Inhaltsverzeichnis

1 Einleitung

Ich habe angefangen mit der Programmiersprache Julia zu arbeiten, um zu testen, ob diese leicht zu lernen, einfach zu parallelisieren und dabei noch schnell ist, fuer verschiedene numerische Anwendungen. In meinen anfaenglichen Beispielen, vergleiche ich Julia mit Python. Python ist eine universelle, blicherweise interpretierte höhere Programmiersprache. Sie will einen gut lesbaren, knappen Programmierstil fördern.So wird beispielsweise der Code nicht durch geschweifte Klammern, sondern durch Einrckungen strukturiert.(Wikipedia) Julia hingegen ist eine höhere High-Performance-Programmiersprache, welche vor allem für numerisches und wissenschaftliches Rechnen entwickelt wurde, während sie dennoch als eine General Purpose Language verwendet werden kann, bei gleichzeitiger Wahrung einer hohen Ausfhrungsgeschwindigkeit.(Wikipedia) Julia ist eine Programmiersprache, die auf der einen Seite sehr einfach zu lernen ist und trotzdem eine hohe Ausfhrungsgeschwindigkeit hat. Man kann sehr gut numerisch mit Julia arbeiten. Alle nötigen Methoden sind vorimplementiert und die zugehörige Doku ist bis jetzt sehr bersichtlich aber trotzdem gut verständlich. Zudem soll mit Julia die Parallelisierung von Programmen einfacher sein als in Python. Dies ist fr diese Tests nicht wichtig, aber kann im weiteren Verlauf noch wichtig werden. Die mathematischen Laufzeitvortest die ich durchgefhrt habe, basieren auf den Angaben der Webseite von `http://julialang.org/` und selbst ausgetesteten numerischen Funktionen wie beispielsweise der LU Zerlegung. Ich konnte dabei bestätigen, dass Julia größtenteils schneller ist als Python. Die Ausnahme, die ich bestätigen konnte, war ein `https://github.com/kbarbary/website/blob/master/posts/julia-vs-numpy-arrays.rst` von Kale Barbary berprft in dem es um einen Fall geht, in dem Python teilweise schneller ist als Julia. Dieses Ergebnis konnte ich bestätigen.

2 Gegebenheiten

Ich arbeite mit Python 3.5 in der Umwicklungsumgebung Pycharm. In Julia arbeite ich mithilfe Des Jupyter Notebooks in der Juliaversion 0.4.7.

3 Was sind Iterative Löser?

Iterative Löser werden benutzt um große lineare Gleichungen in der Form: $Ax = b$ aufzulösen. Sie berechnen eine Annäherung an die Lösung. Dadurch sind sie viel schneller als direkte Lösungsverfahren für lineare Gleichungen. Dies gilt vor allem, wenn A eine dünn besetzte Matrix ist. Das bedeutet das A viele Nulleinträge hat. Die grundlegende Idee eines iterativen Lösers ist durch eine immer genauere Annäherung an die Lösung bis zu einem bestimmtem Fehler eine möglichst genaue Lösung herauszufinden. Dieser Feh-

ler ist der Unterschied zwischen den letzten beiden Iterationen. Dieser soll kleiner sein als ein Residuum ϵ. Um dies Umzusetzen gibt es verschiedene Verfahren. Ich habe im Grunde mit drei verschiedenen Verfahren gearbeitet: Dem Richardson Verfahren, dem Jacobi Verfahren und dem Gauss-Seidel Verfahren. Zudem habe ich das Jacobi Verfahren und das Gauss-Seidel Verfahren optimiert.

