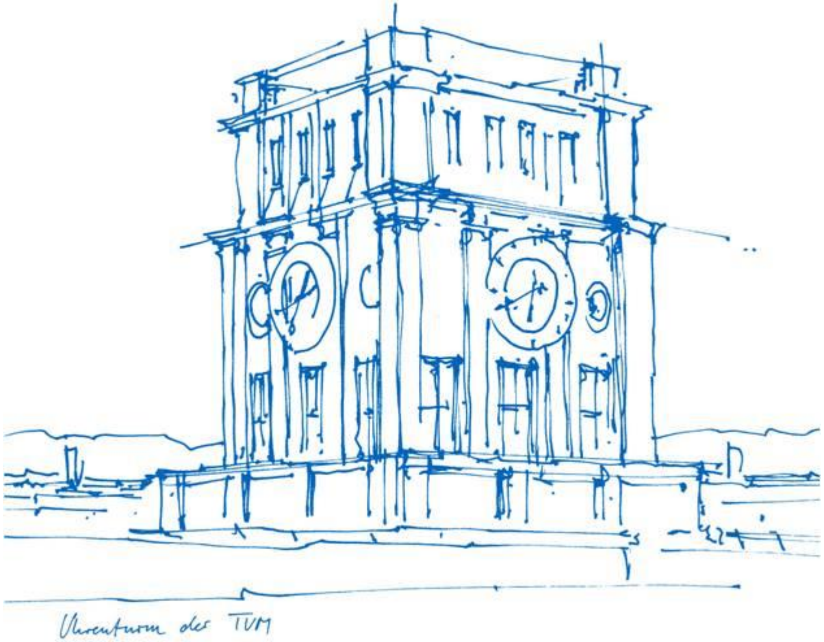
Rechnerarchitektur Praktikum

Daniel Petri Rocha, Dominik Fuchs, Robin Geißler

Technische Universität München

Garching, 26. Februar 2019



Numerische Quadratur

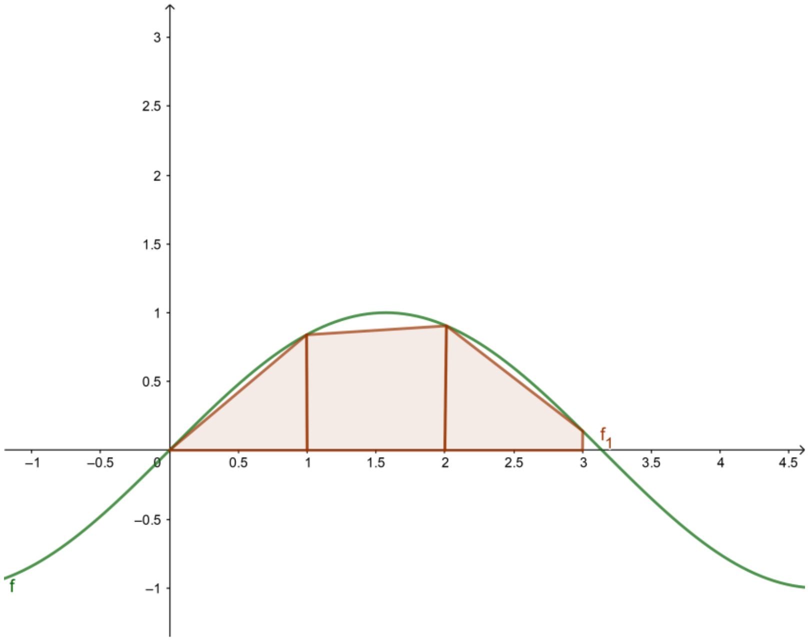
Daniel Petri Rocha, Dominik Fuchs, Robin Geißler

