# Numerical Techniques

Amal Medhi

School of Physics
IISER Thiruvananthapuram

# Floating point numbers

- In computers, numbers are represented in binary format.
- The *floating point numbers* in computer are used to represent the real number system in mathematics.
- But since computer allocates only a *finite number of bits* to represent a number, it introduces an inherent approaximation.
- As a result, the floating point number system has certain peculiarities not present in real number system.
- The rounding error that is inevitable in representing a real number as a floating point number is generally small. *But successive operations on such numbers can magnify microscopic error to macroscopic size*.
- Here we will discuss some of the subleties associated with this approaximate representation of real numbers in computers.

# IEEE format for floating point numbers

- Floating point representation of real numbers is similar to the *scientific notation* of real numbers.
- In the past, several formats were used for representation of floating point numbers in computers.
- The IEEE format is now a standard adopted by all computers today.

# IEEE format for floating point numbers

- Floating point representation of real numbers is similar to the *scientific notation* of real numbers.
- In the past, several formats were used for representation of floating point numbers in computers.
- The IEEE format is now a standard adopted by all computers today.
- In the standard, format of a ***normalized*** floating point number is

$$\pm 1.b_1 b_2 b_3 \ldots \times 2^E$$

- The standard define three levels of precision for floating point numbers: ***single precision, double precision, and extended double precision***.
- Number of bits allocated for these three levels are 32 bits, 64 bits, and 80 bits respectivley.

# IEEE format for floating point numbers

- The bits are divided among parts as follows:

| precision | sign | exponent | mantissa |
|---|---|---|---|
| single | 1 | 8 | 23 |
| double | 1 | 11 | 52 |
| long double | 1 | 15 | 64 |

# IEEE format for floating point numbers

- The bits are divided among parts as follows:

| precision | sign | exponent | mantissa |
|---|---|---|---|
| single | 1 | 8 | 23 |
| double | 1 | 11 | 52 |
| long double | 1 | 15 | 64 |

## The machine epsilon $\epsilon_M$

- Consider the ***single precision*** numbers. Here number of bits for the exponent & the mantissa are $M = 8$ and $N = 23$, respectively.
- The **normalized** representation of the number $1.0_{10}$ is

$$+1.\boxed{000\ 000\ 000\ 000\ 000\ 000\ 000\ 00} \times 2^0$$

- The next floating point number **greater than** $1.0_{10}$ is

$$+1.\boxed{000\ 000\ 000\ 000\ 000\ 000\ 000\ 01} \times 2^0$$

which is $1.0 + 2^{-23}$.

# IEEE format for floating point numbers

## The machine epsilon $\epsilon_M$

- The *machine epsilon* $\epsilon_M$ is defined as the distance between 1.0 and the smallest floating point number greater than 1.0.
- For *single precision* ($N = 23$) numbers: $\epsilon_M = 2^{-23}$.
- For *double precision* ($N = 52$) numbers: $\epsilon_M = 2^{-52}$.
- For *long double precision* ($N = 64$) numbers: $\epsilon_M = 2^{-64}$.

# IEEE format for floating point numbers

## The machine epsilon $\epsilon_M$

- The *machine epsilon* $\epsilon_M$ is defined as the distance between 1.0 and the smallest floating point number greater than 1.0.
- For *single precision* ($N = 23$) numbers: $\epsilon_M = 2^{-23}$.
- For *double precision* ($N = 52$) numbers: $\epsilon_M = 2^{-52}$.
- For *long double precision* ($N = 64$) numbers: $\epsilon_M = 2^{-64}$.

## Rounding

- Consider the binary representation of number 9.4.
  $9.4_{10} = 1001.0110011001100110\ldots\ldots$
  In **normalized** form,

$$9.0_{10} = +1.\boxed{001\;0110\;0110\;0110\;0110\;0110}\;0110\,0110\ldots \times 2^3$$

- In *single precision*, we have space for only 23 bits (inside the box) for the mantissa. The integral part, '1' is not stored as it is understood that it is there.

# Rounding error

- In such cases as above, we have to either discard all the bits from 24th onwards or do some rounding.
- The IEEE standard is to round the number *to its nearest vlaue* as follows.
  - Add 1 to the bit-23 if the bit-24 is 1 (round up), do nothing (simply truncate) if bit-24 is 0 (round down).
- Applying the *round-to-the-nearest* rule, the number $x = 9.4$ in single precision would be represented by the number $fl(x)$ given by

$$fl(9.4) = +1.\boxed{001\ 0110\ 0110\ 0110\ 0110\ 0110} \times 2^3$$

- The rounding error involved in this case is,

$$fl(9.4) - 9.4 = -0.\overline{0110} \times 2^{-23} \times 2^3 = -0.4 \times 2^{-20}$$

- The *absolute rounding error* is therefore,

$$|fl(9.4) - 9.4| = 0.4 \times 2^{-20}$$

- Thus we see that floating point number representation of a real number $x$ may not be exactly $x$ though very close.

# IEEE format for floating point numbers

- A useful quantity to define is the *relative rounding error* given by

$$\delta = \frac{|fl(x) - x|}{|x|}, \quad \text{if } x \neq 0$$

- It turns out, the relative rounding error $\delta \leq \frac{1}{2}\epsilon_M$.

# IEEE format for floating point numbers

- A useful quantity to define is the *relative rounding error* given by

$$\delta = \frac{|fl(x) - x|}{|x|}, \quad \text{if } x \neq 0$$

- It turns out, the relative rounding error $\delta \leq \frac{1}{2}\epsilon_M$.

## Machine representation details

- Let us examine the actual machine representation of a normalized floating point number.

$$\pm 1.b_1 b_2 b_3 \ldots \times 2^E$$

- The bit field of a single precision number has the form

$$\boxed{s \,|\, a_1 a_2 \ldots a_8 \,|\, b_1 b_2 \ldots b_{23}}$$

- The sign bit $s$ is 0 for +vs numbers and 1 for −ve numbers.

# Machine representation

- Next 8 bits are used to represent the exponent value $E$.
  - This field does not actually store $E$ but store $(b+E)$ where $b$ is called the *exponent bias*. The bias for single precision numbers is $b = 2^{10} - 1 = 127$.
  - For normalized numbers, $E$ can have values from -126 to +127. These are 254 values. The rest 2 possible values are reserved for special numbers.
  - Thus for the normalized numbers, possible values of $(b+E)$ are 1 to 254. The special values will have $(b+E) = 0$ and 255.
  - The special value 255 for $(b+E)$ is used to represent $\infty$ if the mantissa bit string are all zeros and *NaN* (not a number) otherwise.
  - The special value 0 for $(b+E)$ is used to represent the number 0 (***the most important number***) if the mantissa bit string are all zeros. But in this case $E$ is intepreted as $-126$ not $-127$. More on this next.

