

Algorithmik kontinuierlicher Systeme

Direkte Verfahren für Lineare Gleichungssysteme



- Direkte Verfahren, d.h. solche, die in endlich vielen Schritten das exakte Ergebnis liefern (exakte Rechnung voraus-gesetzt) basieren auf der Faktorisierung von A
 - ▶ $A = A_1 A_2$ dann ist $Ax = b$ äquivalent zu

$$A_1 y = b \quad \text{und} \quad A_2 x = y$$
- LR-Zerlegung
- Vorwärts/Rückwärts-Substitution für Dreiecksmatrizen: $O(n^2)$

[illegible]

- $N_{ij}(\alpha)$ A addiert zur i -ten Zeile von A das α -fache der j -ten.
- $(N_{ij}(\alpha))^{-1} = N_{ij}(-\alpha)$

$$\begin{bmatrix}
 1 & & & & \\
 & \ddots & & & \\
 & & 1 & & \\
 & & & \ddots & \\
 & & & & 1
 \end{bmatrix}
 \begin{bmatrix}
 0 & & & & \\
 \vdots & & & & \\
 0 & & & & \\
 & & & & \\
 & & & &
 \end{bmatrix}
 \begin{bmatrix}
 0 & & & & \\
 & & & & \\
 & & & & \\
 & & & & \\
 & & & &
 \end{bmatrix}
 \begin{bmatrix}
 r_{11} & \dots & * & * & * & \dots & * \\
 0 & \ddots & & \vdots & \vdots & & \vdots \\
 \vdots & \ddots & r_{j-1,j-1} & * & \vdots & & \vdots \\
 0 & & 0 & r_{jj} & * & \dots & * \\
 \vdots & & \vdots & 0 & \vdots & & \vdots \\
 0 & \dots & 0 & \tilde{a}_{ij} & * & \dots & * \\
 \vdots & & \vdots & * & \vdots & & \vdots \\
 \vdots & & \vdots & \vdots & \vdots & & \vdots \\
 0 & \dots & 0 & * & * & \dots & *
 \end{bmatrix}
 =
 \begin{bmatrix}
 r_{11} & \dots & * & * & * & \dots & * \\
 0 & \ddots & & \vdots & \vdots & & \vdots \\
 \vdots & \ddots & r_{j-1,j-1} & * & \vdots & & \vdots \\
 0 & & 0 & r_{jj} & * & \dots & * \\
 \vdots & & \vdots & 0 & \vdots & & \vdots \\
 0 & \dots & 0 & 0 & \tilde{*} & \dots & \tilde{*} \\
 \vdots & & \vdots & * & \vdots & & \vdots \\
 \vdots & & \vdots & \vdots & \vdots & & \vdots \\
 0 & \dots & 0 & * & * & \dots & *
 \end{bmatrix}$$

$$N_{ij}(\alpha) \cdot \tilde{A} = \tilde{A}$$

dabei ist $\alpha = -\frac{\tilde{a}_{ij}}{r_{jj}}$ und r_{jj} das Pivotelement

- Wenn man sich also einfach nur die Eliminationsfaktoren der Gauss-Elimination in einer unteren Dreiecksmatrix L merkt, generiert man eine Faktorisierung der Matrix $A = L R$
- Geschickte Programmierer verwenden hierzu genau den durch Elimination frei werdenden Speicherplatz in der Originalmatrix A

$$\begin{aligned}
 A &= N_1(-\alpha_1)N_1(\alpha_1)A && \text{(Elimination der 1. Spalte)} \\
 &= N_1(-\alpha_1)N_2(-\alpha_2)N_2(\alpha_2)N_1(\alpha_1)A && \text{(Elimination der 2. Spalte)} \\
 &= N_1(-\alpha_1)N_2(-\alpha_2)N_3(-\alpha_3)N_3(\alpha_3)N_1(\alpha_2)N_1(\alpha_1)A && \text{(3. Spalte)} \\
 &= \vdots \\
 &= L \cdot R
 \end{aligned}$$

$$L = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ l_{21} & 1 & \ddots & \vdots \\ \vdots & \cdots & \ddots & 0 \\ l_{n1} & \cdots & l_{n,n-1} & 1 \end{bmatrix} \quad R = \begin{bmatrix} r_{11} & r_{12} & \cdots & r_{1n} \\ 0 & r_{22} & & r_{2n} \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & r_{nn} \end{bmatrix}$$

