solution.
  - $ax^2 + bx + c = 0$ can suffer from cancellation when $b^2 \approx 4ac$.
  - $x^2 = 2$; well, what *is* $\sqrt{2}$ anyway? (i.e., how could we compute it?)
  - Model problem for harder problems (PDEs)
- Closed form solutions may:
  - Not exist.
  - Be unstable.
  - Require evaluation of special functions.
- Notation:

  Given $f : [a, b] \to \mathbb{R}$, find $c \in [a, b]$ st. $f(c) = 0$.
  - (Note, we will limit ourselves to the case where f is continuous.)

Technique #1: Bisection

- Recall the intermediate value theorem:
  If $f : [a, b] \to \mathbb{R}$ is cts and $f(a) < u < f(b)$ or $f(b) < u < f(a)$,
  then $\exists c \in (a, b)$ st. $f(c) = u$.

- Bisection method:
  - Subdivide interval iteratively.
  - Keep the half where f changes sign.



Bisection: pseudo code

```
function c = bisect(f, a, b, ...)
C = BISECT(F, A, B) finds a zero of F in [A, B] using bisection.
fa = f(a); fb = f(b);
while ...                  % <-- to be defined
    c = (a-b)/2; fc = f(c);
    if ( fc == 0 )
        break              % lucky!
    elseif ( sign(fc) == sign(fa) )
        a = c; fa = fc; % keep right interval!
    else
        b = c; fb = fc; % keep left interval!
    end
end
```

Bisection: Details

- Termination criteria (i.e., where to stop?)
  - $|b - a| < \text{tol.}$ (solution tolerance)
  - $|f(c)| < \text{tol.}$ (residual tolerance)
- Typically one or more of these are used. (Choice is often problem specific!)
- Convergence
  - $c \in [a, b] \Rightarrow$ maximum error in c is $b - a$, i.e., the width of the interval.
  - Each step halves the interval and hence the error at the nth iteration is:

$$|c_n - c| \leq \frac{b-a}{2^n}.$$

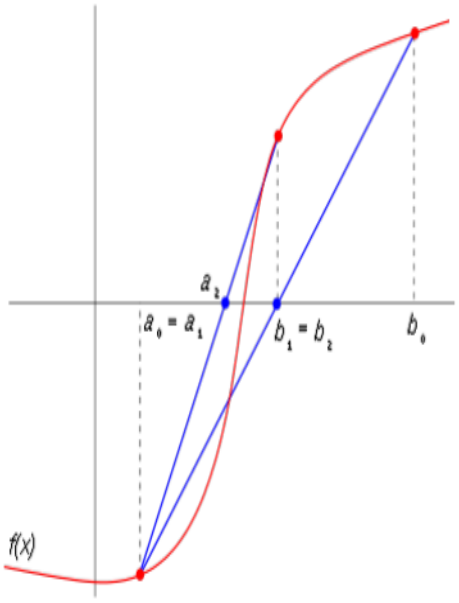
  - $\therefore$ each iteration gives one more bit of precision $\Rightarrow M_\varepsilon$ in ~ 52 steps.
  - Number of iterations for ε accuracy is $n^{\varepsilon}_{\max} = \log_2\left(\frac{b-a}{2}\right) = \frac{\log(b-a)-\log(\varepsilon)}{\log(2)}$.
- Demo: `bisection(@(x) sin(x) + exp(sin(x)), -1, 1);`

Technique #2: "Regula Falsi"

- (Or "false position", if you prefer!)
- Interpolate f on $[a, b]$ by a linear function

$$g_1(x) = f(a)\frac{x-b}{a-b} + f(b)\frac{x-a}{b-a}$$

- Then g_1 has a root at $c = \frac{af(b)-bf(a)}{f(b)-f(a)}$.
- Choose this as c in bisection code!



- Here we're making more use of f; bisection simply used sign of $f(a)$ and $f(b)$. Here we're using their magnitude.