# Machine representation

- Next 8 bits are used to represent the exponent value $E$.
  - This field does not actually store $E$ but store $(b+E)$ where $b$ is called the *exponent bias*. The bias for single precision numbers is $b = 2^{10} - 1 = 127$.
  - For normalized numbers, $E$ can have values from -126 to +127. These are 254 values. The rest 2 possible values are reserved for special numbers.
  - Thus for the normalized numbers, possible values of $(b+E)$ are 1 to 254. The special values will have $(b+E) = 0$ and 255.
  - The special value 255 for $(b+E)$ is used to represent $\infty$ if the mantissa bit string are all zeros and *NaN* (not a number) otherwise.
  - The special value 0 for $(b+E)$ is used to represent the number 0 (***the most important number***) if the mantissa bit string are all zeros. But in this case $E$ is intepreted as $-126$ not $-127$. More on this next.

## Subnormal numbers

- For the special exponent $(b+E) = 0$, with $E$ interpreted as $-126$ (instead of $-127$), numbers have the following form called ***subnormal*** numbers,

$$\pm 0.b_1 b_2 \ldots b_2 3 \times 2^{-126}$$

# Machine representation

- The reason $E$ is intepreted this way is because it allows to represent numbers closer (on lower side) to the lowest +ve normalized numbers.
- This scheme of machine representation of single precision numbers is illustrated in the following table.
- Thus the **smallest normalized single precision** +ve number is $1.00000000000000000000000 \times 2^{-126}$ which is $\approx 1.2 \times 10^{-38}$. For double precision, it is $\approx 2.2 \times 10^{-308}$.
- Thus the **smallest subnormal single precision** +ve number is $0.00000000000000000000001 \times 2^{-126}$ which is $\approx 5.9 \times 10^{-39}$. For double precision, it is $\approx 1.1 \times 10^{-308}$.
- Note that this smallest number is different from machine precision $\epsilon_M$. Numbers much smaller that $\epsilon_M$ can be represented in a computer, even though adding them to 1 may have no effect.

# Machine representation

- This scheme of single precision machine represntation of floating point numbers is illustrated in the following table.

| If exponent bitstring $a_1 \dots a_8$ is | Then numerical value represented is |
|---|---|
| $(00000000)_2 = (0)_{10}$ | $\pm(0.b_1b_2b_3\dots b_{23})_2 \times 2^{-126}$ |
| $(00000001)_2 = (1)_{10}$ | $\pm(1.b_1b_2b_3\dots b_{23})_2 \times 2^{-126}$ |
| $(00000010)_2 = (2)_{10}$ | $\pm(1.b_1b_2b_3\dots b_{23})_2 \times 2^{-125}$ |
| $(00000011)_2 = (3)_{10}$ | $\pm(1.b_1b_2b_3\dots b_{23})_2 \times 2^{-124}$ |
| $\downarrow$ | $\downarrow$ |
| $(01111111)_2 = (127)_{10}$ | $\pm(1.b_1b_2b_3\dots b_{23})_2 \times 2^{0}$ |
| $(10000000)_2 = (128)_{10}$ | $\pm(1.b_1b_2b_3\dots b_{23})_2 \times 2^{1}$ |
| $\downarrow$ | $\downarrow$ |
| $(11111100)_2 = (252)_{10}$ | $\pm(1.b_1b_2b_3\dots b_{23})_2 \times 2^{125}$ |
| $(11111101)_2 = (253)_{10}$ | $\pm(1.b_1b_2b_3\dots b_{23})_2 \times 2^{126}$ |
| $(11111110)_2 = (254)_{10}$ | $\pm(1.b_1b_2b_3\dots b_{23})_2 \times 2^{127}$ |
| $(11111111)_2 = (255)_{10}$ | $\pm\infty$ if $b_1 = \dots = b_{23} = 0$, NaN otherwise |

# Root finding

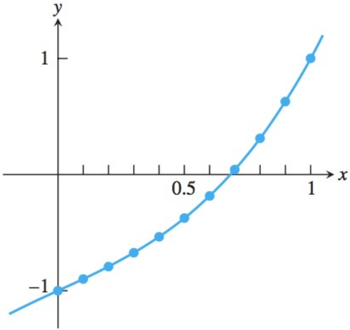
- **For a function $f(x)$, *find* $r$ for which $f(r) = 0$.**

# Root finding

- **For a function $f(x)$, *find* $r$ for which $f(r) = 0$.**
- Does a root exist?
- **Theorem:** Let $f(x)$ be a *continous* function on $[a, b]$, satisfying $f(a)f(b) < 0$. Then there exist a number $r \in (a, b)$ such that $f(r) = 0$.

# Root finding

- **For a function $f(x)$, *find* $r$ for which $f(r) = 0$.**
- Does a root exist?
- **Theorem:** Let $f(x)$ be a *continous* function on $[a, b]$, satisfying $f(a)f(b) < 0$. Then there exist a number $r \in (a, b)$ such that $f(r) = 0$.
- Bracketing a root.

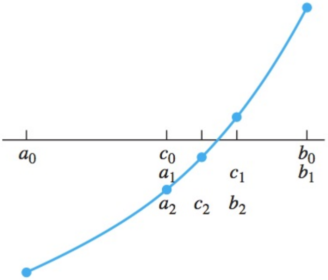



- Here root is bracketed in $[-1, 1]$ as $f(1)f(-1) < 0$.

# Bisection method

# Bisection method

- First find an initial interval $[a, b]$ which bracket the root.
- Pick the midpoint $c = \frac{a+b}{2}$. That is *bisect the interval*.
  - If $f(c) = 0 \rightarrow$ stop.
  - If $f(a)f(c) \rightarrow$ new bracket $[a, c]$.
  - If $f(c)f(b) \rightarrow$ new bracket $[c, b]$.




- Bisection Method

# Bisection method

- **Accuracy:** A solution is **correct with p decimal places** if the error is less than $0.5 \times 10^{-p}$.
- For the bisection method, after $n$ iterations:
  - The interval $[a_n, b_n]$ has the length $(b-a)/2^n$.
  - Best estimate for the solution $r$ is $x_c = (a_n + b_n)/2$.
  - Solution error = $|x_c - r| = \frac{b-a}{2^{n+1}}$.
  - Function evaluations = $n + 2$.

