Prepared by Julian García, based on material by
Arun Konagurthu

# Lecture 7
# Solving Non-linear equations

## FIT 3139
## Computational Modelling and Simulation

# Outline

- Non-linear equations
- Some standard methods to solve non-linear equations
  - Bisection
  - Newton
  - Secant
- Issues related to numerical “convergence”.

# Bisection method

```
def bisection(a_function, a, b, tol, max_iter=1000):
    assert np.sign(a_function(a)) != np.sign(a_function(b))
    it = 1
    while it <= max_iter:
        c = (a+b)/2
        if abs(b-a) < tol:
            return c
        it = it + 1
        if np.sign(a_function(c)) == np.sign(a_function(a)):
            a = c
        else:
            b = c
raise ValueError("Maximum iterations achieved")
```

| Iteration | a | b | f(c) | la-bl | |
|---|---|---|---|---|---|
| | 1 | 1 | 3 | 0.36281029 | 2 |
| | 2 | 1 | 2 | -1.7399799 | 1 |
| | 3 | 1.5 | 2 | -0.8734438 | 0.5 |
| | 4 | 1.75 | 2 | -0.3007181 | 0.25 |
| | 5 | 1.875 | 2 | 0.01984913 | 0.125 |
| | 6 | 1.875 | 1.9375 | -0.1432552 | 0.0625 |
| | 7 | 1.90625 | 1.9375 | -0.0624057 | 0.03125 |
| | 8 | 1.921875 | 1.9375 | -0.0214536 | 0.015625 |
| | 9 | 1.9296875 | 1.9375 | -0.000846 | 0.0078125 |
| | 10 | 1.93359375 | 1.9375 | 0.00949061 | 0.00390625 |
| | 11 | 1.93359375 | 1.93554688 | 0.00431956 | 0.00195313 |
| | 12 | 1.93359375 | 1.93457031 | 0.00173608 | 0.00097656 |
| | 13 | 1.93359375 | 1.93408203 | 0.00044486 | 0.00048828 |
| | 14 | 1.93359375 | 1.93383789 | -0.0002006 | 0.00024414 |
| | 15 | 1.93371582 | 1.93383789 | 0.00012211 | 0.00012207 |
| | 16 | 1.93371582 | 1.93377686 | -3.93E-05 | 6.10E-05 |

# Issues to consider

Unlike linear equations, most **nonlinear equations <u>CANNOT</u>** be solved in a **finite number of steps**.

One resorts to **iterative methods**, that produce increasingly accurate **approximations** to a solution.

The process terminates once the result is **“sufficiently”** accurate, as determined by a given tolerance.

# Convergence rate

| Iteration | a | b | f(c) | |a-b| |
|---|---|---|---|---|
| 1 | 1 | 3 | 0.36281029 | 2 |
| 2 | 1 | 2 | -1.7399799 | 1 |
| 3 | 1.5 | 2 | -0.8734438 | 0.5 |
| 4 | 1.75 | 2 | -0.3007181 | 0.25 |
| 5 | 1.875 | 2 | 0.01984913 | 0.125 |
| 6 | 1.875 | 1.9375 | -0.1432552 | 0.0625 |
| 7 | 1.90625 | 1.9375 | -0.0624057 | 0.03125 |
| 8 | 1.921875 | 1.9375 | -0.0214536 | 0.015625 |
| 9 | 1.9296875 | 1.9375 | -0.000846 | 0.0078125 |
| 10 | 1.93359375 | 1.9375 | 0.00949061 | 0.00390625 |
| 11 | 1.93359375 | 1.93554688 | 0.00431956 | 0.00195313 |
| 12 | 1.93359375 | 1.93457031 | 0.00173608 | 0.00097656 |
| 13 | 1.93359375 | 1.93408203 | 0.00044486 | 0.00048828 |
| 14 | 1.93359375 | 1.93383789 | -0.0002006 | 0.00024414 |
| 15 | 1.93371582 | 1.93383789 | 0.00012211 | 0.00012207 |
| 16 | 1.93371582 | 1.93377686 | -3.93e-05 | 6.10e-05 |