(Recall) Bisection: pseudo code

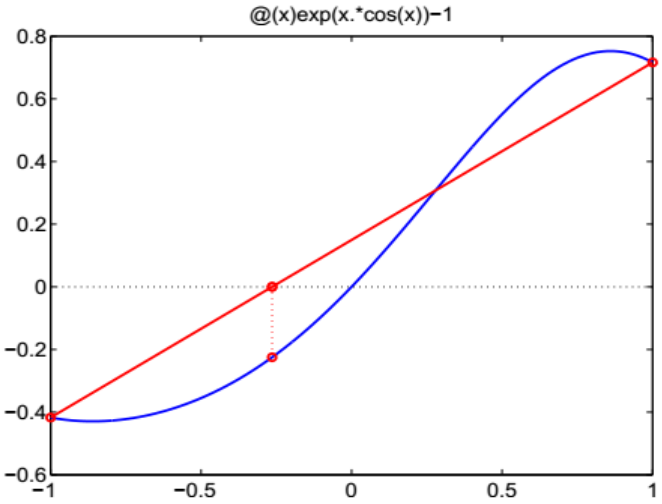
```
function c = bisect(f, a, b, ...)
C = BISECT(F, A, B) finds a zero of F in [A, B] using bisection.
fa = f(a); fb = f(b);
while ...                  % <-- to be defined
    c = (a-b)/2; fc = f(c);
    if ( fc == 0 )
        break              % lucky!
    elseif ( sign(fc) == sign(fa) )
        a = c; fa = fc; % keep right interval!
    else
        b = c; fb = fc; % keep left interval!
    end
end
```

Regula falsi: pseudo code

```
function c = regulaFalsi(f, a, b, ...)
C = REGULAFALSI(F, A, B) finds a zero of F in [A, B] using RF.
fa = f(a); fb = f(b);
while ...                  % <-- to be defined
    c = (a*fb-b*fa)/(fb-fa); fc = f(c);
    if ( fc == 0 )
        break              % lucky!
    elseif ( sign(fc) == sign(fa) )
        a = c; fa = fc; % keep right interval!
    else
        b = c; fb = fc; % keep left interval!
    end
end
```

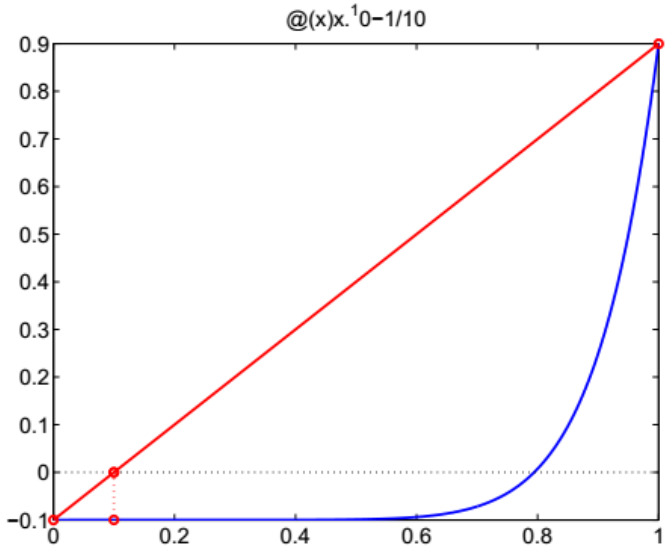
Regula Falsi: When it works...

```
>> f = @(x) exp(x.*cos(x))-1
>> regulaFalsi(f, -1, 1);
```



Regula Falsi: When it doesn't...

```
>> f = @(x) x.^10 - 1/10
>> regulaFalsi(f, 0, 1);
```



Regula Falsi: When it doesn't...

```
>> f = @(x) x^10 - 1/10
```

- Convergence can be slow when curvature of f is large.
  (the linear approx. is no good in this case!)
- But there is a simple fix! - 'combine' bisection and Regula Falsi.
  - Detect if one of a or b are not changing
  - If algorithm goes right twice, then halve f(b)
  - i.e.,

$$c = \frac{af(b)/2 - bf(a)}{f(b)/2 - f(a)}.$$

  - (similarly, If algorithm goes left twice, then halve f(a))
- This modified Regula Falsi is sometimes called the "Illinois" algorithm.
- It looks like a hack, but it really works! (superlinear convergence)
- Demo: `illinois(@(x) x.^10 - 1/10, 0, 1);`