# Bisection method

- **Accuracy:** A solution is **correct with p decimal places** if the error is less than $0.5 \times 10^{-p}$.
- For the bisection method, after $n$ iterations:
  - The interval $[a_n, b_n]$ has the length $(b-a)/2^n$.
  - Best estimate for the solution $r$ is $x_c = (a_n + b_n)/2$.
  - Solution error = $|x_c - r| = \frac{b-a}{2^{n+1}}$.
  - Function evaluations = $n + 2$.
- **Question:** Find the root of $f(x) = (\cos x - x)$ in $[0, 1]$ to within **six** correct decimal places.
  - Error after $n$ steps is $(b-a)/2^{n+1} = 1/2^{n+1}$.
  - We require $1/2^{n+1} < 0.5 \times 10^{-6}$. This means $n > \frac{6}{\log_{10} 2} \approx 19.9$.
  - We need $n = 20$ iterations to achieve the accuracy.

# Forward error and backward error

- Consider finding the root of the following equation:

$$f(x) = x^3 - 2x^2 + \frac{4}{3}x - \frac{8}{27} = 0$$

- The analytical answer is $r = 2/3 = 0.66666666\ldots$.
- Suppose numerically we want to find $r$ to **six** significant digits.

# Forward error and backward error

- Consider finding the root of the following equation:

$$f(x) = x^3 - 2x^2 + \frac{4}{3}x - \frac{8}{27} = 0$$

- The analytical answer is $r = 2/3 = 0.66666666\ldots$.
- Suppose numerically we want to find $r$ to **six** significant digits.
- A typical bisection calculation would stop iterating and declare the root to be $r = 0.6666641$ becuase numerically $f(0.6666641) = 0$.
- So we never get the answer correct to six decimal place in this case.

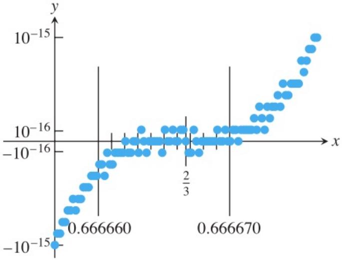
# Forward error and backward error

- Consider finding the root of the following equation:

$$f(x) = x^3 - 2x^2 + \frac{4}{3}x - \frac{8}{27} = 0$$

- The analytical answer is $r = 2/3 = 0.66666666\ldots$.
- Suppose numerically we want to find $r$ to **six** significant digits.
- A typical bisection calculation would stop iterating and declare the root to be $r = 0.6666641$ becuase numerically $f(0.6666641) = 0$.
- So we never get the answer correct to six decimal place in this case.
- The reason is the nature of $f(x)$ near the root $r = 2/3$. There are many numbers near $r = 2/3$ where numerically $f(x) = 0$.
- Such things occur if the root is of *higher order.*

# Forward error and backward error



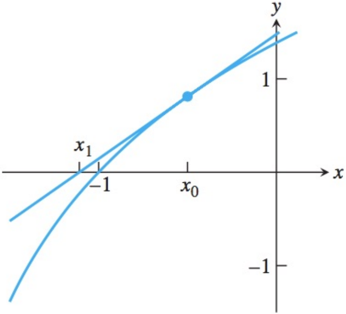

- Suppose $r$ is the correct root and $x_r$ is the numerical root.
- **Backward error** $= |f(x_r)|$ is the error in function value.
- **Forward error** $)x_r - r|$ is the error in the root value.
- A stopping criteria can be on either of these two.

# Newton-Raphson method

- A method that needs derivative information. Usually converges faster than bisection.

# Newton-Raphson method

- A method that needs derivative information. Usually converges faster than bisection.




- In Newton's method, to find the root of $f(x) = 0$, we start with guess solution $x_0$.
- Draw a tangent line to $f(x)$ at $x = x_0$. The tangent line will follow the $f(x)$ down to the axis towards the root.
- The intersection point of the line with the $x$-axis is the approximate root.
- The steps are repeated to get more closer to the answer.

# Newton-Raphson method

- The equation of the tangent line is $y - f(x_0) = f'(x_0)(x - x_0)$.
- The intersection point with $x$-axis is obtained by putting $y = 0$.
- The next guess for the root is

$$x \equiv x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

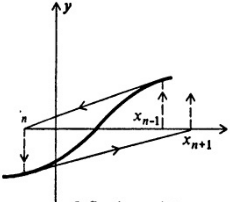
- One can repeat the steps and hope for convergence.

# Newton-Raphson method

- The equation of the tangent line is $y - f(x_0) = f'(x_0)(x - x_0)$.
- The intersection point with $x$-axis is obtained by putting $y = 0$.
- The next guess for the root is

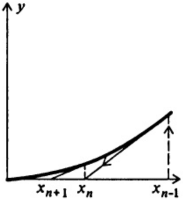
$$x \equiv x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

- One can repeat the steps and hope for convergence.
- Potential problems:



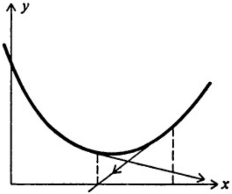

Inflection point

Can cycle and never converge




Multiple zero

Slow approach with $f' \to 0$ and trouble in division step




'Local minimum

Risks being sent very far away for next approximation

# Modified Newton method

- A better strategy for faster convergence is as follows.
- Lets $\Delta x_i = -f(x_i)/f'(x_i)$ be the step size in $i$-th iteration.
- Calculate the $f(x)$ at $x = x_i = x_i - \Delta x_i$.
- Check if $|f(x_{i+1})| < |f(x_i)|$. If yes accept, else halve the step.
- Keep halving the steps till $|f(x_{i+1})| < |f(x_{i+1})|$.

# Modified Newton method

- A better strategy for faster convergence is as follows.
- Lets $\Delta x_i = -f(x_i)/f'(x_i)$ be the step size in $i$-th iteration.
- Calculate the $f(x)$ at $x = x_i = x_i - \Delta x_i$.
- Check if $|f(x_{i+1})| < |f(x_i)|$. If yes accept, else halve the step.
- Keep halving the steps till $|f(x_{i+1})| < |f(x_{i+1})|$.
- DEMO: Modified Newton Method

# Modified Newton method

- A better strategy for faster convergence is as follows.
- Lets $\Delta x_i = -f(x_i)/f'(x_i)$ be the step size in $i$-th iteration.
- Calculate the $f(x)$ at $x = x_i = x_i - \Delta x_i$.
- Check if $|f(x_{i+1})| < |f(x_i)|$. If yes accept, else halve the step.
- Keep halving the steps till $|f(x_{i+1})| < |f(x_{i+1})|$.
- DEMO: Modified Newton Method
- **Convergence:** Let $e_i$ be the error after $i_{th}$ iteration. The iteration is quadratically convergent if

$$\lim_{i \to \infty} \frac{e_{i+1}}{e_i^2} \text{ is finite.}$$

- Let $r$ be a root of the function $f(x)$. Let we are at $x_i$ in $i$-th iteration.
- By Taylor expansion,

$$f(r) \approx f(x_i) + (r - x_i)f'(x_i) + \frac{(r - x_i)^2}{2} f''(c_i)$$

$c_i$ is between $x_i$ and $r$.