4 Verfahren

4.1 Wie ist der grundlegende Aufbau der Verfahren?

Verfahrenabhängig wird die Iterationsmatrix M aus der Matrix A gebildet. Solange die Abbruchbedingung nicht erfllt ist,wird immer wieder die Gleichung:

$$x_{k+1} = Mx_k + b \tag{1}$$

4.2 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

$$x = (I - A)x + b \tag{2}$$

(I ist eine Einheitsmatrix mit gleichen Dimensionen wie A)arbeitet. Daraus ergibt sich das

$$M_{Rich} = I - A \tag{3}$$

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

$$BAX = Bb \tag{4}$$

loesen. 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:

$$x_{k+1} = Mx_k + Bb \tag{5}$$

4.3 Verfahren mit Matrix Splitting(Jacobi, Gauss-Seidel)

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 das

$$M_{Jac} = I - D^{-}1A. \tag{6}$$

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

$$M_{GS} = I - (L+D)^{-1}A \Leftrightarrow M_{GS} = -(L+D)^{-1}U. \tag{7}$$

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

$$M_{BGS} = -(U+D)^{-1}L \tag{8}$$

oder den Symmetrischen Gauss Seidel

$$M_{SGS} = M_{GS}M_{BGS}, \tag{9}$$

die ich aber nicht implementiert habe, da man keine großartige Verbesserung von diesen erwarten kann.(Das vielleicht noch mehr erklären!)

4.4 Wann funktionieren die Verfahren?

Um zu beurteilen ob eines dieser Verfahren funktioniert, muss ich überprüfen, ob dieses konvergiert. Ausschlaggebend dafür ist der Spektralradius. Dieser wird durch die Eigenwerte der Iterationsmatrix M bestimmt. Da ich mich dafür entschieden habe mit der Maximumsnorm zu arbeiten ist der größte Eigenwert ausschlaggebend. Wenn dieser Wert kleiner als eins ist konvergiert das Verfahren. Je kleiner der Spektralradius ist, desto schneller konvergiert das Verfahren. (Wahrscheinlich besser erst nach den Verfahren, dann ist das mit dem Spektralradius leichter).

4.5 Aufbauende Verfahren(SOR, gewichtetes Jacobi Verfahren)

Um die Splitting Verfahren zu verbessern, kann man diese mit gewissen Parametern multiplizieren, damit diese schneller werden. Das Jacobi Verfahren wird zum gewichteten Jacobi Verfahren indem man den Parameter $w = 2/3$ mit M multipliziert. Das Gauss Seidel Verfahren kann man verbessern, indem man das Gauss Seidel Verfahren $w = \frac{2}{2+sin(\pi * \frac{1}{n+1})}$.(Das gefällt mir noch nicht so gut. Ich weiß aber nicht wie sonst. Soll ich da wirklich ne herleitung machen?)

5 Implementierung

Ich habe bei der Implementierung darauf geachtet, dass ich in Julia und Python das Gleiche mache. Ich habe in beiden Programmen mit Sparse Matrizen gearbeitet, da ich so die Vorteile der dünn besetzten Matrix A gut ausnutzen kann und die Laufzeit und den Speicherverbrauch optimieren kann. Das Problem bei Sparse Matrizen ist in Julia, dass man diese nicht invertieren kann, und ich so meinen vorherigen Plan mit Inversen zu arbeiten aufgeben musste. Ich habe die Gleichung also dementsprechend geändert(am Beispiel Jacobi):

$$x_{k+1} = Mx_k + Bb \tag{10}$$

$$\Leftrightarrow x_{k+1} = (I - D^{-1} * A)x_k + D^{-1}b \tag{11}$$

$$\Leftrightarrow x_{k+1} = Ix_k - D^{-1}Ax_k + D^{-1}b \tag{12}$$

$$\Leftrightarrow x_{k+1} = x_k + D^{-1}(b - Ax_k) \tag{13}$$

Da $B = D^{-1}$ lautet die Iterationsgleichung allgemein:

$$x_{k+1} = x_k + B(b - A * x_k) \tag{14}$$

In meiner Berechnung wende ich nur einen Trick an ich berechne immer:

$$B^{-1}v = b - Ax_k x_{k+1} = x_k + v \tag{15}$$

Da B^{-1} nun die Inversen einer Inversen ist, kann ich diesen Algorithmus implementieren. Ich ueberpruefe das Ergebnis mithilfe eines direkten Loesers. Also berechne ich den entstandenen Fehler und den Spektralradius. Bei genauerer Betrachtung der vorimplementierten Methoden zum Lösen von linearen Gleichungssystemen ist mir aufgefallen, dass diese verschieden implementiert sind. In Julia wurde eine intelligente Implementierung gewählt welche das LGS mit einsetzen löst als das stumpfe Vorgehen in Python. Um die Verfahren anzugleichen habe ich den Ansatz über eine LU-Zerlegung gewählt. Die LU-Zerlegung teilt die Matrix in eine obere und eine untere Dreiecksmatrix. Da die Matrix die dafür genutzt wird aber entweder eine Untere Dreiecksmatrix oder eine Diagonalmatrix ist, muss das Verfahren im Grunde nichts machen. Die Methode die man daraufhin zum lösen benutzen kann ist in Julia und Python vom Vorgehen die gleiche.