**By how much do I reduce the error with each iteration**

# Convergence rate

True solution

Let $e_k = x_k - x^*$ be the error at iteration $k$

Approximation after k iterations

An **iterative** method is said to converge with **rate** $\mathbf{r}$ if:

$$\lim_{k\to\infty} \frac{|e_{k+1}|}{|e_k|^{\mathbf{r}}} = C$$

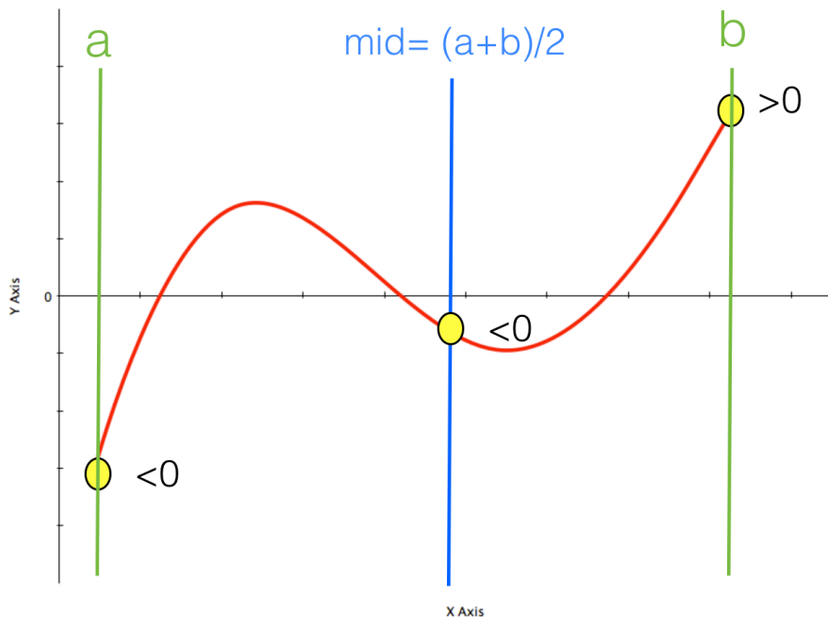
for some constant $C > 0$

If $r=1$ and $C < 1$, the convergence rate is **linear**.

If $r > 1$, the convergence rate is **superlinear**.

If $r = 2$, the convergence rate is **quadratic**.

# Convergence rate for the bisection method.




- We can bound the error by the size of the interval at each iteration.
- The interval is reduced by half in each iteration

$$\lim_{k\to\infty} \frac{|e_{k+1}|}{|e_k|^{\mathbf{1}}} = 0.5 \longrightarrow$$

**Linear convergence**
(i.e., not great)

# Bisection method

## Advantages

- Error is bounded by the interval at each step.
- Always converges.

## Disadvantages

- Only works for functions that are continuous (in the initial bracket)
- Does not use magnitudes, only signs. Thus, slow.
- Cannot find roots in some equations (e.g, $x^2+\alpha$, $\alpha > 0$).

# scipy.optimize.bisect

**scipy.optimize.bisect(***f, a, b, args=(), xtol=2e-12, rtol=8.881784197001252e-16, maxiter=100, full_output=False, disp=True***)** [source]

Find root of a function within an interval.

Basic bisection routine to find a zero of the function *f* between the arguments *a* and *b*. *f(a)* and *f(b)* cannot have the same signs. Slow but sure.

**Parameters:**

**f : *function***

Python function returning a number. *f* must be continuous, and f(a) and f(b) must have opposite signs.

**a : *number***

One end of the bracketing interval [a,b].

**b : *number***

The other end of the bracketing interval [a,b].

**xtol : *number, optional***

The computed root `x0` will satisfy `np.allclose(x, x0, atol=xtol, rtol=rtol)`, where `x` is the exact root. The parameter must be nonnegative.