- 1. Gauss-Scherung: $A_1 = L_1 A$
- 2. Gauss-Scherung: $A_2 = L_2 A_1 = L_2 L_1 A$
- ...
- (n-1)-te Gauss-Sch. $A_{n-1} = L_{n-1} A_{n-2} = L_{n-1} \dots L_2 L_1 A$
ist eine obere/rechte Dreiecksmatrix !!
- Setze $R = A_{n-1}$ und $L = (L_{n-1} \dots L_2 L_1)^{-1} = L_1^{-1} L_2^{-1} \dots L_{n-1}^{-1}$
dann gilt $LR = A$
- Bestimmung von L : Spalten der L_i ohne Minus-Zeichen!
- Bestimmung von R : i -te Zeile von A_{i-1} ($i=1,2,\dots,n$)

- Beispiel

$$\begin{bmatrix} 2 & 2 & -3 & 1 \\ 4 & 6 & -5 & 0 \\ -4 & -2 & 10 & -3 \\ 2 & 6 & -7 & -7 \end{bmatrix}$$

$L =$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & & & \\ -2 & & & \\ 1 & & & \end{bmatrix}$$

$$A = \begin{bmatrix} 2 & 2 & -3 & 1 \\ 4 & 6 & -5 & 0 \\ -4 & -2 & 10 & -3 \\ 2 & 6 & -7 & -7 \end{bmatrix}$$

$$A_1 = \begin{bmatrix} * & * & * & * \\ 0 & 2 & 1 & -2 \\ 0 & 2 & 4 & -1 \\ 0 & 4 & -4 & -8 \end{bmatrix}$$

 $R =$

$$\begin{bmatrix} 2 & 2 & -3 & 1 \\ 0 & & & \\ 0 & & & \\ 0 & & & \end{bmatrix}$$

$L =$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ -2 & 1 & & \\ 1 & 2 & & \end{bmatrix}$$

$$A = \begin{bmatrix} 2 & 2 & -3 & 1 \\ 4 & 6 & -5 & 0 \\ -4 & -2 & 10 & -3 \\ 2 & 6 & -7 & -7 \end{bmatrix}$$

 $R =$

$$\begin{bmatrix} 2 & 2 & -3 & 1 \\ 0 & 2 & 1 & -2 \\ 0 & 0 & & \\ 0 & 0 & & \end{bmatrix}$$

$$A_1 = \begin{bmatrix} * & * & * & * \\ 0 & 2 & 1 & -2 \\ 0 & 2 & 4 & -1 \\ 0 & 4 & -4 & -8 \end{bmatrix}$$

$$A_2 = \begin{bmatrix} * & * & * & * \\ * & * & * & * \\ * & 0 & 3 & 1 \\ * & 0 & -6 & -4 \end{bmatrix}$$

$$A = \begin{bmatrix} 2 & 2 & -3 & 1 \\ 4 & 6 & -5 & 0 \\ -4 & -2 & 10 & -3 \\ 2 & 6 & -7 & -7 \end{bmatrix}$$

$$L =$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ -2 & 1 & 1 & 0 \\ 1 & 2 & -2 & 0 \end{bmatrix}$$

$$R =$$

$$\begin{bmatrix} 2 & 2 & -3 & 1 \\ 0 & 2 & 1 & -2 \\ 0 & 0 & 3 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$A_1 = \begin{bmatrix} * & * & * & * \\ 0 & 2 & 1 & -2 \\ 0 & 2 & 4 & -1 \\ 0 & 4 & -4 & -8 \end{bmatrix}$$

$$A_2 = \begin{bmatrix} * & * & * & * \\ * & * & * & * \\ * & 0 & 3 & 1 \\ * & 0 & -6 & -4 \end{bmatrix}$$

$$A_3 = \begin{bmatrix} * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & 0 & -2 \end{bmatrix}$$