# Newton-Raphson method

- Since $f(r) = 0$,

$$-\frac{f(x_i)}{f'(x_i)} = r - x_i + \frac{(r - x_i)^2}{2}\frac{f''(c_i)}{f'(x_i)}$$

- Assuming $f'(x_i) \neq 0$,

$$x_i - \frac{f(x_i)}{f'(x_i)} - r = \frac{(r - x_i)^2}{2}\frac{f''(c_i)}{f'(x_i)}$$

$$x_{i+1} - r = e_i^2 \frac{f''(c_i)}{2f'(x_i)}$$

- Since $c_i \to r$ as $i \to \infty 0$,

$$\lim_{i \to \infty} \frac{e_{i+1}}{e_i^2} = \left| \frac{f''(r)}{2f'(r)} \right|$$

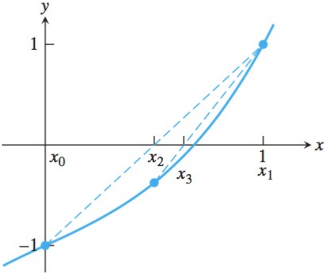
- Hence Newton's method is convergent quadratically if $f'(r) \neq 0$.

# Secant method

- A method without derivative.

# Secant method

- A method without derivative.




- If $x_{i-1}$ and $x_i$ are the last two guesses, it replaces the derivative by the approaximation

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

- The iteration step is:

$$x_{i+1} = x_i - \frac{f(x_i)(x_i - x_{i-1})}{f(x_i) - f(x_{i-1})}, \quad i = 1, 2, 3, \ldots$$

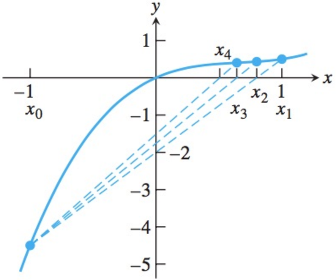
- Example: find the root $f(x) = x^3 + x - 1$ with starting guesses 0 and 1.

# The method of false position (*regula falsi*)

- Similar to bisection method, but the midpoint is repaced by a secant like approaximation.

# The method of false position (*regula falsi*)

- Similar to bisection method, but the midpoint is repaced by a secant like approaximation.

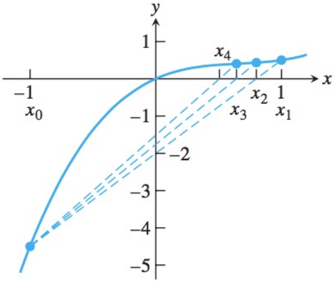



- Given an interval $[a, b]$ that brackets a root, the next point is

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

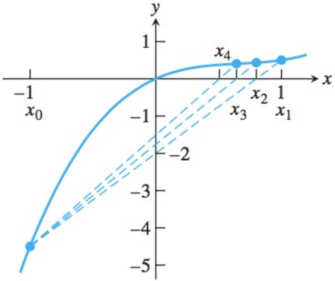
- The new point $c$ is guaranteed to be in $[a, b]$ since the points $(a, f(a))$ and $(b, f(b))$ lie on separate side of the $x$-axis.

# The method of false position (*regula falsi*)




- In the above, we approach the zero from one side. But this can be slow.
- We can improve by approacing from both sides.
- If the other side is at $x_0$, then we simply replace $f(x_0) \to f(x_0)/2$ in each iteration.

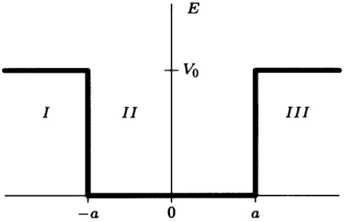
# The method of false position (*regula falsi*)




- In the above, we approach the zero from one side. But this can be slow.
- We can improve by approacing from both sides.
- If the other side is at $x_0$, then we simply replace $f(x_0) \to f(x_0)/2$ in each iteration.
- DEMO: Regula Falsi

# The Quantum Well problem

- Consider a quantum particle in a *finite 1D* potential well.




- The Schrodinger equation is

$$-\frac{\hbar^2}{2m}\frac{\partial^2\Psi}{\partial x^2} + V_0\Psi = E\Psi$$

- We can write it as

$$\frac{\partial^2\Psi}{\partial x^2} - \beta^2\Psi = 0, \quad \beta = \sqrt{2m(V_0 - E)/\hbar^2}$$

# The Quantum Well problem

- The general solutions are:

$$\Psi_I(x) = Ce^{\beta x}$$

$$\Psi_{II}(x) = A\sin\alpha x + B\cos\alpha x, \quad \alpha = \sqrt{2mE/\hbar^2}$$

$$\Psi_{III}(x) = De^{-\beta x}$$

# The Quantum Well problem

- The general solutions are:

$$\Psi_I(x) = Ce^{\beta x}$$
$$\Psi_{II}(x) = A\sin\alpha x + B\cos\alpha x, \quad \alpha = \sqrt{2mE/\hbar^2}$$
$$\Psi_{III}(x) = De^{-\beta x}$$

- Boundary conditions are: $\Psi(x)$ and $\Psi'(x)$ are continous at $x = \pm a$.
- BC at $x = -a$ gives

$$-A\sin\alpha a + B\cos\alpha a = Ce^{-\beta a}$$
$$A\alpha\cos\alpha a + B\alpha\sin\alpha a = \beta Ce^{-\beta a}$$

# The Quantum Well problem

- The general solutions are:

$$\Psi_I(x) = Ce^{\beta x}$$
$$\Psi_{II}(x) = A\sin\alpha x + B\cos\alpha x, \quad \alpha = \sqrt{2mE/\hbar^2}$$
$$\Psi_{III}(x) = De^{-\beta x}$$