**rtol : *number, optional***

The computed root `x0` will satisfy `np.allclose(x, x0, atol=xtol, rtol=rtol)`, where `x` is the exact root. The parameter cannot be smaller than its default value of `4*np.finfo(float).eps`.

**maxiter : *number, optional***

if convergence is not achieved in *maxiter* iterations, an error is raised. Must be >= 0.

**args : *tuple, optional***

containing extra arguments for the function *f*. *f* is called by `apply(f, (x)+args)`.

**full_output : *bool, optional***

If *full_output* is False, the root is returned. If *full_output* is True, the return value is `(x, r)`, where x is the root, and r is a *RootResults* object.

**disp : *bool, optional***

If True, raise RuntimeError if the algorithm didn't converge.

**Returns:**

**x0 : *float***

Zero of *f* between *a* and *b*.

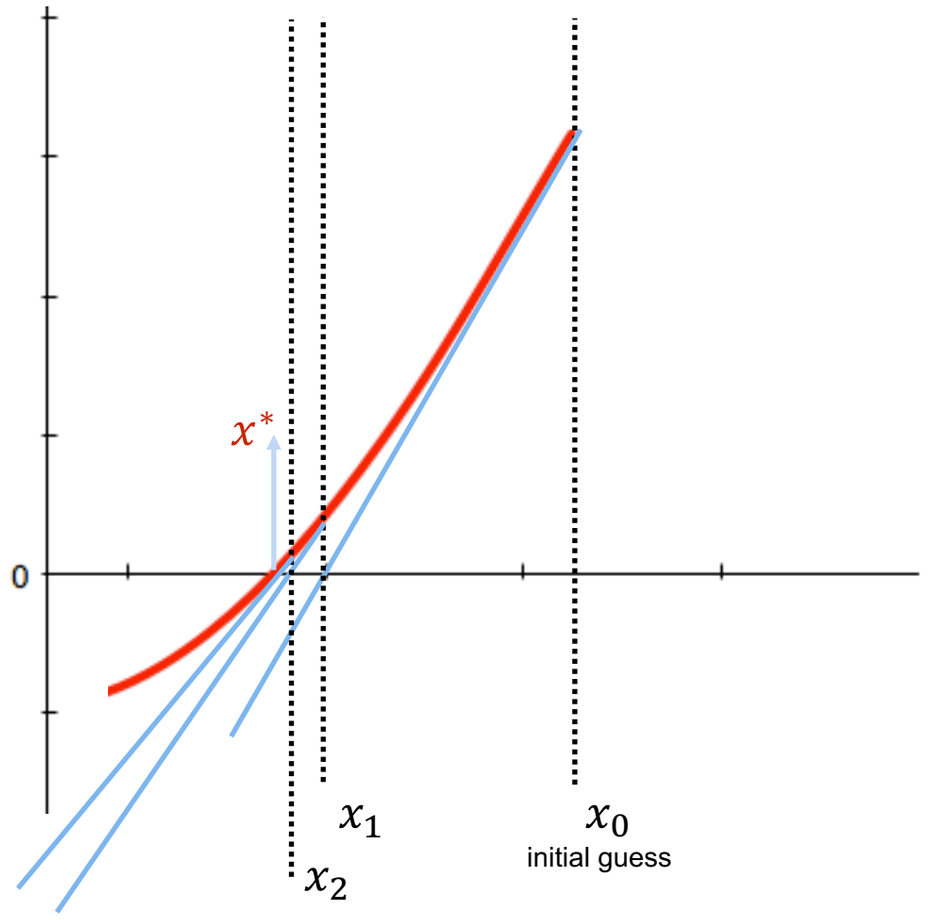
**r : *RootResults (present if `full_output = True`)***

Object containing information about the convergence. In particular, `r.converged` is True if the routine converged.

https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.bisect.html#scipy.optimize.bisect

Other than signs, we **do not use any information from the function** itself…

# Newton’s method




Until I am close enough.