Pseudo-Code für LR (ohne Pivotsuche)

```

T = A;
for j = 1..n-1                # Spaltenindex
    for i = j+1..n            # Bearbeitung Spalte j
        T[i,j] = T[i,j]/T[j,j]    # Bestimmung L[i,j]
        for k = j+1..n        # Update Zeile i
            T[i,k] -= T[i,j]*T[j,k];

# Das folgende nur, wenn man L und R separat
# haben möchte und Speicher durch viele 0er verschwenden
for j = 1..n
    L[j,j] = 1;
    for i = 1..n
        if i <= j
            R[i,j] = T[i,j]
        else
            L[i,j] = T[i,j]

```

- Aufwandsabschätzung für LR-Zerlegung
(Analyse Pseudo-Code):

$$\text{Div.: } (n-1) + (n-2) + \dots + 1 = n(n-1)/2$$

$$\text{Mult.: } (n-1)^2 + (n-2)^2 + \dots + 1^2 = (n-1)n(2n-1)/6$$

$$\text{Add.: } (n-1)^2 + (n-2)^2 + \dots + 1^2 = (n-1)n(2n-1)/6$$

$$\text{insgesamt ca. } \frac{2n^3}{3}$$

- Komplexität der LR-Zerlegung: $O(n^3)$

- Verfahren funktioniert nur wenn **Pivot-Element** $a_{11} \neq 0$.
- Andernfalls müssen Zeilen-Vertauschungen durchgeführt werden (d.h. formal Multiplikationen des Gleichungssystems mit Permutationsmatrix $P_{i,j} = \dots$)
- wg. numerischen Stabilität sollte die Pivotsuche immer durchgeführt werden, zwingend wenn das Pivot-Element klein (relativ zu den Elementen der (Rest-)Spalte) ist.
- Eine mögliche Strategie: In der (Rest-)Spalte das betragsgrößte Element suchen und dies durch Zeilenvertauschen auf die Diagonale bringen.
- Ausnahme: Bei symmetrisch positiv definiten Matrizen ist die Pivotsuche überflüssig (→ Verfahren von Cholesky siehe unten)

- Wegen der erforderlichen Spalten-Pivotsuche muss auch die Faktorisierung noch um die (Zeilen-) Permutationen ergänzt werden:
 - ▶ „Dazwischenschieben“ von Vertauschungsoperationen $P_{i,j}$
- Die Permutationen können aber alle „nach links“ gezogen werden, dabei müssen auch die Einträge in den bereits berechneten Gauss-Scherungen (L_i -Matrizen) und der rechten Seite b vertauscht werden!
- so dass sich zusammen genommen formal eine Faktorisierung der Form

$$A = P L R$$

ergibt (alle Permutationen zusammenmultipliziert)

1. Suche in 1.Spalte Pivotelement und vertausche
1. mit i.-ter Zeile: $\rightarrow P_{1i}A$
2. Erste Gauss-Scherung: $\rightarrow L_1(P_{1i}A)$
3. Suche in 2.Rest-Spalte Pivotelement und vertausche 2.
Zeile: $\rightarrow P_{2i}(L_1(P_{1i}A))$
4. Zweite Gauss-Scherung: $\rightarrow L_2(P_{2i}(L_1(P_{1i}A)))$
5.
6. Suche in (n-1).Spalte Pivotelement und vertausche Zeilen
 $\rightarrow P_{n-1,i} \dots L_2 P_{2i} L_1 P_{1i} A$
7. (n-1).-te Gauss-Scherung $\rightarrow L_{n-1} P_{n-1,i} \dots L_2 P_{2i} L_1 P_{1i} A$