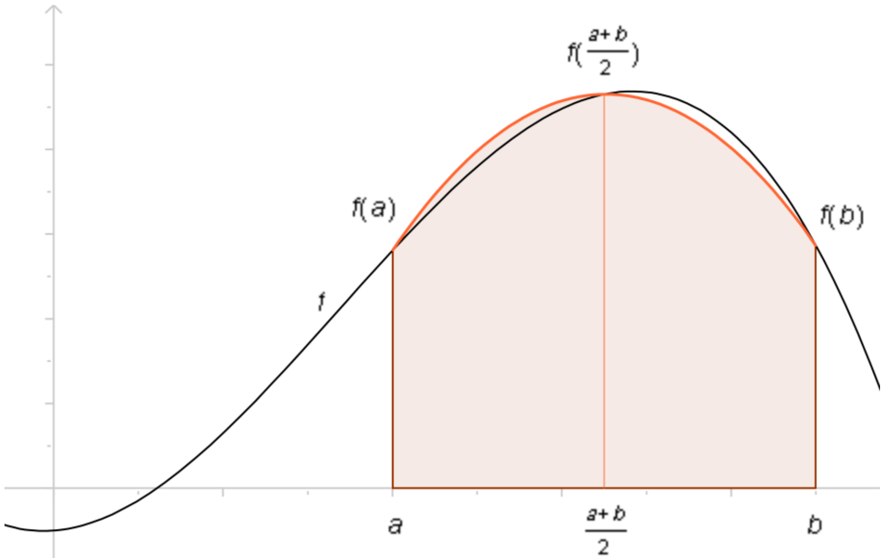
München, 26. Februar 2019

Gliederung

- Numerische Quadratur
- Problemstellung und Spezifikation
- Lösungsfindung
  - Konzeptioneller Teil – NewtonCotes Formeln
  - Praktischer Teil – Assembly Code
  - Praktischer Teil – Rahmenprogramm
- Tests: Güte der Näherung
- Laufzeitanalyse
- Security
- Fazit und Ausblick

Numerische Quadratur





https://de.wikipedia.org/wiki/Newton-Cotes-Formeln

- Zwei Parameter um die Annäherung genauer zu machen:
  - Anzahl der Stützstellen
  - Grad des approximierenden Polynoms

Problemstellung und Spezifikation

Numerische Quadratur

- Mithilfe der Newton-Cotes-Formel Annäherung an die Integration einer Funktion

- Konzeptioneller Teil
  - Abgeschlossene Newton-Cotes Formel zu Newton-Cotes-1 und Newton-Cotes-2
  - Einarbeitung mit Funktionspointer
  - Güte der Näherung, Grafische Darstellung
  - Security-Bedenken

- Praktischer Teil
  - Implementierung des Assembly für newton_cotes_1 und newton_cotes_2
  - Implementierung eines C-Rahmenprogramms

Lösungsfindung – Newton Cotes Formeln

Allgemeine Formel:

$$I_n[f] = \int_a^b p(x)\,\mathrm{d}x = (b-a) \cdot \sum_{i=0}^{n} \alpha_{in} \cdot p(x_i)$$

Lösungsfindung – Newton Cotes Formeln

Allgemeine Formel:

$$I_n[f] = \int_a^b p(x)\,\mathrm{d}x = (b-a) \cdot \sum_{i=0}^{n} \alpha_{in} \cdot p(x_i)$$

Stützstellen:

$$x_i = a + \frac{i \cdot (b-a)}{n}$$

Lösungsfindung – Newton Cotes Formeln

Allgemeine Formel:

$$I_n[f] = \int_a^b p(x)\,\mathrm{d}x = (b-a) \cdot \sum_{i=0}^{n} \alpha_{in} \cdot p(x_i)$$

Stützstellen:

$$x_i = a + \frac{i \cdot (b-a)}{n}$$

Gewicht der Stützstellen:

$$\alpha_{in} = \frac{1}{n} \int_0^n \prod_{\substack{j=0 \\ j \neq i}}^{n} \frac{x-j}{i-j}\,\mathrm{d}x$$

Ergebnis – Newton Cotes Formeln

Ergebnis für n = 1:

$$I_1[f] = (b - a) * \frac{p(a) + p(b)}{2}$$

Ergebnis – Newton Cotes Formeln

Ergebnis für n = 1:

$$I_1[f] = (b-a) * \frac{p(a) + p(b)}{2}$$

Ergebnis für n = 2:

$$I_2[f] = (b-a) * \frac{p(a) + 4 * p(\frac{a+b}{2}) + p(b)}{6}$$

Lösungsfindung – Assembly Implementierung

```
float *newton_cotes_1(float ( *p)(float), int a, int b)
```

```
.intel_syntax noprefix
.global newton_cotes_1

.data
result: .global float

.text

# Registeranfangsbelegung laut Calling Convention
#   rdi = Zeiger auf dynamisch gebundene Funktion
#   esi = integer a
#   edx = integer b
#   rax = Rueckgaberegister, enthaelt Zeiger auf Ergebnis im Speicher
```