# Remember Taylor…

$$f(x+h) = f(x) + f'(x)h + f''(x)\frac{h^2}{2!} + \cdots$$

$$f(x+h) = f(x) + f'(x)h + f''(x)\frac{h^2}{2!} + \cdots$$

At iteration $k$

$$f(x_k + h) \approx f(x_k) + f'(x_k)h$$

$$x^* = x_k + h \longrightarrow h = x^* - x_k$$

$$f(x^*) \approx f(x_k) + f'(x_k)(x^* - x_k)$$

$$0 \approx f(x_k) + f'(x_k)(x^* - x_k)$$

$$0 \approx \frac{f(x_k)}{f'(x_k)} + (x^* - x_k)$$

$$0 = \frac{f(x_k)}{f'(x_k)} + \frac{f'(x_k)}{f'(x_k)}(x^* - x_k)$$

$$x^* \approx x_k - \frac{f(x_k)}{f'(x_k)} \longrightarrow x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

# Newton's method — algorithm

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

1. Set $i = 0$ and choose an initial value $x_0$
2. Find $x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$
3. Find $|\epsilon_a| = |\frac{x_{i+1} - x_i}{x_i}| * 100$
4. If $|\epsilon_a| <$ `tol` **Stop.**

Else, repeat from step 2 with $i = i + 1$ and $x_i = x_{i+1}$

Convergence?

At iteration $k$

$$f(x+h) = f(x) + f'(x)h + f''(x)\frac{h^2}{2!} + \cdots$$

$$f(x_k + h) \approx f(x_k) + f'(x_k)h + f''(x_k)\frac{h^2}{2!}$$

I am close to the true value

$$x^* = x_k + h \longrightarrow h = x^* - x_k$$

$$f(x^*) \approx f(x_k) + f'(x_k)(x^* - x_k) + f''(x_k)\frac{(x^* - x_k)^2}{2}$$

$$0 \approx f(x_k) + f'(x_k)(x^* - x_k) + f''(x_k)\frac{(x^* - x_k)^2}{2}$$

$$0 \approx \frac{f(x_k)}{f'(x_k)} + (x^* - x_k) + f''(x_k)\frac{(x^* - x_k)^2}{2f'(x_k)}$$

$$0 \approx \frac{f(x_k)}{f'(x_k)} + (x^* - x_k) + f''(x_k)\frac{(x^* - x_k)^2}{2f'(x_k)}$$

$$0 \approx x^* - \underbrace{(x_k - \frac{f(x_k)}{f'(x_k)})}_{x_{k+1}} + \frac{f''(x_k)}{2f'(x_k)}(x^* - x_k)^2$$

$$0 \approx \underbrace{x^* - x_{k+1}}_{e_{k+1}} + \frac{f''(x_k)}{2f'(x_k)}\underbrace{(x^* - x_k)^2}_{e_k}$$

$$0 \approx e_{k+1} + \frac{f''(x_k)}{2f'(x_k)}e_k^2$$

$$\text{as} \quad k \to \infty \quad \longrightarrow \quad C = \frac{e_{k+1}}{e_k^2}$$

**Quadratic is faster than linear…**

Given a sequence of approximations $x_0, x_1, x_2, ...$, can we approximate the convergence rate $r$?

$$\lim_{k\to\infty} \frac{|e_{k+1}|}{|e_k|^{\mathbf{r}}} = C$$

For large $k$, we have

$$\frac{|e_{k+1}|}{|e_k|^{\mathbf{r}}} \approx C \text{ and } \frac{|e_k|}{|e_{k-1}|^{\mathbf{r}}} \approx C$$

$$\longleftarrow \qquad \frac{|e_{k+1}|}{|e_k|^{\mathbf{r}}} \approx \frac{|e_k|}{|e_{k-1}|^{\mathbf{r}}}$$