und setzt man nun $R = L_{n-1} P_{n-1,i} \dots L_2 P_{2i} L_1 P_{1i} A$ dann gilt:

$$A = P_{1i} L_1^{-1} P_{2i} \dots L_{n-2}^{-1} P_{n-1,i} L_{n-1}^{-1} R$$

Beachte, dass

$$L_1^{-1} P_{2i} = \begin{bmatrix} 1 & 0 & 0 \\ \alpha & 1 & 0 \\ \beta & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ \alpha & 0 & 1 \\ \beta & 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 \\ \beta & 1 & 0 \\ \alpha & 0 & 1 \end{bmatrix} = P_{2i} \tilde{L}_1$$

dabei geht \tilde{L}_1 aus L_1^{-1} durch Vertauschen der entsprechenden Elemente in der Spalte hervor!

Aus diesem Grund kann man alle P_i nach links ziehen, muss nur in den bereits berechneten L_i^{-1} die Einträge vertauschen

$$\rightarrow A = P_{1i} P_{2i} \dots P_{n-1,i} \tilde{L}_1 \tilde{L}_2 \dots \tilde{L}_{n-1} R = P \cdot L \cdot R$$

- Wh.: $A = P_{1i} P_{2i} \dots P_{n-1,i} \tilde{L}_1 \tilde{L}_2 \dots \tilde{L}_{n-1} R = P \cdot L \cdot R$
- Somit $Ax = b \Leftrightarrow LRx = P^{-1}b = P_{n-1,i} \dots P_{2,i} P_{1,i} b$
- Vorgehen: finde in erster Spalte größtes Element (Zeile i)
- **Vertausche in A und b Zeile 1 und Zeile i**
- Schreibe erste Zeile von R und erste Spalte von L und führe die Gauss-Elimination durch \rightarrow aktualisiertes **A** .
- Für $i = 2 \dots n-1$:
 - Finde größtes Element in der i -ten Restspalte
 - Vertausche die beiden, Zeilen' in A , b und L ;
 - Schreibe i -te Zeile von R und i -te Spalte von L und führe die Gauss-Elimination durch \rightarrow aktualisiertes **A** .

Aus Gründen der numerischen Stabilität ist bei der Gauss-Elimination die Pivot-Suche nicht nur erforderlich, wenn die Pivotelemente $=0$ sind, sondern auch dann, wenn die Pivotelemente nur „klein“ sind.

Andernfalls kann die Kondition des Gleichungssystems durch die Eliminationsschritte drastisch verschlechtert werden.

Spalten-Pivotsuche:

Suche Betrags-größtes Element der Spalte unterhalb der Diagonalen.

Tausche Zeilen so, dass das Betrags-größte Element auf der Diagonalen steht und zum Eliminieren aller anderen Elemente der Spalte verwendet wird.

Implementierung mittels Permutationsarray $P[i]$. Zugriff auf die Matrix dann mittels Indirektion:

$A[P[i], j]$

oder bei C/C++-Arrays, die sowieso Zeilen-Zeiger verwenden, durch Austauschen der Pointer auf die Zeilen.

Alternativ kann man auch eine *vollständige* Pivotsuche durchführen und Zeilen und Spalten (der jeweiligen Restmatrix) vertauschen. Dies muss jedoch am Ende rückgängig gemacht werden, da sich damit natürlich die Unbekannten vertauschen.

- Die LR-Zerlegung einer Matrix A kann man nutzen
 - ▶ zum Berechnen der Determinante von A

$$\det A = \det L \times \det R = 1 \cdot r_{11} r_{22} \dots r_{nn}$$
 - ▶ zum effizienten Lösen mehrerer Gleichungssysteme mit der selben Koeffizientenmatrix A und (vielen) rechten Seiten b
 - ★ Einmal die LR-Zerlegung von A bestimmen : $A = L R$
Aufwand $O(n^3)$
 - ★ Für jede rechte Seite:
 - Löse $L y = b$ mit Vorwärtssubstitution kostet $O(n^2)$ Operationen
 - Löse $R x = y$ mit Rückwärtssubstitution kostet $O(n^2)$ Operationen

Zwei Fälle unterscheiden:

1. Alle rechten Seiten am Anfang bekannt, sie werden in einer m -spaltigen Matrix B zusammengefasst.

Dies kann man mit einer LR Faktorisierung erreichen, dann m -mal Vorwärts- und Rückwärts-Substitution.

