Inhaltsverzeichnis

1 Einleitung

1.1 Die Programmiersprache Julia

Die Programmiersprache Julia ist eine noch sehr junge (2012) höhere dynamische High Performance Programmiersprache, die hauptsächlich für numerisches Rechnen entwickelt worden ist. Dabei wurde insbesondere darauf geachtet, dass der Code einfach zu schreiben ist und es trotzdem eine hohe Ausführungsgeschwindigkeit gibt. Wo beispielsweise Python den Fokus haupsächlich darauf legt, dass der Code einfach zu schreiben ist und C demhingegen sehr schwer zu schreiben ist, aber dafür eine sehr viel bessere Ausführungsgeschwindigkeit hat, versucht Julia beides zu vereinen.
Julia macht die Sprache an sich schnell, da sie das sogenannte "Zwei-Sprachen-Problem" anders angehen. Bei Julia soll es möglich sein alles in Julia selbst zu programmieren. Nur grundlegende Funktionen, wie beispielsweise Integer Operationen, For Schleifen, Rekursion oder Float Operationen benutzen C Operationen oder maipulieren C Structs. Das besondere außerdem ist, das Julia im Grunde eine dynamische Sprache ist, aber trotzdem den Code in Maschienen Code umwandelt, was eigentlich nur typisch für statische Programmiersprachen ist.
Zudem ist Julia auch so entwickelt worden, um Paralleles Arbeiten, egal ob Shared Memory Computing oder Distributed Memory Computing zu erleichtern oder zu automatisieren

1.2 Motivation

Ich möchte in meiner Arbeit herausfinden, in wie weit Julia in der Lage ist diese Versprechen zu erfüllen. Dafür werde ich numerische Programme in Julia schreiben und diese teilweise mit Python vergleichen, aber auch in Julia auf verschiedene Möglichkeiten eingehen, Code zu schreiben. Zudem werde ich später auch auf die Parallelisierbarkeit von Julia eingehen. Ich werde dies auch auf Juropa3 testen und feststellen in wie weit Julia eine Möglichkeit ist, damit auf Supercomputern zu arbeiten.
Ich habe bereits einige Tests durchgeführt um die Timings auf der Webseite von Julia zu bestätigen.(Hier der Vergleich der Webseite und meinem Zeug) Ich habe allerdings auch einen Fall defunden, in dem Julia langsamer als Python ist, indem man Numpy und Cython benutzt.
Ich werde im Verlauf dieser Arbeit einen Mehrgitter Algoritmus implementieren. Es geht also nicht nur um die technischen Aspekte der Programmiersprache Julia, sonder der mathematische Inhalt dieser Arbeit ist genau so wichtig. Grob gesagt ist der Mehrgitter Algorithmus dazu da ein Lineares Gleichungssystem in der Laufzeit O(n) annähernd zu lösen. Dafür braucht man aber einiges an mathematischem Vorwissen, welches ich im folgenden vorstellen werde.

2 Mathematisches Vorwissen

2.1 Iterative Löser

Iterative Löser werden benutzt um große lineare Gleichungssysteme in der Form: $Ax = b$ aufzulösen. Sie berechnen eine Annäherung an die Lösung bis zu einem bestimmten Fehler ϵ. Dadurch sind sie viel schneller als direkte Lösungsverfahren für lineare Gleichungen. Ein direkter Löser berechnet die genau Lösung eines linearen Gleichungssystems. Iterative Löser eignen sich besonders gut um Systeme aufzulösen, in denen die A Matrix sehr viele Nulleinträge hat.
Es gibt verschiedene Iterative Lösungsverfahren. Ich werde hier allerdings nur drei vorstellen. Im Grunde ist es aber immer das gleiche Vorgehen, welches aber für jedes Verfahren modifiziert wird. Das mathematische Vorgehen ist dabei, dass man in jedem Iterationsschritt ein neues x_{k+1} ausrechnen mit der Formel

$$x_{k+1} = Mx_k + b.$$

M ist dabei Verfahrensabhängig. Die Idee zu dieser Gleichung kommt von der Fixpunktgleichung $x = (I - A)x + b$.
Am Anfang des Verfahrens muss angegeben werden, wie genau die Annäherung sein soll. Vor jeder Iteration wird dabei geprüft, ob

$$\|b - Ax_k\| < \epsilon.$$

Wenn dies erfuellt ist, brechen die Iterationen ab und das aktuelle x_k ist das gesuchte.