$$\vdots$$

$$\mathbf{r} \approx \frac{\log(|e_{k+1}/e_k|)}{\log(|e_k/e_{k-1}|)} = \frac{\log(|(x_{k+1}-x^*)/(x_k-x^*)|)}{\log(|(x_k-x^*)/(x_{k-1}-x^*)|)}$$

$$\mathbf{r} \approx \frac{\log(|e_{k+1}/e_k|)}{\log(|e_k/e_{k-1}|)} = \frac{\log(|(x_{k+1} - x^*)/(x_k - x^*)|)}{\log(|(x_k - x^*)/(x_{k-1} - x^*)|)}$$

If we know the true root, we can estimate the convergence rate…

Can you think of a use for this?

Without proof

The following approximation holds:

$$\mathbf{r} \approx \frac{\log(|(x_{k+1} - x_k)/(x_k - x_{k-1})|)}{\log(|(x_k - x_{k-1})/(x_{k-1} - x_{k-2})|)}$$

This does not require knowledge of the true root

Senning, Jonathan R. "Computing and Estimating the Rate of Convergence"

# Newton method

## Advantages

- Faster than bisection.
- One initial guess…

## Disadvantages

- Relies on **being able to compute the derivative** (symbolically or otherwise)
- Division by zero (when $f'(x)=0$)
- Root jumping

# scipy.optimize.newton

**scipy.optimize.newton(***func, x0, fprime=None, args=(), tol=1.48e-08, maxiter=50, fprime2=None***)** [source]

Find a zero using the Newton-Raphson or secant method.

Find a zero of the function *func* given a nearby starting point *x0*. The Newton-Raphson method is used if the derivative *fprime* of *func* is provided, otherwise the secant method is used. If the second order derivative *fprime2* of *func* is provided, then Halley's method is used.

**Parameters:**

**func : *function***

The function whose zero is wanted. It must be a function of a single variable of the form f(x,a,b,c...), where a,b,c... are extra arguments that can be passed in the *args* parameter.

**x0 : *float***

An initial estimate of the zero that should be somewhere near the actual zero.

**fprime : *function, optional***

The derivative of the function when available and convenient. If it is None (default), then the secant method is used.

**args : *tuple, optional***

Extra arguments to be used in the function call.

**tol : *float, optional***

The allowable error of the zero value.

**maxiter : *int, optional***

Maximum number of iterations.

**fprime2 : *function, optional***

The second order derivative of the function when available and convenient. If it is None (default), then the normal Newton-Raphson or the secant method is used. If it is not None, then Halley's method is used.


Previous topic

scipy.optimize.bisect

Next topic

scipy.optimize.fixed_point


https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.newton.html#scipy.optimize.newton

# What if I don't know the derivative?

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

finite difference approx

$$f'(x_i) \approx \frac{f(x_i) - f(x_{i-1})}{x_i - x_{i-1}}$$

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

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

# Secant method — algorithm

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

1. Set $i = 1$ and choose an initial values $x_0, x_1$
2. Find $x_{i+1} = x_i - \frac{f(x_i)(x_i - x_{i-1})}{f(x_i) - f(x_{i-1})}$
3. Find $|\epsilon_a| = |\frac{x_{i+1} - x_i}{x_i}| * 100$
4. If $|\epsilon_a| <$ `tol` **Stop.**

Else, repeat from step 2 with $i = i + 1$ and $x_i = x_{i+1}$ , $x_{i-1} = x_i$

# Homework

- Derive a graphic interpretation of the secant method.
- **Hint:** It's called secant for a reason.