- Boundary conditions are: $\Psi(x)$ and $\Psi'(x)$ are continous at $x = \pm a$.
- BC at $x = -a$ gives

$$-A\sin\alpha a + B\cos\alpha a = Ce^{-\beta a}$$
$$A\alpha\cos\alpha a + B\alpha\sin\alpha a = \beta Ce^{-\beta a}$$

- BC at $x = a$ gives

$$A\sin\alpha a + B\cos\alpha a = De^{\beta a}$$
$$A\alpha\cos\alpha a - B\alpha\sin\alpha a = -\beta De^{\beta a}$$

# The Quantum Well problem

- We get the following two equations:

$$
\begin{aligned}
2B\cos(\alpha a) =& (C+D)e^{-betaa} \\
2A\alpha\cos(\alpha a) =& \beta(C-D)e^{-betaa}
\end{aligned}
$$

# The Quantum Well problem

- We get the following two equations:

$$2B\cos(\alpha a) = (C + D)e^{-betaa}$$
$$2A\alpha\cos(\alpha a) = \beta(C - D)e^{-betaa}$$

- There are two classes of solutions:

$$A = 0, \quad B \neq 0, \quad C = D \Rightarrow \alpha\tan(\alpha a) = \beta, \quad \text{Even states}$$
$$A \neq 0, \quad B = 0, \quad C = -D \Rightarrow \alpha\cot(\alpha a) = -\beta, \quad \text{Odd states}$$

- So we need to solve these two transcendental equations to find the wave function and the energy eigen values.

# The Quantum Well problem

- Let us take the first equation.

$$f(E) = \alpha \tan(\alpha a) - \beta$$

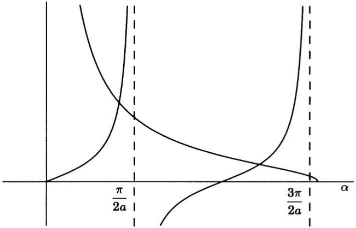
- We need to solve for $E$ such that $f(E) = 0$.

# The Quantum Well problem

- Let us take the first equation.

$$f(E) = \alpha \tan(\alpha a) - \beta$$

- We need to solve for $E$ such that $f(E) = 0$.
- For numerical solution, we need to make an initial guess. How?
- Fix the constants. Let's say: $V_0 = 10$ eV, $a = 3$Å, $m = m_e$.
- First, we expect $0 < E < V_0$.
- Next we can make plots of $\tan(\alpha a)$ and $\beta/\alpha$ versus $\alpha$.



# The Quantum Well problem

- We find that there is a solution between

$$0 < E < \frac{\pi^2 \hbar^2}{8ma^2} \approx 1.045eV \approx 1.045eV$$

# The Quantum Well problem

- We find that there is a solution between

$$0 < E < \frac{\pi^2 \hbar^2}{8ma^2} \approx 1.045eV \approx 1.045eV$$

- We can write the equation in the form,

$$f(E) = \beta \cos(\alpha a) - \alpha \sin(\alpha a) = 0$$

- Finally, we should use *natural units*: $m_e = 1$, $a = 3$Å and $\hbar^2 = 7.609097\ m_e(eV)$Å$^2$.

# The Quantum Well problem

- We find that there is a solution between

$$0 < E < \frac{\pi^2 \hbar^2}{8ma^2} \approx 1.045eV \approx 1.045eV$$

- We can write the equation in the form,

$$f(E) = \beta \cos(\alpha a) - \alpha \sin(\alpha a) = 0$$

- Finally, we should use *natural units*: $m_e = 1$, $a = 3\text{Å}$ and $\hbar^2 = 7.609097\, m_e(eV)\text{Å}^2$.

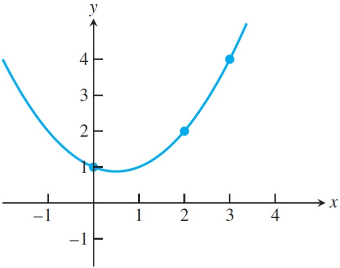
- DEMO: Quantum Well

# Interpolation

- A function $y = P(x)$ **interpolates** a set of data points $(x_1, y_1), \ldots, (y_n, y_n)$ if $P(x_i) = y_i$ for $i = 1, \ldots, n$, i.e. it passes through all the points.

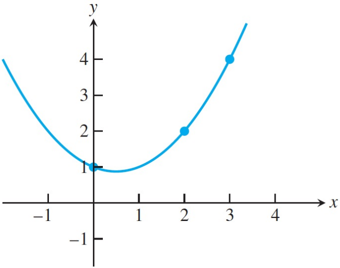
# Interpolation

- A function $y = P(x)$ **interpolates** a set of data points $(x_1, y_1), \dots, (y_n, y_n)$ if $P(x_i) = y_i$ for $i = 1, \dots, n$, i.e. it passes through all the points.
- Example: Parabola $P(x) = x^2/2 - x/2 + 1$ interpolates $(0, 1)$, $(2, 2)$ and $(3, 4)$.



# Interpolation

- A function $y = P(x)$ **interpolates** a set of data points $(x_1, y_1), \dots, (y_n, y_n)$ if $P(x_i) = y_i$ for $i = 1, \dots, n$, i.e. it passes through all the points.
- Example: Parabola $P(x) = x^2/2 - x/2 + 1$ interpolates $(0, 1)$, $(2, 2)$ and $(3, 4)$.




- Interplation can be viewd as a way of *data compression*.
- The numerical problem is - given a set of data points, find the interpolating function.

# Polynomial Interpolation

- Consider data points $(x_i, y_i)$, $i = 1, \ldots, n$. We would like to find an interpolating polynomial.

# Polynomial Interpolation

- Consider data points $(x_i, y_i)$, $i = 1, \ldots, n$. We would like to find an interpolating polynomial.
- **Lagrange interpolating formula**: Let $n = 3$. Construct the polynomial of degree $d = n - 3 = 2$ as follows:

$$P_2(x) = y_1 \frac{(x - x_2)(x - x_3)}{(x_1 - x_2)(x_1 - x_3)} + y_2 \frac{(x - x_1)(x - x_3)}{(x_2 - x_1)(x_2 - x_3)} + y_3 \frac{(x - x_1)(x - x_2)}{(x_3 - x_1)(x_3 - x_2)}$$

- It passes through the points $(x_1, y_1)$, $(x_2, y_2)$ and $(x_3, y_3)$, and called *Lagrange interpolating polynomial*.