Lösungsfindung – Assembly Implementierung

$$I_1[f] = (b-a) * \frac{p(a)+p(b)}{2}$$

Eingabevariablen:

rdi: Funktionspointer

rsi: int a

rdx: int b

```
# Sichere alle Eingabevariablen in callee-saved Register damit sie bei spaeteren Funktionsaufrufen nicht verloren gehen
push r12
push r13
push r14
mov r12, rdi
mov r13d, esi
mov r14d, edx

```

Lösungsfindung – Assembly Implementierung

$$I_1[f] = (b-a) * \frac{\boxed{p(a)} + p(b)}{2}$$

Aktuelle Belegung:

r12: Funktionspointer

r13d: int a

r14d: int b

```
    # Berechnung:
    cvtsi2sd xmm0, r13d      # xmm0 = double a        -convert und move
    call r12                 # xmm0 = double p(a)     -calc p(a)
    cvtsd2ss xmm1, xmm0      # xmm1 = float p(a)      -convert und move

```

Lösungsfindung – Assembly Implementierung

$$I_1[f] = \boxed{(b-a)} * \frac{\boxed{p(a)} + p(b)}{2}$$

Aktuelle Belegung:

r12: Funktionspointer

r13d: int a

r14d: int b

xmm1: float p(a)

```
    cvtsi2sd xmm0, r14d     # xmm0 = double b        -convert und move
    sub r14d, r13d          # r14 = int b - a        -calc b - a

```

Lösungsfindung – Assembly Implementierung

$$I_1[f] = \boxed{(b-a)} * \frac{\boxed{p(a)} + \boxed{p(b)}}{2}$$

Aktuelle Belegung:

r12: Funktionspointer

r13d: int a

r14d: int b - a

xmm1: float p(a)

```
movq r13, xmm1          # r13 = float p(a)       -move fuer Sicherung
call r12                # xmm0 = double p(b)     -calc p(b)
movq xmm1, r13          # xmm1 = double p(a)     -move fuer Wiederherstellung
cvtsd2ss xmm0,xmm0      # xmm0 = float p(b)      -convert
```

Lösungsfindung – Assembly Implementierung

$$I_1[f] = \boxed{(b-a)} * \frac{\boxed{p(a)} + \boxed{p(b)}}{\boxed{2}}$$

Aktuelle Belegung:

- r12: Funktionspointer
- r13d: float p(a)
- r14d: int b – a
- xmm0: float p(b)
- xmm1: float p(a)

```
mov r10d, 2             # r10 = int 2          -move
cvtsi2ss xmm2, r10d     # xmm2 = float 2.0     -convert und move
cvtsi2ss xmm3, r14d     # xmm3 = float b - a   -convert und move
```

Lösungsfindung – Assembly Implementierung

$$I_1[f] = \boxed{\boxed{(b-a)} * \frac{\boxed{p(a)} + \boxed{p(b)}}{\boxed{2}}}$$

Aktuelle Belegung:

r14d: int b – a

xmm0: float p(b)

xmm1: float p(a)

xmm2: float 2.0

xmm3: float b - a

```
    addss xmm0, xmm1        # xmm0 = float p(a) + p(b)                  -calc
    divss xmm0, xmm2        # xmm0 = float (p(a) + float p(b))/2.0      -calc
    mulss xmm0, xmm3        # xmm0 = float (b-a) * (p(a) + p(b))/2.0    -calc

```

Lösungsfindung – Assembly Implementierung

$$I_1[f] = \boxed{\boxed{(b-a)} * \frac{\boxed{p(a)} + \boxed{p(b)}}{\boxed{2}}}$$

Aktuelle Belegung:

- r14d: int b – a
- xmm0: float Ergebnis
- xmm1: float p(a)
- xmm2: float 2.0
- xmm3: float b - a

```
    # Ergebnis abspeichern:
    movss dword ptr ds:[result], xmm0 # *result = flat (b-a) * (p(a) + p(b))/2.0    -move in den Speicher
    mov rax, result                   # rax = result                                -move Speicheradresse