(Recall) Regula falsi: pseudo code

```
function c = regulaFalsi(f, a, b, ...)
C = REGULAFALSI(F, A, B) finds a zero of F in [A, B] using RF.
fa = f(a); fb = f(b);
while ...                % <-- to be defined
    c = (a*fb-b*fa)/(fb-fa); fc = f(c);
    if ( fc == 0 )
        break            % lucky!
    elseif ( sign(fc) == sign(fa) )
        a = c; fa = fc; % keep right interval!
    else
        b = c; fb = fc; % keep left interval!
    end
end
```

Illinois: pseudo code

```
function c = regulaFalsi(f, a, b, ...)
C = ILLINOIS(F, A, B) finds a zero of F in [A, B] using Illinois.
fa = f(a); fb = f(b);
side = 0;
while ...                % <-- to be defined
    c = (a*fb-b*fa)/(fb-fa); fc = f(c);
    if ( fc == 0 )
        break            % lucky!
    elseif ( sign(fc) == sign(fa) )
        a = c; fa = fc; % keep right interval!
        if ( side == 1 ), fb = fb/2; else side = 1; end
    else
        b = c; fb = fc; % keep left interval!
        if ( side == -1 ), fa = fa/2; else side = -1; end
    end
end
```

Technique #3 - Newton-Raphson

- Newton (1669 - for polynomials), Raphson (1690 - general functions)
- Instead of interpolating polynomial through different points interpolate different derivatives at the same point.
- Newton-Raphson interpolates $f(x_k)$ and $f'(x_k)$ by a linear function

$$\begin{aligned} g_1(x) &= \alpha_0 + \alpha_1 x \quad \text{st.} \begin{cases} g_1(x_k) = f(x_k), \\ g_1'(x_k) = f'(x_k), \end{cases} \\ \Rightarrow g_1(x) &= f(x_k) + f'(x_k)(x - x_k) \end{aligned}$$

- (Another interpretation is that this is the truncated Taylor series)
- Root of $g_1(x)$ is then given simply by $c = x_0 - f(x_0)/f'(x_0)$.
- This leads to the system (Simpson, 1740): $x_{k+1} = x_k - f(x_k)/f'(x_k)$.
- Convergence is quadratic if "sufficiently close" to the solution.
- This means the number of correct digits doubles at each iteration.
- Demo: `newton(@(x)sin(x)+exp(sin(x)), @(x)cos(x)+cos(x).*exp(sin(x)), 0);`

NEWTON RAPHSON – PRACTICAL CONSIDERATIONS

- Practical considerations:
  - How close is close? (There's no real answer!)
  - We need to know $f'(x)$.
  - What if $f'(s_k) = 0$ or $f'(s) = 0$?

- Modifications:
  - Secant method: $x_{k+1} = x_k - f_k \frac{x_k - x_{k-1}}{f_k - f_{k-1}}$.
  - Damped Newton: $x_{k+1} = x_k - \sigma_k \frac{f_k}{f'_k}$.
  - Halley's method: $x_{k+1} = x_k - \frac{2f_k f'_k}{2(f'_k)^2 - f_k f''_k}$.
  - Optimisation: $x_{k+1} = x_k - \frac{f'_k}{f''_k}$.

SUMMARY:

- We've seen a number of approaches:
  - Bisection - slow, but guaranteed.
  - Regula Falsi / Illinois - faster, but can stagnate.
  - Newton-Raphson - very fast, requires f' and x_0, "interesting" convergence behaviour.
- Each of the above use interpolation! (Implicitly or explicitly)
- Best choice is often problem dependent. Black-box routines use a selection.
- `fzero` uses Bisection, Secant method, and Inverse quadratic interpolation.
- Global optimisation is hard! Use different initial guesses or a global proxy.
- In higher dimensions things get much more complicated!
  (Of those we've seen, only Newton applies).