# Rate of convergence

**bisection**

[L, U]

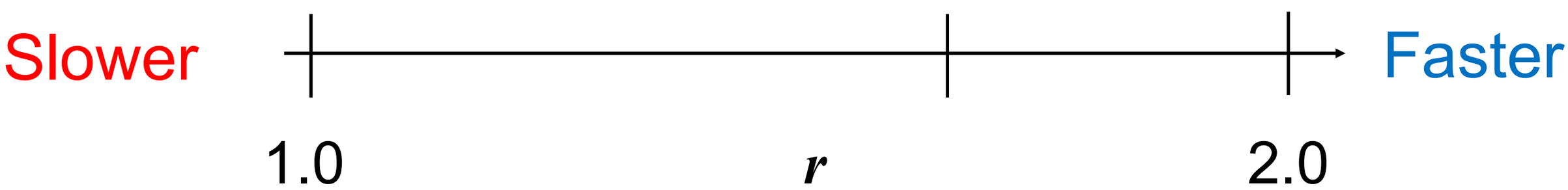
$\phi \approx 1.6$

**Secant**

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

**Newton**

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




https://nptel.ac.in/courses/122104019/numerical-analysis/Rathish-kumar/ratish-1/f3node5.html

# Secant method

## Advantages

- Superlinear convergence
- Only one new function evaluation per iteration
- Low cost per iteration, when compared to Newton.

## Disadvantages

- Two guesses

# scipy.optimize.brentq

**scipy.optimize.brentq(***f, a, b, args=(), xtol=2e-12, rtol=8.881784197001252e-16, maxiter=100, full_output=False, disp=True***)** [source]

Find a root of a function in a bracketing interval using Brent's method.

Uses the classic Brent's method to find a zero of the function *f* on the sign changing interval [a , b]. Generally considered the best of the rootfinding routines here. It is a safe version of the secant method that uses inverse quadratic extrapolation. Brent's method combines root bracketing, interval bisection, and inverse quadratic interpolation. It is sometimes known as the van Wijngaarden-Dekker-Brent method. Brent (1973) claims convergence is guaranteed for functions computable within [a,b].

[Brent1973] provides the classic description of the algorithm. Another description can be found in a recent edition of Numerical Recipes, including [PressEtal1992]. Another description is at http://mathworld.wolfram.com/BrentsMethod.html. It should be easy to understand the algorithm just by reading our code. Our code diverges a bit from standard presentations: we choose a different formula for the extrapolation step.

https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.brentq.html#scipy.optimize.brentq

# What about systems of non-linear equations?

What values of $x_1$ and $x_2$ satisfy the equations

$\underbrace{x_1 + 2x_2 = 2}_{\text{Linear}}$ and $\underbrace{x_1^2 + 4x_2^2 = 4}_{\text{Non-Linear}}$ ?

- Harder to solve than single equations
- Computational complexity scales rapidly with dimension of problem
- Many of the 1D methods for solving nonlinear systems scale to higher dimensions

First, some multivariable calculus..

# Partial Derivatives

$$f(x, y)$$

- We use $\frac{\partial f}{\partial x}$ to denote the partial derivative of $f$ with respect to $x$

- Describes how the gradient of $f$ changes in the $x$ direction for a given $y$ value

- Found by treating all other variables as constant values and differentiating as if it was a single variable function

# Partial Derivatives

Example.

$$f(x, y) = 3x + 2y^2 + x\sin(y)$$

$$\frac{\partial f}{\partial x} = ?$$

$$\frac{\partial f}{\partial y} = ?$$

# Partial Derivatives

Example.

$$f(x, y) = 3x + 2y^2 + x\sin(y)$$

$$\frac{\partial f}{\partial x} = 3 + \sin(y)$$

$$\frac{\partial f}{\partial y} = ?$$

# Partial Derivatives

Example.

$$f(x, y) = 3x + 2y^2 + x\sin(y)$$

$$\frac{\partial f}{\partial x} = 3 + \sin(y)$$

$$\frac{\partial f}{\partial y} = 4y + x\cos(y)$$

# The Jacobian

Say we have a vector containing $m$ functions in $n$ variables:

$$\boldsymbol{f}(\boldsymbol{x}) = \begin{bmatrix} f_1(x_1, x_2, \dots, x_n) \\ \vdots \\ f_m(x_1, x_2, \dots, x_n) \end{bmatrix}$$

