Computerphysik Programmiertutorial 5b

Prof. Dr. Matteo Rizzi und Dr. Markus Schmitt - Institut für Theoretische Physik, Universität zu Köln

ILIAS: https://www.ilias.uni-koeln.de/ilias/goto_uk_crs_3862489.html

Github: https://github.com/markusschmitt/compphys2021

Inhalt dieses Notebooks: Timing (`BenchmarkTools`), Komplexität

Timing

Bei wissenschaftlichen Programmen kommt es oft auf die Effizienz an. Um die zu optimieren ist es wichtig die Resourcen bestimmen zu können, die zum Ausführen des Programms benötigt werden. Die wichtigen Resourcen sind Speicherplatz und Laufzeit. Wir werden uns hier auf Zeitmessungen beschränken.

In [1]:
```
function f(x)
    for j in eachindex(x)
        x[j] += 3.7
    end

    return nothing
end
```

Out[1]: `f (generic function with 1 method)`

Das Makro `@time` wird einem Funktionsaufruf vorangestellt um eine Ausgabe der verwendeten Resourcen zu erhalten:

In [2]:
```
@time f(rand(10000))
```

```
  0.000195 seconds (13 allocations: 97.766 KiB)
```

Achtung: Der erste Funktionsaufruf dauert länger, weil die Funktion dann just-in-time-kompiliert (jit-kompiliert) wird.

In [11]:
```
@time f(rand(10000))
```

```
  0.000030 seconds (2 allocations: 78.203 KiB)
```

`@elapsed` gibt nur die Laufzeit (in Sekunden) zurück.

In [4]:
```
@elapsed f(rand(10000))
```

Out[4]: `8.35e-5`

Mit dem Paket `BenchmarkTools` können genauere Messungen vorgenommen werden

In [5]:
```
using BenchmarkTools
```

In [13]:
```
@btime f(rand(10000))
```

```
  13.750 μs (2 allocations: 78.20 KiB)
```

In [14]:
```
@belapsed f(rand(10000))
```

Out[14]: `1.375e-5`

Komplexität

In [15]:
```
using PyPlot
using LinearAlgebra
```

Zur Anschauung definieren wir eine eigene Funktion für Matrixmultiplikation:

In [16]:
```
function my_matmul(A,B)
    C = Array{Float64}(undef,size(A)[1], size(B)[2])

    for j in 1:size(C)[2]
        for i in 1:size(C)[1]
            C[i,j] = 0.0
            for k in 1:size(A)[2]
                C[i,j] += A[i,k] * B[k,j]
            end
        end
    end

    return C
end
```

Out[16]: `my_matmul (generic function with 1 method)`

In [17]:
```
A = rand(100,100)
B = rand(100,100)

isapprox(my_matmul(A,B), A*B)
```

Out[17]: `true`

Benchmark unserer Funktion:

In [19]:
```
@btime my_matmul(A,B);
```

```
  655.708 μs (2 allocations: 78.20 KiB)
```

Benchmark der vorimplementierten Matrixmultiplikation:

In [20]:
```
@btime A * B;
```

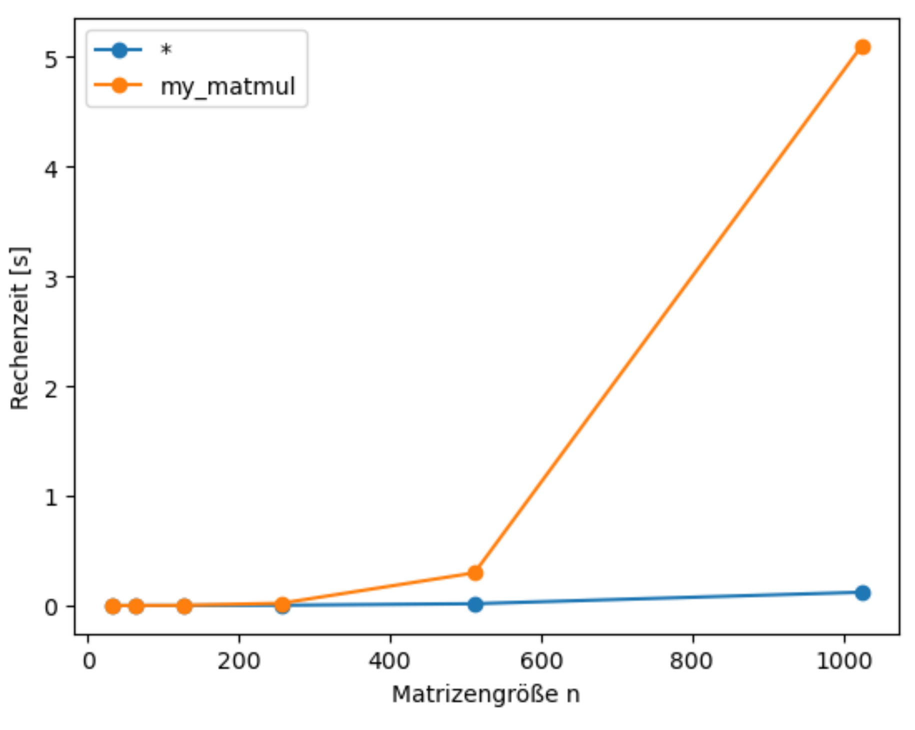
```
  52.416 μs (2 allocations: 78.20 KiB)
```