```

Lösungsfindung – Assembly Implementierung

$$I_1[f] = \boxed{\boxed{(b-a)} * \frac{\boxed{p(a)} + \boxed{p(b)}}{\boxed{2}}}$$

Aktuelle Belegung:

r14d: int b – a

xmm0: float Ergebnis

xmm1: float p(a)

xmm2: float 2.0

xmm3: float b – a

rax: float* result

```
    # Genutzte callee-save register wieder herstellen
    pop r14
    pop r13
    pop r12
ret
```

Rahmenprogramm main.c

- Deklaration der extern .S Assembly Dateien
- Prüft Eingaben auf Korrektheit mit strtol
- Bindet Funktionen Dynamisch mit <dlfcn.h> ein
  - ELF-Datei mit dlopen() laden
  - Adresse des Entry-Points der Funktion (0x2bb) mit dlsym() finden
  - Fehler mit dlerror() abgefangen
  - In der Makefile: Flag -ldl
- Zeitmessung
- Tests

Tests: Güte der Näherung

Tests werden mit Flag –*t* aktiviert

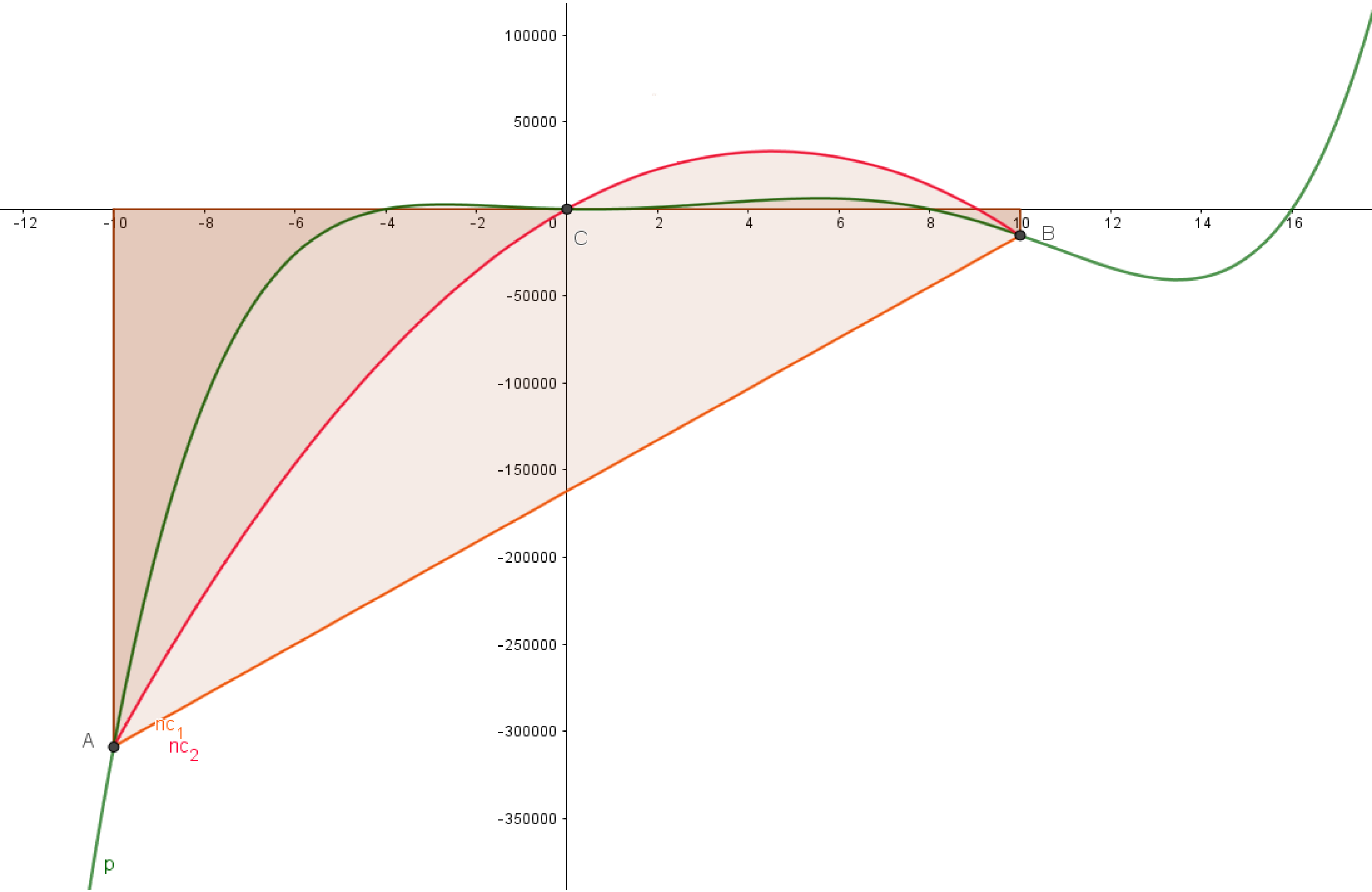
- Trapezformel (Newton Cotes 1) 100% genau für Polynome vom Grad 0 und 1
- Keplersche Fassregel (Newton Cotes 2) 100% genau für Polynome vom Grad 0, 1, 2 und 3 (einen Grad höher als erwartet!)