6 Vergleich der einzelnen Verfahren

Um die einzelnen Verfahren zu testen benutze ich für A eine dünn besetzte, symmetrische tridiagonale und positiv definite Matrix, wobei am wichtigsten ist, dass diese dünn besetzt ist. In diesem Fall ist es die Matrix die bei der Diskretisierung des Laplace-Operators mit Finiten Differenzen zweiter Ordnung mit Dirichlet-Randbedingungen in einer Dimension entsteht. Sie wie folgendermaßen erzeugt:
vorfaktor = (dimension+1)*(dimension+1)
stencil = [vorfaktor*-1, vorfaktor*2, vorfaktor*-1]
A = sparse.diags(stencil, [-1, 0, 1], shape=(dimension, dimension))
Diese Matrix hat zweien auf der Hauptdiagonalen und minus Einsen auf den beiden Nebendiagonalen. Multipliziert wird die Matrix mit $(n+1)^2$. Man

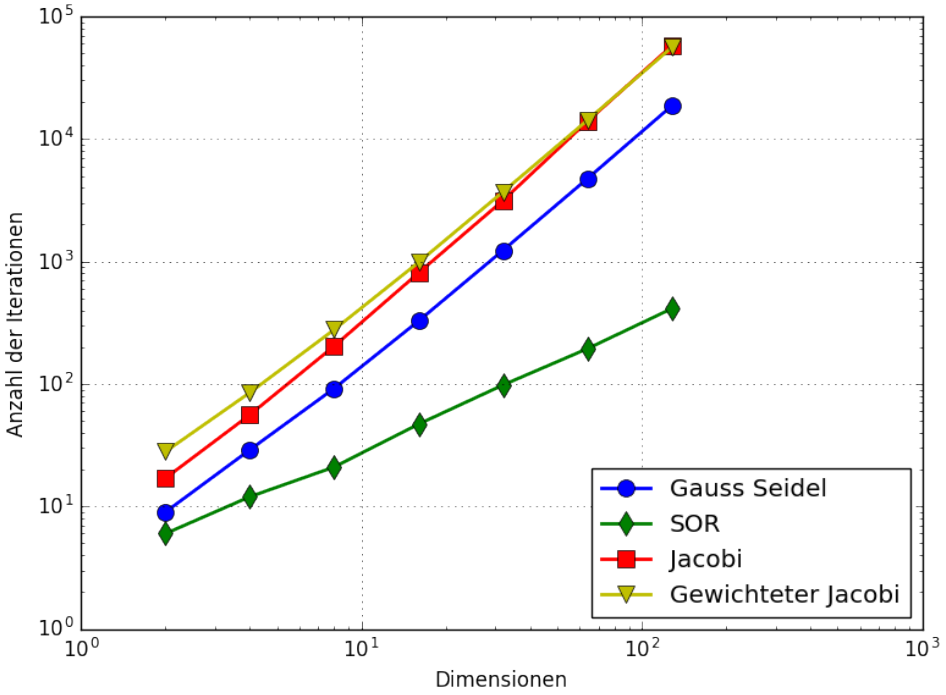

Abbildung 1: Vergleich der grundlegenden Verfahren

kann sehen das das SOR Verfahren das Verfahren ist, dass die wenigsten Iterationen braucht. Am schlechtesten schneidet das optimierte Jacobi Verfahren ab, da diese für einen speziellen Fall ist und nicht für diesen. Es wurde bestätigt, dass man mit mehr Informationen auch die Anzahl der Iterationen verringern kann.