Schauen wir uns das etwas systematischer für verschiedene Matrixgrößen an:

In [22]:
```
BLAS.set_num_threads(1);

ns=2 .^ (collect(5:10))

times1=Float64[]
times2=Float64[]

for n in ns
    # zwei Matrizen erstellen
    A=rand(n,n)
    B=rand(n,n)

    # Zeit für Multiplikation messen
    t1 = @elapsed A*B
    t2 = @elapsed my_matmul(A,B)

    # Ausgabe
    println("n=$n:")
    println("  Zeit für A*B: $(t1)s")
    println("  Zeit für my_matmul(A,B): $(t2)s")

    # Messergebnis an Arrays anhängen
    push!(times1, t1)
    push!(times2, t2)
end
```

```
n=32:
  Zeit für A*B: 1.9792e-5s
  Zeit für my_matmul(A,B): 1.8791e-5s
n=64:
  Zeit für A*B: 3.9208e-5s
  Zeit für my_matmul(A,B): 0.000173833s
n=128:
  Zeit für A*B: 0.000268416s
  Zeit für my_matmul(A,B): 0.0018395s
n=256:
  Zeit für A*B: 0.00204725s
  Zeit für my_matmul(A,B): 0.01996625s
n=512:
  Zeit für A*B: 0.016147334s
  Zeit für my_matmul(A,B): 0.297469292s
n=1024:
  Zeit für A*B: 0.120631541s
  Zeit für my_matmul(A,B): 5.099413292s
```

In [24]:
```
plot(ns,times1, "-o", label="*")
plot(ns,times2, "-o", label="my_matmul")
legend()
xlabel("Matrizengröße n")
ylabel("Rechenzeit [s]")
```

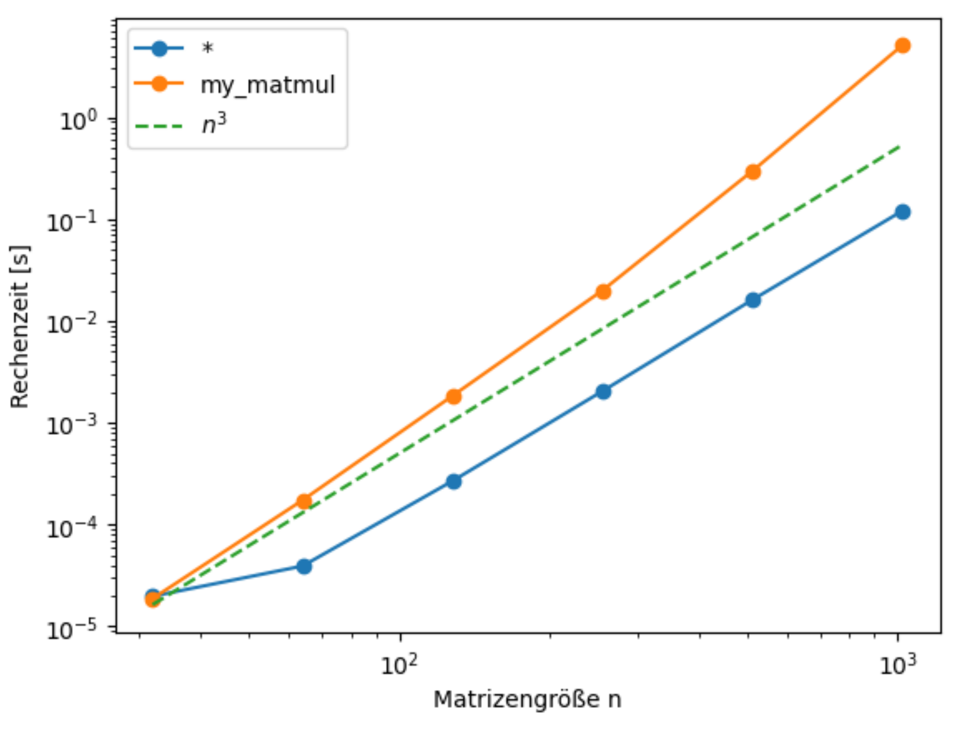


Out[24]: `PyObject Text(24.000000000000007, 0.5, 'Rechenzeit [s]')`

Die **Komplexität** eines Algorithmus gibt an wie die benötigten *Rechenresourcen* (Rechenzeit oder Speicherplatz) wachsen, wenn man das zu lösende Problem vergrößert. In unserem Beispiel ist die entspricht die Größe des Problems der Größe unserer $n \times n$-Matrizen.