Aber auch einfach so:

- ▶ Löse $AX = B$ wobei X und B Matrizen mit n Zeilen und m Spalten
- ▶ durch die LR-Zerlegung der „erweiterten $n \times (n+m)$ -Matrix“ $(A | B)$
- ▶ $\rightarrow L$ und $(R | B^*)$
- ▶ Rückwärtsauflösen $RX = B^*$ (das Vorwärtsauflösen $LY = B$ entfällt)

2. Fallen die rechten Seiten sukzessive an (z.B. weil sie in einer Iteration durch wiederholtes Lösen von $Ax = b$ entstehen), dann muss man die Gleichungssysteme durch Faktorisierung lösen.

- ▶ die Vorwärts-Rückwärts-Substitution kostet dann **nur** $O(n^2)$ Operationen pro rechte Seite.
- ▶ Die Kosten für die Elimination will man nur einmal zahlen, wenn die Matrix gleich bleibt.
- ▶ Der Hauptrechenaufwand steckt in der Elimination bei der Faktorisierung: $O(n^3)$ Operationen.
- ▶ **Falsch ist:** Berechnen der inversen Matrix (siehe unten)

- ▶ Gauss-Elimination mit vollständiger (d.h. über Spalten und Zeilen) Pivotsuche für $m \times n$ -Matrix A .
- Wenn nach r Schritten, kein Pivot $\neq 0$ mehr gefunden wird ist $\text{Rang}(A) = r$ (exakte Rechnung vorausgesetzt!)
 - In der Praxis unbrauchbar, weil kleinste (Eingabe-)Störung oder Rundungsfehler dazu führen, dass

$$\text{Rang}(A) = \min(m, n)$$
- Die Frage nach dem Rang einer (mit Rundungsfehlern behafteten) Matrix ist so gesehen unsinnig und (mit numerischen Algorithmen) nicht vernünftig beantwortbar. Die richtige Frage lautet:
 - ▶ Wie groß ist der Abstand der gegebenen Matrix A zur Menge der Rang- r -Matrizen, oder
 - ▶ Wie klein kann δA gewählt werden, so dass

$$\text{Rang}(A + \delta A) \leq r$$
 (Problem wird später behandelt (unter SVD))

- Ist die Cramer'sche Regel eine Alternative?
- Aus der Mathematik: $x_i = \det A_{i,b} / \det A$, wobei $A_{i,b}$ dadurch entsteht, dass man in A die i -te Spalte durch b ersetzt (**Cramer'sche Regel**).
- Ist viel teurer als Gauss-Elimination (weil Determinanten sehr teuer sind – **außer** man verwendet Faktorisierung)
 - ▶ Wenn man aber die Faktorisierung hat, ist das Gleichungssystem eh so gut wie gelöst.
- Ist viel viel ungenauer als Gauss mit Pivotsuche.
- Die algorithmische Nutzung der Cramer'schen Regel ist ein schwerer Kunstfehler
- Die Cramer'sche Regel ist gut für theoretische Überlegungen, nie in einem Algorithmus

- Ist die Berechnung der Inversen Matrix eine Alternative?

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$$
- Dies ist auch teurer und wesentlich weniger genau als die Gauss-Elimination
- Selbst wenn man viele Gleichungssysteme mit der gleichen Matrix \mathbf{A} zu lösen hat, wird nicht die Inverse berechnet.
- Es ist günstiger (und genauer), eine LR -Zerlegung durchzuführen und die Gleichungssysteme mit Vorwärts-Rückwärtssubstitution zu lösen.
- Beachte: Die Kombination von Vorwärts- und Rückwärtssubstitution ist nicht teurer als die Multiplikation mit der Inversen Matrix: beides $O(n^2)$
- (Aber: Divisionen + Parallele Ausführung)
- z.B. falls \mathbf{A} tridiagonal, dann ist \mathbf{A}^{-1} voll besetzt!