The Jacobian of this vector is the matrix that contains all partial derivatives of all functions:

$$J_{\boldsymbol{f}}(\boldsymbol{x}) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

- Each row represents a function
- Each column represents a variable

# The Jacobian

Example.

$$\boldsymbol{f}(\boldsymbol{x}) = \begin{bmatrix} f_1(x_1, x_2) \\ f_2(x_1, x_2) \end{bmatrix} = \begin{bmatrix} x_1 + 2x_2 \\ e^{x_1} + 4x_2^2 \end{bmatrix}$$

$$\frac{\partial f_1}{\partial x_1} = 1 \qquad \frac{\partial f_1}{\partial x_2} = 2$$

$$\frac{\partial f_1}{\partial x_1} = e^{x_1} \qquad \frac{\partial f_1}{\partial x_2} = 8x_2$$

$$J_f(x) \equiv \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} \end{bmatrix} = \begin{bmatrix} 1 & 2 \\ e^{x_1} & 8x_2 \end{bmatrix}$$

# Newton's Method for Non-Linear System

We want to find $\boldsymbol{x}^*$ such that $\boldsymbol{f}(\boldsymbol{x}^*) = \boldsymbol{0}$

Recall, linear approximation in one dimension:

$$f(x + h) \approx f(x) + f'(x)h$$

Extending to multiple dimensions:

$$\boldsymbol{f}(\boldsymbol{x} + \boldsymbol{h}) \approx \boldsymbol{f}(\boldsymbol{x}) + J_{\boldsymbol{f}}(\boldsymbol{x})\boldsymbol{h}$$

**Note:**

- $\boldsymbol{f}$ is a vector of functions
- $\boldsymbol{x}$ and $\boldsymbol{h}$ are vectors of values

At iteration $k$

$$\boldsymbol{f(x_k + h) \approx f(x_k) + J_f\,(x_k)h}$$

$$\boxed{\boldsymbol{x^* = x_k + h}}$$

$$\boldsymbol{f(x^*) \approx f(x_k) + J_f\,(x_k)h}$$

$$\boldsymbol{0 \approx f(x_k) + J_f\,(x_k)h}$$

$$\boldsymbol{J_f\,(x_k)h \approx -f(x_k)}$$

This is a linear system with unknown vector $\boldsymbol{h}$

Solve for $\boldsymbol{h}$ then $\boldsymbol{x^* \approx x_k + h}$

⟵ Solve $\boldsymbol{J_f\,(x_i)h_i = -f(x_i)}$ for $\boldsymbol{h_i}$ then $\boldsymbol{x_{i+1} = x_i + h_i}$

# Newton's method Multiple Dimensions — algorithm

Solve $J_f\,(\boldsymbol{x}_i)\boldsymbol{h}_i = -\boldsymbol{f}(\boldsymbol{x}_i)$ for $\boldsymbol{h}_i$ then $\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + \boldsymbol{h}_i$

1. Set $i = 0$ and choose an initial value $\boldsymbol{x}_0$

2. Solve $J_f\,(\boldsymbol{x}_i)\boldsymbol{h}_i = -\boldsymbol{f}(\boldsymbol{x}_i)$ for $\boldsymbol{h}_i$ then $\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + \boldsymbol{h}_i$

3. Find $|\epsilon_a| = \dfrac{|\boldsymbol{x}_{i+1} - \boldsymbol{x}_i|_2^2}{|\boldsymbol{x}_i|_2^2} * 100 = \dfrac{|\boldsymbol{h}_i|_2^2}{|\boldsymbol{x}_i|_2^2} * 100$

4. If $|\epsilon_a| <$ `tol` **Stop.**

Else, repeat from step 2 with $i = i + 1$ and $\boldsymbol{x}_i = \boldsymbol{x}_{i+1}$

We'll do the implementations in the labs!

**Next time**: use the foundations we have learned to model and simulate things…