Um eine Gesetzmäßigkeit zu finden, schauen wir noch einmal genauer hin, indem wir doppelt-logarithmisch plotten:

In [27]:
```
loglog(ns,times1, "-o", label="*")
loglog(ns,times2, "-o", label="my_matmul")
loglog(ns, 5e-10 * ns .^ 3, "--", label=L"$n^3$")
legend()
xlabel("Matrizengröße n")
ylabel("Rechenzeit [s]")
```



Out[27]: `PyObject Text(24.000000000000007, 0.5, 'Rechenzeit [s]')`

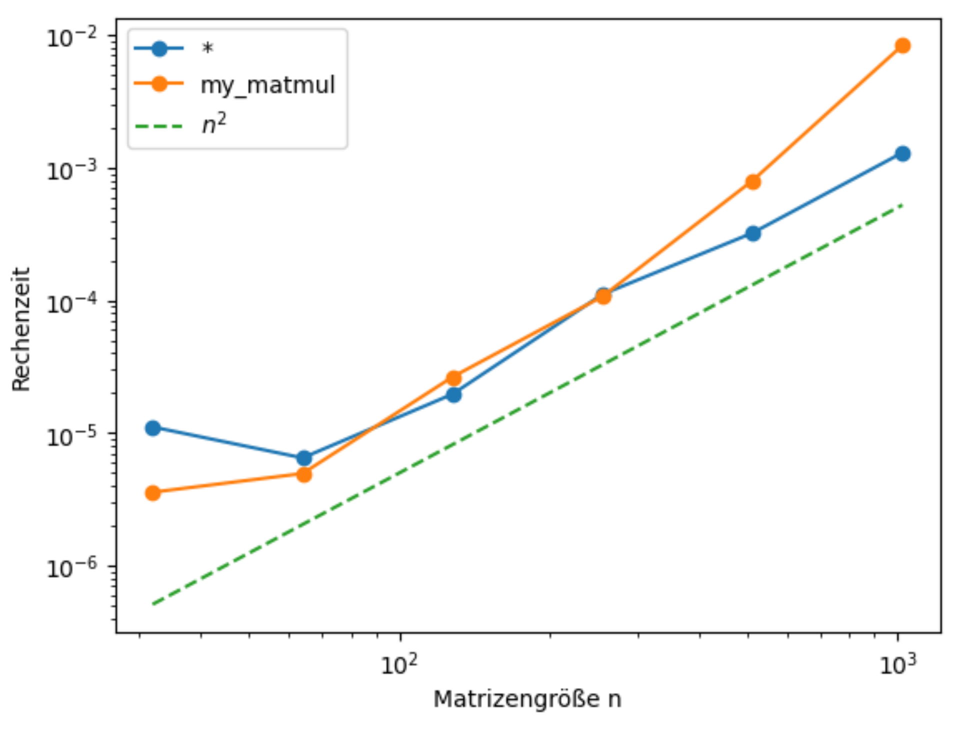
Die Komplexität von Algorithmen wird in O-Notation angegeben. Die Multiplikation von zwei $n \times n$ Matrizen wie wir sie hier implementiert haben ist z.B. $O(n^3)$. Das bedeutet, dass die Rechenzeit bei großen Matrizen kubisch wächst. Der Vorfaktor dieses Wachstumsgesetzes kann jedoch von den Details der Implementierung abhängen.

Eine Zusammenfassung der Komplexität bekannter Algorithmen steht auf Wikipedia.

In [28]:
```
ns=2 .^ (collect(5:10))
times1=Float64[]
times2=Float64[]

for n in ns
    # Matrix und Vektor erstellen
    A=rand(n,n)
    B=rand(n,1) # Diesmal Matrix-Vektor Produkt

    # Zeit für Multiplikation messen
    t1 = @elapsed A*B
    t2 = @elapsed my_matmul(A,B)

    # Ausgabe
    println("n=$n:")
    println("  Zeit für A*B: $t1")
    println("  Zeit für my_matmul(A,B): $t2")

    # Messergebnis an Arrays anhängen
    push!(times1, t1)
    push!(times2, t2)
end
```

```
n=32:
  Zeit für A*B: 1.1125e-5
  Zeit für my_matmul(A,B): 3.583e-6
n=64:
  Zeit für A*B: 6.5e-6
  Zeit für my_matmul(A,B): 4.958e-6
n=128:
  Zeit für A*B: 1.9667e-5
  Zeit für my_matmul(A,B): 2.65e-5
n=256:
  Zeit für A*B: 0.000111125
  Zeit für my_matmul(A,B): 0.000107125
n=512:
  Zeit für A*B: 0.000322292
  Zeit für my_matmul(A,B): 0.000799166
n=1024:
  Zeit für A*B: 0.001298792
  Zeit für my_matmul(A,B): 0.008359167
```

In [29]:
```
loglog(ns,times1, "-o", label="*")
loglog(ns,times2, "-o", label="my_matmul")
loglog(ns, 5e-10*ns.^2, "--", label=L"$n^2$")
legend()
xlabel("Matrizengröße n")
ylabel("Rechenzeit")
```



Out[29]: `PyObject Text(24.000000000000007, 0.5, 'Rechenzeit')`