7 Vergleich zwischen Julia und Python anhand des SOR Verfahren

Ich habe gesehen, dass das SOR Verfahren das schnellste Verfahren ist und zudem die wenigsten Iterationen braucht. Deswegen habe ich beschlossen meinen Vergleich zwischen Julia und Python mit dem SOR Verfahren zu machen. Ich benutze die gleichen Vektoren wie beim Vergleich der einzelnen Verfahren. Dafür habe ich das Verfahren mit verschieden großen Matrizen und verschiedenen Residuen getestet. Die Größe der Matrizen liegt zwischen 200x200 und 1600x1600. Das Residuum lasse ich von 10^{-1} bis 10^{-9} laufen. Anhand dieses Diagramms, kann man sehen, dass je kleiner das Residuum

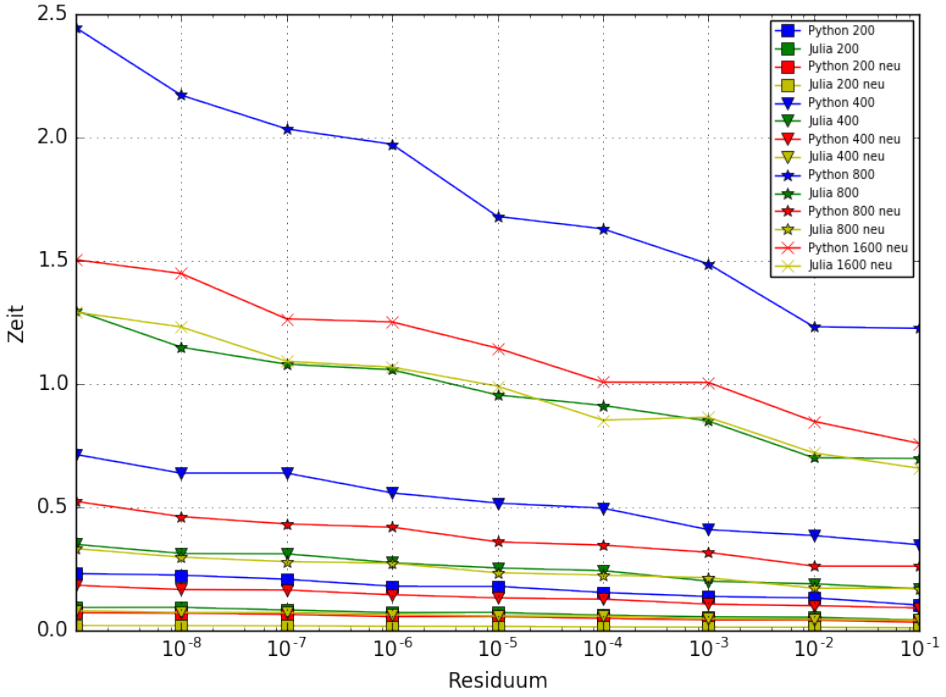

Abbildung 2: Vergleich SOR Verfahren aufgrund deren Matrixgroesse

ist, desto länger braucht das lösen des LGS. Zudem kann man erkennen, dass je größer die Matrix ist, desto mehr Zeit braucht der Löser. Hier ist auch ungefähr ein quadratischer Zusammenhang zu erkennen. Es fällt zudem auf, dass das lösen ohne die LU Zerlegung bei gleicher Größe und gleicher Iterationsanzahl immer langsamer ist als das lösen mit LU Zerlegung.Man kann außerdem erkennen, dass das Verfahren mit LU Zerlegung in Julia am schnellsten ist. Darauf folgt das LU-Verfahren mit Python und danach das normale lösen in Julia und am langsamsten ist das lösen mit Python.