# Polynomial Interpolation

- Consider data points $(x_i, y_i)$, $i = 1, \ldots, n$. We would like to find an interpolating polynomial.
- **Lagrange interpolating formula**: Let $n = 3$. Construct the polynomial of degree $d = n - 3 = 2$ as follows:

$$P_2(x) = y_1 \frac{(x - x_2)(x - x_3)}{(x_1 - x_2)(x_1 - x_3)} + y_2 \frac{(x - x_1)(x - x_3)}{(x_2 - x_1)(x_2 - x_3)} + y_3 \frac{(x - x_1)(x - x_2)}{(x_3 - x_1)(x_3 - x_2)}$$

- It passes through the points $(x_1, y_1)$, $(x_2, y_2)$ and $(x_3, y_3)$, and called *Lagrange interpolating polynomial*.
- For the general case of $n$ points, first define the degree $n - 1$ polynomial:

$$L_k(x) = \frac{(x - x_1) \ldots (x - x_{k-1})(x - x_{k+1}) \ldots (x - x_n)}{(x_k - x_1) \ldots (x_k - x_{k-1})(x_k - x_{k+1}) \ldots (x_k - x_n)}$$

- Then the Lagrange interpolating polynomial is:

$$P_{n-1}(x) = y_1 L_1(x) + \cdots + y_n L_n(x)$$

# Polynomial Interpolation

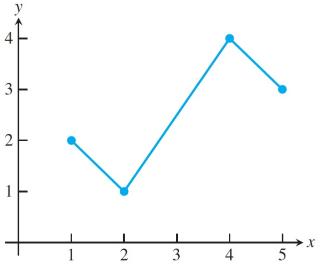
- Lagrange Interpolation Demo

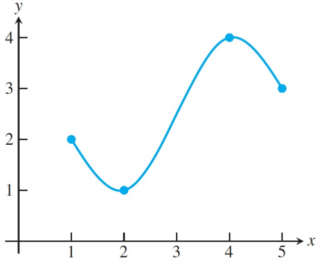
# Cubic Splines

- In interpolation, a single formula is used to meet all data points.
- In splines, we use several formulas, each a low degree polynomial, to pass through successive sets of data points.

# Cubic Splines

- In interpolation, a single formula is used to meet all data points.
- In splines, we use several formulas, each a low degree polynomial, to pass through successive sets of data points.
- Consider $n$ data points - $(x_i, y_i)$ with $x_1 < x_2 < \ldots < x_n$.
- A linear spline consists of $n - 1$ line segments that are drawn between neighbouring points.
- A cubic spline replaces linear functions between the data points by degree 3 polynomial.



# Cubic Splines

## Construction

# Cubic Splines

## Construction

- Consider $n$ data points - $(x_i, y_i)$ with $x_1 < x_2 < \ldots < x_n$.
- A **cubic spline** through the data points is a set of cubic polys

$$
\begin{aligned}
S_1(x) =& y_1 + b_1(x - x_1) + c_1(x - x_1)^2 + d_1(x - x_1)^3 \quad \text{on } [x_1, x_2] \\
S_2(x) =& y_2 + b_2(x - x_2) + c_2(x - x_2)^2 + d_2(x - x_2)^3 \quad \text{on } [x_2, x_3] \\
\vdots =& \quad \vdots \\
S_{n-1}(x) =& y_{n-1} + b_{n-1}(x - x_{n-1}) + c_{n-1}(x - x_{n-1})^2 + d_{n-1}(x - x_{n-1})^3
\end{aligned}
$$

on$[x_{n-1}, x_n]$

# Cubic Splines

- The polynomials have the following properties.
  1. $S_i(x_i) = y_i$ and $S_i(x_{i+1}) = y_{i+1}$ for $i = 1, 2, \ldots n-1$. Interpolates at the points.

# Cubic Splines

- The polynomials have the following properties.
  1. $S_i(x_i) = y_i$ and $S_i(x_{i+1}) = y_{i+1}$ for $i = 1, 2, \ldots n-1$. Interpolates at the points.
  2. $S'_{i-1}(x_i) = S'_i(x_i)$ for $i = 2, 3, \ldots n-1$. Slopes match at interior points - *smoothness*.

# Cubic Splines

- The polynomials have the following properties.
  1. $S_i(x_i) = y_i$ and $S_i(x_{i+1}) = y_{i+1}$ for $i = 1, 2, \ldots n-1$. Interpolates at the points.
  2. $S'_{i-1}(x_i) = S'_i(x_i)$ for $i = 2, 3, \ldots n-1$. Slopes match at interior points - *smoothness*.
  3. $S''_{i-1}(x_i) = S''_i(x_i)$ for $i = 2, 3, \ldots n-1$. Second derivatives match at interior points - *curvature matches*.

# Cubic Splines

- The polynomials have the following properties.
  1. $S_i(x_i) = y_i$ and $S_i(x_{i+1}) = y_{i+1}$ for $i = 1, 2, \ldots n-1$. Interpolates at the points.
  2. $S'_{i-1}(x_i) = S'_i(x_i)$ for $i = 2, 3, \ldots n-1$. Slopes match at interior points - *smoothness*.
  3. $S''_{i-1}(x_i) = S''_i(x_i)$ for $i = 2, 3, \ldots n-1$. Second derivatives match at interior points - *curvature matches*.
- There are a total of $3n-5$ equations.
- But $3n-3$ unknowns $a_i, b_i, c_i$.
- Set two more constraints - *natural splines*:

$$S''_1(x_1) = 0 \quad \text{and} \quad S''_{n-1}(x_n) = 0$$

# Cubic Splines

- Property-1 generates $n-1$ equations:

$$
\begin{aligned}
y_2 &= y_1 + b_1(x_2 - x_1) + c_1(x_2 - x_1)^2 + d_1(x_2 - x_1)^3 \\
&\vdots \\
y_n &= y_{n-1} + b_{n-1}(x_n - x_{n-1}) + c_{n-1}(x_n - x_{n-1})^2 + d_{n-1}(x_n - x_{n-1})^3
\end{aligned}
$$

# Cubic Splines

- Property-1 generates $n-1$ equations:

$$
\begin{aligned}
y_2 &= y_1 + b_1(x_2 - x_1) + c_1(x_2 - x_1)^2 + d_1(x_2 - x_1)^3 \\
&\vdots \\
y_n &= y_{n-1} + b_{n-1}(x_n - x_{n-1}) + c_{n-1}(x_n - x_{n-1})^2 + d_{n-1}(x_n - x_{n-1})^3
\end{aligned}
$$