2.1.1 Richardson

Richardson ist das am einfachsten umzusetzende Verfahren. Es ist aber in der Praxis kaum anwendbar, da es im Vergleich zu den anderen Verfahren sehr viele Iterationen braucht und oft nicht konvergiert. Die Idee ist, dass man mit der Fixpunktgleichung $x = (I - A)x + b$ arbeitet. Daraus ergibt sich das

$$M_{Rich} = I - A$$

ist. Um dieses Verfahren zu verbessern kann man mit Vorkonditionierern arbeiten. Das bedeutet, dass man grundsätzlich nicht die Gleichung $Ax = b$ löst, sondern die Gleichung

$$BAX = Bb$$

lösen. Durch intelligentes wählen von B kann man das Gleichungssystem so sehr vereinfachen. B sollte man so wählen, dass es nah an A^{-1} ist oder die Gleichung einfacher zu berechnen wird. Im speziellen Fall bei Richardson, dass $B = I$ ist, also die Matrix einfacher zu berechnen macht. Daraus folgt,

dass ich $M = I$ wählen kann. Deswegen verändert sich auch die Iterationsberechnung:

$$x_{k+1} = Mx_k + Bb$$

2.1.2 Jacobi

Das Jacobi Verfahren, ist ein Verfahren, welches mit Matrix Splitting funktioniert. Die Idee beim Matrix Splitting ist, dass man die Matrix aufteilt in einen Teil der einfach zu invertieren ist und einen der schwer zu invertieren ist. Hier teilt man die Matrix als erstes in eine untere Dreiecksmatrix L, die Diagonalmatrix D und die obere Dreiecksmatrix U. Daraus folgt, dass

$$A = L + D + U.$$

Dies bestimmt nun den Vorkonditionierer B. Für das Jacobi Verfahren wähle ich $B_{Jac} = D^{-1}$. Daraus folgt, dass

$$M_{Jac} = I - D^{-1}A.$$

2.1.3 Gauss Seidel

Da man mit desto mehr Informationen auch niedrigere Iterationszahlen erreichen kann benutzt man für das Gauss-Seidel Verfahren $B_{GS} = (L+D)^{-1}$. Daraus folgt

$$M_{GS} = I - (L + D)^{-1}A \Leftrightarrow M_{GS} = -(L + D)^{-1}U.$$

Vom Gauss-Seidel Verfahren gibt es auch andere Varianten wie den Backward Gauss Seidel bei dem

$$M_{BGS} = -(U + D)^{-1}L$$

oder den Symmetrischen Gauss Seidel

$$M_{SGS} = M_{GS}M_{BGS}.$$

Mit diesen Verfahren arbeite ich allerdings nicht, da diese nicht wichtig für das Mehrgitterverfahren sind.

2.1.4 Konvergenz

Um zu überprüfen, ob diese Verfahren konvergieren, berechnet man den Spektralradius der Iterationsmatrix. Wenn $p(M) < 1$ so konvergiert das Verfahren. Je kleiner der Spektralradius, desto schneller konvergiert das Verfahren.

2.2 Fouriertransformation

Periodische Funktionen, wie etwa $f(x) = sin(x)$ können in einem Bereich mit Breite l an bestimmten Stützstellen abgetastet werden und daraufhin vektorisiert werden. Diese Stellen sind äquidistant im Abstand $\Delta h = \frac{l}{N+1}$, wobei N die Anzahl der Stützstellen darstellt. Diese Werte werden in einen Vektor der Länge N geschrieben, da die Randwerte nicht beachtet werden.

2.3 Zentriertes finites Differenzenverfahren mit Dirichlet Null Randbedingung

Dis Idee hinter dem zentrierten finiten Differenzenverfahren ist, eine Approximation der Ableitung durch den Differenzenquotienten zu schaffen. Es gilt also:

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \Rightarrow f'(x) \approx \frac{f(x+h) - f(x)}{h}.$$

Wenn h klein genug ist gilt dementsprechend:

$$f'(x+h) = \frac{f(x+h) - f(x)}{h}.$$

Ich möchte allerdings:

$$f''(x) = (f')'(x) \approx$$

2.4 Prolongation

2.5 Restriktion

3 Mehrgitter

3.1 Aufgabenstellung

Meine Aufgabe ist es, mithilfe das Mehrgitter Algorithmus ein lineares zweidimensionales Gleichungssystem der Form $Ax = b$ zu lösen. Die Matrix A wähle ich dabei so, dass diese die Matrix ist die aus dem Finiten Differenzen Verfahren entsteht(gezeigt in 2.3). Da diese Matrix eine symetrische positiv definite Matrix ist, und zudem nur Elemente auf der Hauptiagonalen und den beiden Nebendiagonalen stehen hat. Den Vektor b waehle ich als Nullvektor. Den Vektor $x$initialisiere ich mit den Werten die bei der Fouriertransformation mit der Funktion $f(x, y)$ entsteht.

3.2 Was ist ein Mehrgitter?

Der Mehrgitter Algorithmen wird benutzt um eine näherungsweise Lösung für Gleichungssysteme zu berechnen. Der Vorteil dieses Algorithmus ist es, dass er für meine Problemstellung das LGS in der Laufzeit $O(n)$ lösen kann.