8 Überprüfung der Ergebnisse mithilfe des Jacobi Verfahren

Um zu überprüfen, dass dieses nicht nur beim SOR Verfahren auftritt, habe ich mich entschlossen dieses mithilfe des Jacobi Verfahren zu überprüfen. Da aber die Test mit den gleichen Bedingungen wie beim SOR Verfahren sehr viel länger brauchen als bei den Tests des SOR Verfahren, habe ich mich dazu entschlossen das gewichtete Jacobi Verfahren zu benutzen und den Vektor x0 als Fourier Mode zu setzen. Das bedeutet ich benutze eine Sinusschwingung, die durch ein Parameter k bestimmt wird, welcher abhängig von der Größe n der Matrix ist. Die Sinusschwingung betrachte ich zwischen 0 und 1. Um verschiedene Werte für den Vektor zu bekomme, benutze ich die Werte im Abstand von 1/n+1. Hier die Formel für den i-ten Eintrag im Vektor. Ich benutze den gewichteten Jacobi, da er optimal für diese Problemstellung ist. Je größer man k dabei wählt, desto schneller und mit weniger Iterationen kann der iterative Löser das LGS lösen. Man sieht in

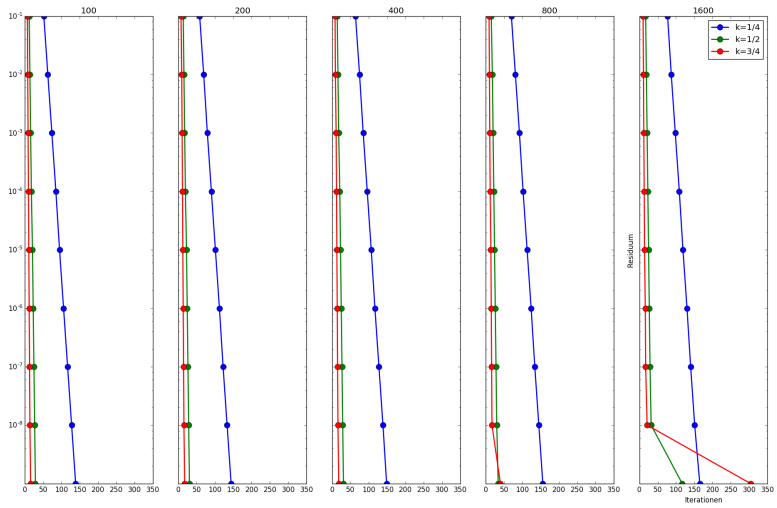

Abbildung 3: Gewichteter Jacobi anhand der Matrixgroesse

diesem Diagramm, dass egal wie groß man die Matrix wählt man immer die gleiche Anzahl der Iterationen, bei allen Matrixgrößen für gleiches Residuum sieht. Wenn man allerdings die Matrizen zu groß wählt und das Residuum zu klein, passiert etwas was ich nicht erklären kann. Ich wähle für Zeitvergleich zwischen Python und Julia k=1/10, da bei größeren Werten der Algorithmus zu schnell wird und ich zu ungenaue Ergebnisse bekomme, da die Iterationszeit im Millisekunden Bereich liegt. Man kann erkennen, dass sich die Ausführungszeiten ähnlich verändern wie bei dem SOR Verfahren.