- Property-2 generates $n-2$ equations:

$$
\begin{aligned}
0 =& S_1'(x_2) - S_2'(x_2) = b_1 + 2c_1(x_2 - x_1) + 3d_1(x_2 - x_1)^2 - b_2 \\
&\vdots \\
0 =& S_{n-2}'(x_{n-1}) - S_{n-1}'(x_{n-1}) = b_{n-2} + 2c_{n-2}(x_{n-1} - x_{n-2}) \\
&+ 3d_{n-2}(x_{n-1} - x_{n-2})^2 - b_{n-1}
\end{aligned}
$$

# Cubic Splines

- Property-3 generates $n-2$ equations:

$$
\begin{aligned}
0 =& S_1''(x_2) - S_2''(x_2) = 2c_1 + 6d_1(x_2 - x_1) - 2c_2 \\
&\vdots \\
0 =& S_{n-2}''(x_{n-1}) - S_{n-1}''(x_{n-1}) = 2c_{n-2} + 6d_{n-2}(x_{n-1} - x_{n-2})^2 - 2c_{n-1}
\end{aligned}
$$

# Cubic Splines

- Property-3 generates $n-2$ equations:

$$\begin{aligned}
0 =& S_1''(x_2) - S_2''(x_2) = 2c_1 + 6d_1(x_2 - x_1) - 2c_2 \\
&\vdots \\
0 =& S_{n-2}''(x_{n-1}) - S_{n-1}''(x_{n-1}) = 2c_{n-2} + 6d_{n-2}(x_{n-1} - x_{n-2})^2 - 2c_{n-1}
\end{aligned}$$

- Property-4 generates 2 equations:

$$\begin{aligned}
S_1''(x_1) =& 0 \Rightarrow 2c_1 = 0 \\
S_1''(x_{n-1}) =& 0 \Rightarrow 2c_n = 0
\end{aligned}$$

# Cubic Splines

- Property-3 generates $n-2$ equations:

$$
\begin{aligned}
0 =& S_1''(x_2) - S_2''(x_2) = 2c_1 + 6d_1(x_2 - x_1) - 2c_2 \\
&\vdots \\
0 =& S_{n-2}''(x_{n-1}) - S_{n-1}''(x_{n-1}) = 2c_{n-2} + 6d_{n-2}(x_{n-1} - x_{n-2})^2 - 2c_{n-1}
\end{aligned}
$$

- Property-4 generates 2 equations:

$$
\begin{aligned}
S_1''(x_1) =& 0 \Rightarrow 2c_1 = 0 \\
S_1''(x_{n-1}) =& 0 \Rightarrow 2c_n = 0
\end{aligned}
$$

- Define $\delta_i = x_{i+1} - x_i$ and $\Delta_i = y_{i+1} - y_i$.

# Cubic Splines

- Determine $c_i$-s from the following equation:

$$
\begin{bmatrix}
1 & 0 & 0 & & & \\
\delta_1 & 2\delta_1 + 2\delta_2 & \delta_2 & & & \\
0 & \delta_2 & 2\delta_2 + 2\delta_3 & \delta_3 & & \\
 & \ddots & \ddots & \ddots & \ddots & \\
 & & & \delta_{n-2} & 2\delta_{n-2} + 2\delta_{n-1} & \delta_{n-1} \\
 & & & 0 & 1 & 0
\end{bmatrix}
\begin{bmatrix}
c_1 \\ \\ \vdots \\ \\ c_n
\end{bmatrix}
=
\begin{bmatrix}
0 \\
3(\frac{\Delta_2}{\delta_2} - \\
\vdots \\
3\frac{\Delta_{n-1}}{\delta_{n-1}} - \\
0
\end{bmatrix}
$$

# Cubic Splines

- Determine $c_i$-s from the following equation:

$$\begin{bmatrix} 1 & 0 & 0 & & & \\ \delta_1 & 2\delta_1 + 2\delta_2 & \delta_2 & & & \\ 0 & \delta_2 & 2\delta_2 + 2\delta_3 & \delta_3 & & \\ & \ddots & \ddots & \ddots & \ddots & \\ & & & \delta_{n-2} & 2\delta_{n-2} + 2\delta_{n-1} & \delta_{n-1} \\ & & & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} c_1 \\ \\ \vdots \\ \\ c_n \end{bmatrix} = \begin{bmatrix} 0 \\ 3(\frac{\Delta_2}{\delta_2} - \\ \vdots \\ 3\frac{\Delta_{n-1}}{\delta_{n-1}} - \\ 0 \end{bmatrix}$$

- Determine $d_i$-s from the following equations:

$$d_i = \frac{c_{i+1} - c_i}{3\delta_i}, \quad i = 1, 2, \ldots n - 1$$

# Cubic Splines

- Determine $c_i$-s from the following equation:

$$
\begin{bmatrix}
1 & 0 & 0 & & & \\
\delta_1 & 2\delta_1 + 2\delta_2 & \delta_2 & & & \\
0 & \delta_2 & 2\delta_2 + 2\delta_3 & \delta_3 & & \\
 & \ddots & \ddots & \ddots & \ddots & \\
 & & & \delta_{n-2} & 2\delta_{n-2} + 2\delta_{n-1} & \delta_{n-1} \\
 & & & 0 & 1 & 0
\end{bmatrix}
\begin{bmatrix}
c_1 \\ \\ \vdots \\ \\ c_n
\end{bmatrix}
=
\begin{bmatrix}
0 \\
3(\frac{\Delta_2}{\delta_2} - \\
\vdots \\
3\frac{\Delta_{n-1}}{\delta_{n-1}} - \\
0
\end{bmatrix}
$$

- Determine $d_i$-s from the following equations:

$$d_i = \frac{c_{i+1} - c_i}{3\delta_i}, \quad i = 1, 2, \ldots n - 1$$

- Determine $b_i$-s from the following equations:

$$b_i = \frac{\Delta_i}{\delta_i} - \frac{\delta_i}{3}(2c_i + c_{i+1}), \quad i = 1, 2, \ldots n - 1$$

# Least Squares

- **Inconsistent systems of equations**: Consider the following system of linear equations

$$\begin{aligned} x_1 + x_2 &= 2 \\ x_1 - x_2 &= 1 \\ x_1 + x_2 &= 3 \end{aligned} \qquad \Rightarrow \begin{bmatrix} 1 & 1 \\ 1 & -1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \\ 3 \end{bmatrix} \qquad \Rightarrow A\mathbf{x} = \mathbf{b}$$

- It has no solution. In general $m$ equations of $n$ unknowns with $m > n$ has no solutions and is called inconsistent.