- Für Polynome vom Grad ≥ 4 ungenaue Ergebnisse
- Bsp. $x^5-21x^4+52x^3+480x^2-512x$ von -10 bis 10 Integrieren
- Newton Cotes 1: Fehler liegt bei 523%

Ausblick: Summierte Numerische Quadratur

- durch iterative Anwendung auf kleinere Intervalle Ungenauigkeit nur 2,38%
- Mit Newton Cotes 2 - 0,000673%

$x^5-21x^4+52x^3+480x^2-512x$



Laufzeitanalyse

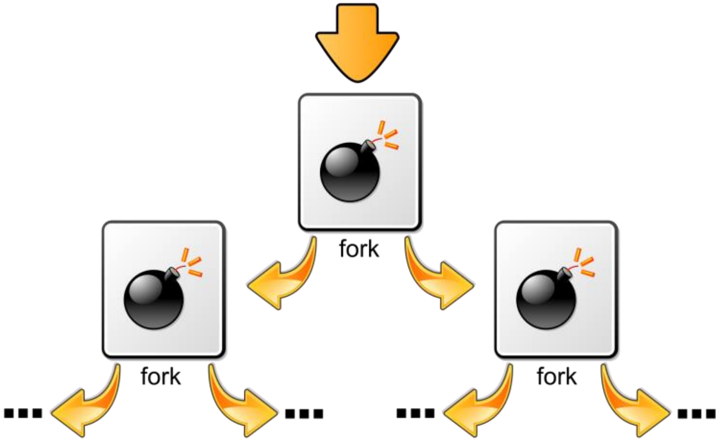
- Zeitmessungen mit CLOCK_MONOTONIC
  - Newton Cotes 1 und 2 werden 5 Millionen mal durchgeführt
  - Alle 1 Million Iterationen wird 1 Sekunde gewartet
  - Ergebnis auf der Rechnerhalle je nach aufgerufener Funktion:

NC1: 13 - 37 ns	NC2: 21 - 57 ns

- Zum Vergleich: in 60ns legt Licht 18m zurück!
- Laufzeit Summierte Numerische Quadratur: (b-a)-Fache von der durchschnittlichen Berechnungsdauer + Addition der Flächen

Security-Bedenken

- Funktionen enthalten Symbol „f“
- Rahmenprogramm arbeitet immer nur mit der Adresse dieser Labels
- Dynamische Bibliotheken werden eingebunden ohne Inhalt zu prüfen
  - Einbinden eigener Shared-Object-Dateien möglich
  - Fork Bomb: in einer Endlosschleife Systemcall „fork“ aufrufen



https://de.wikipedia.org/wiki/Forkbomb

Fazit

- Sehr gute Performance bei einzelnen Newton-Cotes Berechnungen
- Exakte Ergebnisse bei Polynomen vom Grad < 4
- Ungenaue Näherung bei Polynomen mit höherem Grad
- Deutlich genauere Ergebnisse mit summierter Newton-Cotes Formel

Ausblick

- Integrationsgrenzen zu Floats umwandeln
- Perfomanz-Verbesserung mit Packed-SIMD-Befehle