- Es gibt nur wenige Anwendungen, in denen man die inverse Matrix explizit berechnen sollte.
Man sollte sehr sehr lange nachdenken, bevor man algorithmisch eine inverse Matrix ausrechnet.
- Weil es so paradox klingt, hier ganz explizit:
Selbst wenn man die Inverse A^{-1} exakt kennt, kann die Multiplikation $A^{-1}b$ numerisch nicht akzeptable Lösungen von $Ax = b$ erzeugen, während die Gauss-Elimination (mit Pivot) akzeptable Resultate erzeugt.
<http://de.mathworks.com/help/matlab/ref/inv.html>
- Dies liegt an (unvermeidbaren) Rundungsfehlern!
Darauf wird später noch genauer eingegangen.
Dazu ein Beispiel

Das Gleichungssystem $A\mathbf{x}=\mathbf{b}$ mit $A = \begin{bmatrix} a & a \\ a & a - \varepsilon \end{bmatrix}, \mathbf{b} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$

hat die exakte Lösung $\mathbf{x}_{exact} = \begin{bmatrix} 1/a \\ 0 \end{bmatrix}$ und es gilt

$$A^{-1} = \frac{1}{-a \cdot \varepsilon} \begin{bmatrix} a - \varepsilon & -a \\ -a & a \end{bmatrix} = \begin{bmatrix} 1/a - 1/\varepsilon & 1/\varepsilon \\ 1/\varepsilon & -1/\varepsilon \end{bmatrix}$$

Für $a=3.0, \varepsilon = .0000000001$ und Rechnung mit

single precision erhält man $A^{-1}\mathbf{b} = \begin{bmatrix} 0.29999999523 \\ 0.0 \end{bmatrix}$

Gleichungssystem $A\mathbf{x}=\mathbf{b}$ mit $A = \begin{bmatrix} a & a \\ a & a - \varepsilon \end{bmatrix}, \mathbf{b} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$

Die LR-Zerlegung von A ergibt

$$\mathbf{L} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}, \quad \mathbf{R} = \begin{bmatrix} a & a \\ 0 & (a - \varepsilon) - 1.0 \cdot a \end{bmatrix}$$

Vorwärtsaufösen: $\mathbf{y} = \begin{bmatrix} 1.0 \\ 1.0 - 1.0 \end{bmatrix} = \begin{bmatrix} 1.0 \\ 0.0 \end{bmatrix}$

Rückwärtsaufösen: $\mathbf{x} = \begin{bmatrix} (1.0 - 3.0 \cdot 0.0) / 3.0 \\ 0.0 \end{bmatrix} = \begin{bmatrix} 0.333333333333 \\ 0.0 \end{bmatrix}$

Aus Netlib: kostenlose SW höchster Qualität

LINPACK (veraltet)

- <http://www.netlib.org/linpack/index.html>

LAPACK (effizient u.a. wegen Cache-Blocking, etc.)

- <http://www.netlib.org/lapack/index.html>

Numerical Recipes: als Ausgangspunkt, falls man die SW selbst anpassen und weiterentwickeln möchte

Integrierte SW-Umgebungen:

Maple (stark im symbolischen Rechnen, eher schwach im numerischen Rechnen)

Matlab (teuer, schön, schick)

Octave: Poor mans version of Matlab:

- www.octave.org

Scilab:

- <http://scilabsoft.inria.fr/>

python/numpy/scipi

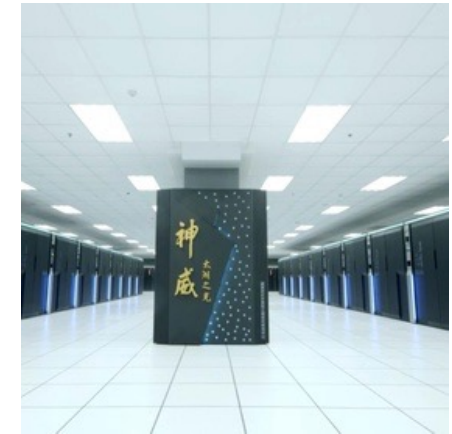
TOP-500: Linpack

<http://www.top500.org>

The Linpack Benchmark: As a yardstick of performance we are using the 'best' performance as measured by the LINPACK Benchmark. LINPACK was chosen because it is widely used and performance numbers are available for almost all relevant systems.

The LINPACK Benchmark was introduced by Jack Dongarra. A detailed description as well as a list of performance results on a wide variety of machines is available in postscript form from netlib. A parallel implementation of the Linpack benchmark and instructions on how to run it can be found at <http://www.netlib.org/benchmark/hpl/>.

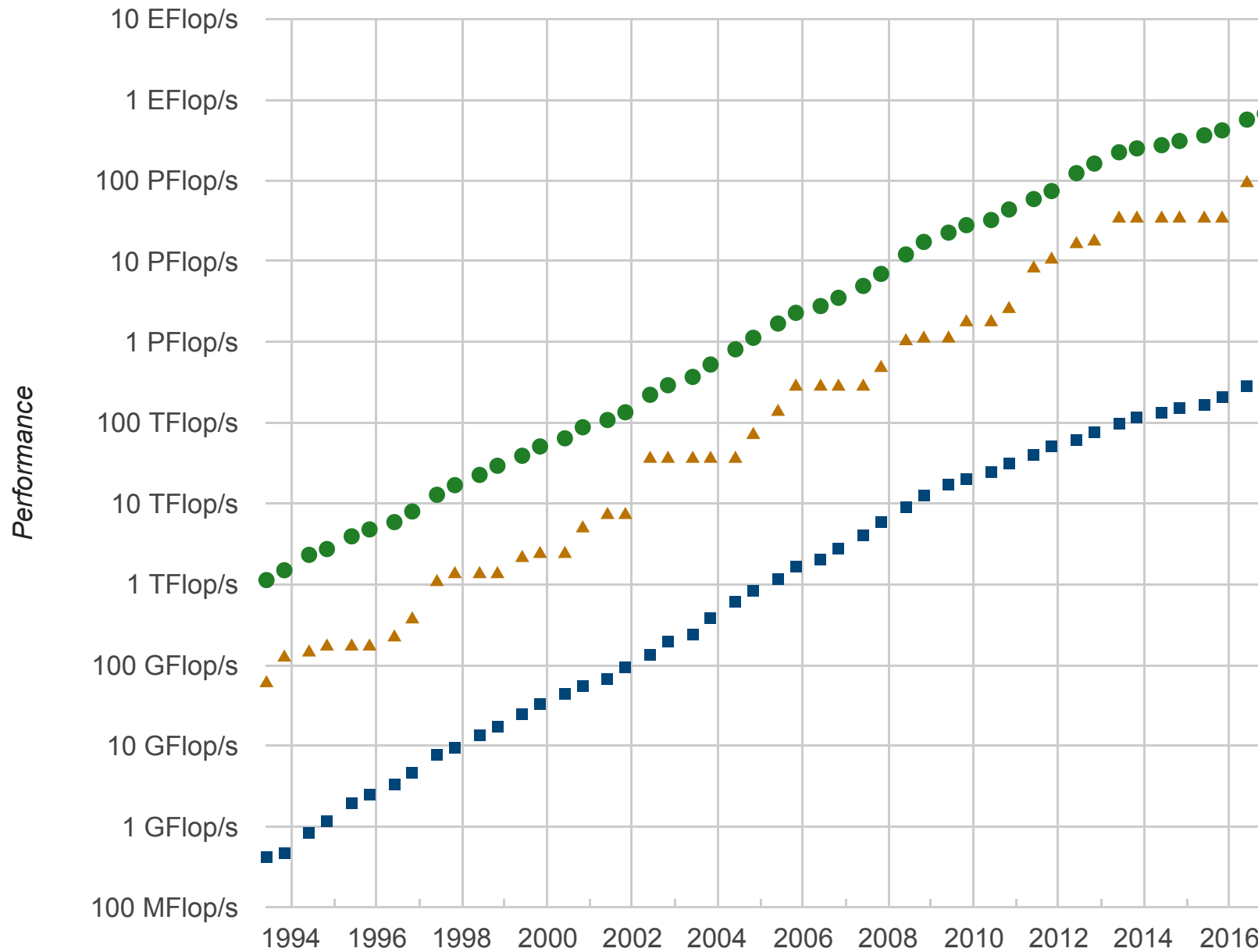
Rank	Site	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	National Supercomputing Center in Wuxi (/site/50623) China	Sunway TaihuLight - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway (/system/178764) NRCP	10,649,600	93,014.6	125,435.9	15,371
2	National Super Computer Center in Guangzhou (/site/50365) China	Tianhe-2 (MilkyWay-2) - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P (/system/177999) NUDT	3,120,000	33,862.7	54,902.4	17,808
3	DOE/SC/Oak Ridge National Laboratory (/site/48553) United States	Titan - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x (/system/177975) Cray Inc.	560,640	17,590.0	27,112.5	8,209