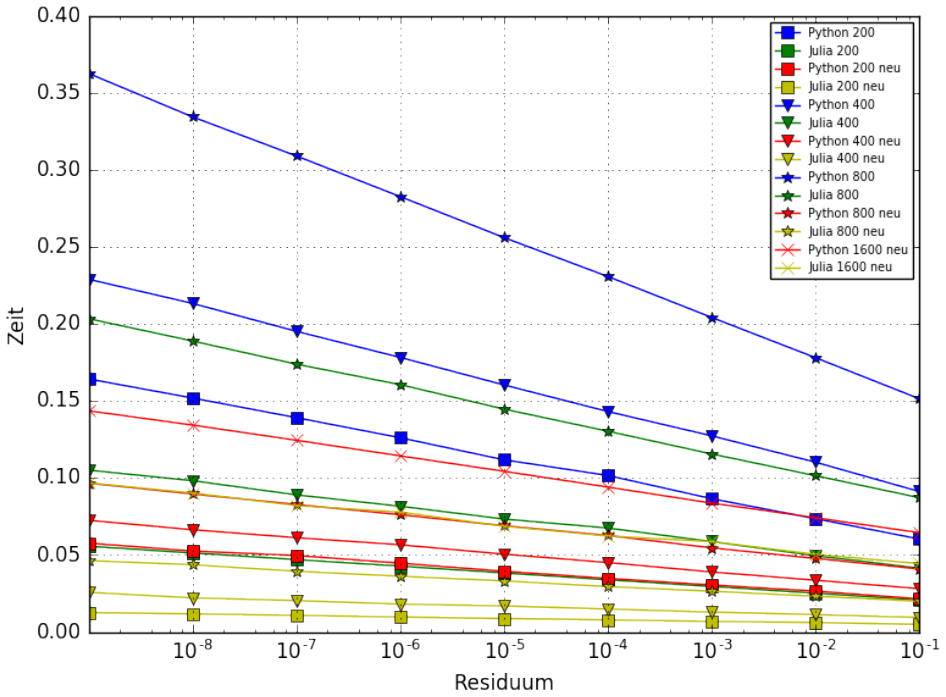

Abbildung 4: Vergleich des gewichteten Jacobi Verfahrens aufgrund der Matrixgroesse

Man kann allerdings erkennen, dass die einzelnen Graphen linearer sind als beim SOR Verfahren. Zudem fällt auf, dass mithilfe der Fourier Mode das gewichtete Jacobi Verfahren schneller ist als das SOR Verfahren.

9 Zwischenfazit

Anhand der verschiedene Tests die ich durchgeführt habe, konnte ich erkennen, dass Julia eine bessere Ausführungsgeschwindigkeit hat als Python. Die Vortest wurden somit bestätigt. Zudem konnte ich erkennen, dass bei einem zufälligem x_0 das SOR Verfahren am besten geeignet ist von den hier Vorgestellten. Man konnte beobachten das bei kleinerem Residuum auch die Ausführungszeit langsamer war. Zudem konnte ich bestätigen, dass bei größeren Matrizen der iterative Löser auch länger braucht. Wenn man allerdings den x_0 Vektor intelligent verändert kann man auch andere Verfahren verschnellern. Bei der Implementierung von Julia ist mir aufgefallen, dass es sehr einfach ist sich in Julia einzuarbeiten. Die Dokumentation ist im Moment noch übersichtlich und alle mathematischen Funktionen sind vorgegeben und man muss diese nicht importieren. Da Julia aber noch eine junge Sprache ist hat diese auch Nachteile. Beispielsweise musste ich oft Umwege über andere Funktionen gehen. Die Implementierung in Python dagegen war teilweise schwieriger, da die Doku sehr umfassend ist und man sich gut mit den Bibliotheken Numpy und Scipy auskennen muss, um geeignete Funktionen zu finden. Auf der anderen Seite hat Python eine sehr viel größere Community und man findet für fast jede Problemstellung schnell eine Lösung. Der nächste Schritt wird sein diese iterativen Löser matrixfrei in Julia und Python zu implementieren. Zudem werde ich mich mit der Parallelisierung von Algorithmen in Julia beschäftigen, da dieses leichter sein soll als in Python.

10 Matrixfreie Implementierung

10.1 Warum Matrixfreie Implementierung?

Um unser Ziel, die Parallelisierung des Jacobi Algoritmus in Julia zu erreichen, ist es notwendig das Verfahren erst in einem Vorschritt zu implementieren. Dies ist eine Matrixfreie Variante, die auf der Unabhaengigkeit der einzelnen Verfahrensschritte beim Jacobialgoritmus basiert. Dies ist fuer die Parralelisierung sehr wichtig. Ein weiterer Vorteil ist, das fuer diese Vorgehen weniger Speicherplatz in Anspruch genommen werden muss.