# Least Squares

- **Inconsistent systems of equations**: Consider the following system of linear equations

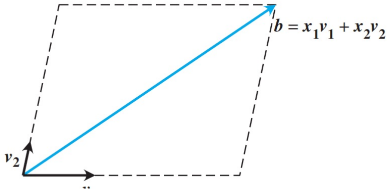
$$\begin{aligned} x_1 + x_2 &= 2 \\ x_1 - x_2 &= 1 \\ x_1 + x_2 &= 3 \end{aligned} \qquad \Rightarrow \begin{bmatrix} 1 & 1 \\ 1 & -1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \\ 3 \end{bmatrix} \qquad \Rightarrow A\mathbf{x} = \mathbf{b}$$

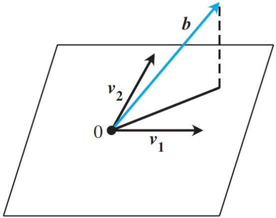
- It has no solution. In general $m$ equations of $n$ unknowns with $m > n$ has no solutions and is called inconsistent.
- Still we want to find an **approaximate** solution to it. How can we do it?
- Write the equations in the following form:

$$x_1 \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} + x_2 \begin{bmatrix} 1 \\ -1 \\ 1 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \\ 3 \end{bmatrix}$$
$$\Rightarrow x_1\mathbf{v}_1 + x_2\mathbf{v}_2 = \mathbf{b}$$

# Least Squares

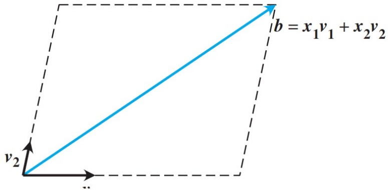
- We can interpret $\mathbf{v}_1$, $\mathbf{v}_2$ and $\mathbf{b}$ as three 3D vectors. We want to find $x_1$ and $x_2$ that satisfies the vector equation $x_1\mathbf{v}_1 + x_2\mathbf{v}_2 = \mathbf{b}$.
- But if $\mathbf{b}$ lies outside the plan containing $\mathbf{v}_1$, $\mathbf{v}_2$, then there is no solution.

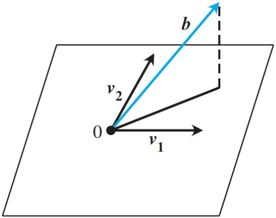


# Least Squares

- We can interpret $\mathbf{v}_1$, $\mathbf{v}_2$ and $\mathbf{b}$ as three 3D vectors. We want to find $x_1$ and $x_2$ that satisfies the vector equation $x_1\mathbf{v}_1 + x_2\mathbf{v}_2 = \mathbf{b}$.
- But if $\mathbf{b}$ lies outside the plan containing $\mathbf{v}_1$, $\mathbf{v}_2$, then there is no solution.




- Consider dropping a perpendicular from the tip of $\mathbf{b}$ on the plan containing $\mathbf{v}_1$, $\mathbf{v}_2$. Let the point be $\bar{x}_1\mathbf{v}_1 + \bar{x}_2\mathbf{v}_2 = A\bar{\mathbf{x}}$.
- The residual vector $\mathbf{b} - A\bar{\mathbf{x}}$ is $\perp$ to the plane.
- $\bar{\mathbf{x}}$ is the **best possible** solution to the inconsistent solutions.

# Least Squares

- Now since the vector $A\mathbf{x}$ is $\perp$ to $\mathbf{b} - A\bar{\mathbf{x}}$, their dot product vanishes,

$$\begin{aligned}
&(A\mathbf{x})^T(\mathbf{b} - A\bar{\mathbf{x}}) = 0 \quad \forall \mathbf{x} \\
\Rightarrow &\mathbf{x}^T A^T(\mathbf{b} - A\bar{\mathbf{x}}) = 0 \quad \forall \mathbf{x} \\
\Rightarrow &A^T(\mathbf{b} - A\bar{\mathbf{x}}) = 0 \\
\Rightarrow &A^T A\bar{\mathbf{x}} = A^T\mathbf{b}
\end{aligned}$$

# Least Squares

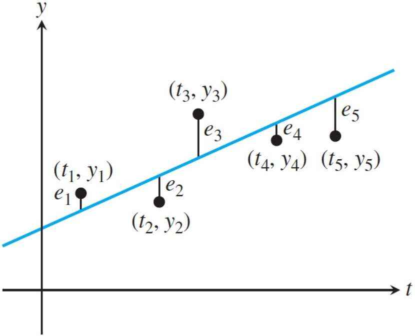
- Now since the vector $A\mathbf{x}$ is $\perp$ to $\mathbf{b} - A\bar{\mathbf{x}}$, their dot product vanishes,

$$\begin{aligned}
&(A\mathbf{x})^T(\mathbf{b} - A\bar{\mathbf{x}}) = 0 \quad \forall \mathbf{x} \\
\Rightarrow &\mathbf{x}^T A^T(\mathbf{b} - A\bar{\mathbf{x}}) = 0 \quad \forall \mathbf{x} \\
\Rightarrow &A^T(\mathbf{b} - A\bar{\mathbf{x}}) = 0 \\
\Rightarrow &A^T A\bar{\mathbf{x}} = A^T\mathbf{b}
\end{aligned}$$

- The last equation is called the **normal equation**. The solution $\bar{\mathbf{x}}$ is the least squares solutions of $Ax = b$.

# Least Squares

- **Data fitting problem**: Consider a set of data points $(t_i, y_i)$ where $i = 1, 2, \ldots, m$.
- We want to **fit** the data with a linear model, e.g. $y = c_1 + c_2 t$.
- The model need not pass through the points $(t_i, y_i)$. The error is defined as $e_i = y_i - (c_1 + c_2 t_i)$.




- We want to find $c_1, c_2$ such that the rms error $\sigma = \sqrt{\sum_i e_i^2 / m}$ is minimized.

# Least Squares

- This problem of minimizing rms error is equivalent to finding the least square solution to a normal equation.
- We first choose the model, such as $y = c_1 + c_2 t$.
- Next substitute the data points into the model. Each data point create an equation with unknowns $c_1$ and $c_2$. This results in a system $A\mathbf{x} = \mathbf{b}$.
- Solve the normal equation $A^T A x = A^T \mathbf{b}$.

# Least Squares

- This problem of minimizing rms error is equivalent to finding the least square solution to a normal equation.
- We first choose the model, such as $y = c_1 + c_2 t$.
- Next substitute the data points into the model. Each data point create an equation with unknowns $c_1$ and $c_2$. This results in a system $A\mathbf{x} = \mathbf{b}$.
- Solve the normal equation $A^T A x = A^T \mathbf{b}$.
- Least Squares Demo

Thank You.