4	DOE/NNSA/LLNL (/site/49763) United States	Sequoia - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom (/system/177556) IBM	1,572,864	17,173.2	20,132.7	7,890
5	DOE/SC/LBNL/NERSC (/site/48429) United States	Cori - Cray XC40, Intel Xeon Phi 7250 68C 1.4GHz, Aries interconnect (/system/178924)	622,336	14,014.7	27,880.7	3,939
6	Joint Center for Advanced High Performance Computing (/site/50673) Japan	Oakforest-PACS - PRIMERGY CX1640 M1, Intel Xeon Phi 7250 68C 1.4GHz, Intel Omni-Path (/system/178932) Fujitsu	556,104	13,554.6	24,913.5	2,719
7	RIKEN Advanced Institute for Computational Science (AICS) (/site/50313) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect (/system/177232) Fujitsu	705,024	10,510.0	11,280.4	12,660

8	Swiss National Supercomputing Centre (CSCS) [/site/50422] Switzerland	Piz Daint - Cray XC50, Xeon E5-2690v3 12C 2.6GHz, Aries interconnect , NVIDIA Tesla P100 [/system/177824] Cray Inc.	206,720	9,779.0	15,988.0	1,312
9	DOE/SC/Argonne National Laboratory [/site/47347] United States	Mira - BlueGene/Q, Power BQC 16C 1.60GHz, Custom [/system/177718] IBM	786,432	8,586.6	10,066.3	3,945
10	DOE/NNSA/LANL/SNL [/site/50334] United States	Trinity - Cray XC40, Xeon E5-2698v3 16C 2.3GHz, Aries interconnect [/system/178610] Cray Inc.	301,056	8,100.9	11,078.9	4,233
11	United Kingdom Meteorological Office [/site/49064] United Kingdom	Cray XC40, Xeon E5-2695v4 18C 2.1GHz, Aries interconnect [/system/178925] Cray Inc.	241,920	6,765.2	8,128.5	
12	CINECA [/site/47495] Italy	Marconi Intel Xeon Phi - CINECA Cluster, Intel Xeon Phi 7250 68C 1.4GHz, Intel Omni-Path [/system/178937] Lenovo	241,808	6,223.0	10,833.0	

Performance Development



- Bislang etwa Steigerung der Leistung um den Faktor 1000 alle 10 Jahre
- Exascale: 10^{18} FLOPS
- ursprünglich Exascale für 2018 erwartet, jetzt vielleicht in 2022
- Sunway TaihuLight in China ist schneller als alle deutschen Hochleistungsrechner zusammen genommen