10.2 Was bedeutet Matrixfreie Implementierung?

Bei der Matrixfreien Implementierung geht es darum, die grossen Matritzen so darzustellen, dass diese moeglichst wenig Speicherplatz brauchen. Dies ist moeglich da wir die Gleichung $A * x = b$ loesen wollen und A in unserem Fall die Matrix ist, die bei der Diskretisierung des Laplace-Operators mit Finiten Differenzen zweiter Ordnung mit Dirichlet-Randbedingungen in einer Dimension ensteht. Wichtig an dieser Matrix ist aber, dass diese nur Eintraege auf der Diagonalen und den beiden Nebendiagonalen hat und dass alle Eintraege auf einer Diagonalen gleich sind. Dies macht es mir moeglich die Matrix als Stencil abzuspeichern. Das bedeutet in diesem Fall das die Matrix folgendermassen abgespeichert wird:
matrix = [-((n+1)*(n+1)), 2*((n+1)*(n+1)), -((n+1)*(n+1))]
Diese Matrix bezeichne ich im weiteren mit a. Das bedeutet auch das ich den Algorithmus dem hingehend veraendern muss, sodass dieser nicht mehr auf die einzelnen Elemente der Matrix A zugreifen kann, sondern immer nur auf diese drei Elemente. Alle anderen Vektoren sollen sich aber nicht veraendern. Das Ergebnis soll auch das gleiche sein.

10.3 Jacobi matrixfrei

Um zur Matrixfreien Methode zu kommen muss man betrachten, was in jeder Zeile einer Iteration bei der Matrixmultiplikation vor sich geht. Bei der Multiplikation von Matrix mit einem Vektor wird fuer jeden Wert b_n im Ergebnisvektor die entsprechende Zeile in der Matrix mit dem Vektor verrechnet. Da in diesem Fall die Matrix sehr speziell ist und nur Nulleintr?ge hat au?er auf der Hauptdiagonalen und den beiden Nebendiagonalen. Um den Wert b_n im Ergebnisvektor zuberechnen sind also nur die drei Eintaege x_{n-1}, x_n und x_{n+1}wichtig. Diese werden mit dem entsprechenden Wert multipliziert.
Die Formel fuer den Jacobi Algorithmus ist folgende:

$$x_{k+1} = w(I - D^{-1}A)x_k + D^{-1}b \tag{16}$$

Diese kann man umstellen damit erleichter fuer das matrixfreie Rechnen wird:

$$x_{k+1} = (1-w)x_k + -wD^{-1} * (Ax_k - b) \tag{17}$$

$$\Rightarrow x_{k+1} = (1-w)x_k + \frac{w}{a_{ii}}(b - \sum_{j \neq i} a_{ij}x_{k,j}) \tag{18}$$

Wenn man jetzt daran denkt, was Matrixfreies implementieren bedeutet sieht man, das man die die Gleichung fuer jedes $x_{k+1,i}$ umstellen kann. Man muss jedoch implementionsseitig bedenken, dass es eine Kopie des x_k geben muss, weil wir auch die alten Eintraege des x_k.
Die Implementierung sieht also folgendermassen aus: x[i]= (1-w)*xalt[i]+((w/matrix[1])*(b[i]-(matrix[0]*xalt[i-1]+matrix[2]*xalt[i+1])))
Ich teste die matrixfreie Variante mit der Fouriermode fuer k=40% bei einem Residuum=0.1. Ich waehle den Parameter w=2/3, da so ein optimales Ergebnis herauskommt. Ich teste dieses in Julia und Python und vergleiche dieses mit der Matriximplementierung und Der Matriximplementierung die mit dem LU Verfahren arbeitet. Ich betrachte dabei die Zeit die eine Iteration braucht im Verhaeltnis zur Matrixgroesse, welches bedeutet, wie lange der Computer braucht um ein besseres x_k zu berechnen. Die Betrachtung der Zeit pro Iteration ist notwendig, um das Verfahren mit dem SOR Verfahren zu vergleichen und um die Veraenderungen in der Matrixgroesse besser nachvollziehen zu koennen. Man koennte beim Jacobi Verfahren auch die absolute Zeit betrachten, da die Anzahl der Iterationen bei konstantem k konstant bleibt. Ich ueberpruefe das Verfahren anhand ihrer Iterationszahlen.
Man kann erkennen, dass das schnellste Verfahren die matrixfreie Implementierung in Julia ist und das langsamste Verfahren die Matrixfreie Implementierung in Python. Die Matrixbasierten Implementierungen liegen zwischen beiden matrixfreien Varianten. Es ist erstaunlich das bei grossen MAtritzen die LU Variante in Python die zweitschnellste wird, dies hatten wir in den vorherigen Tests nicht gesehen.

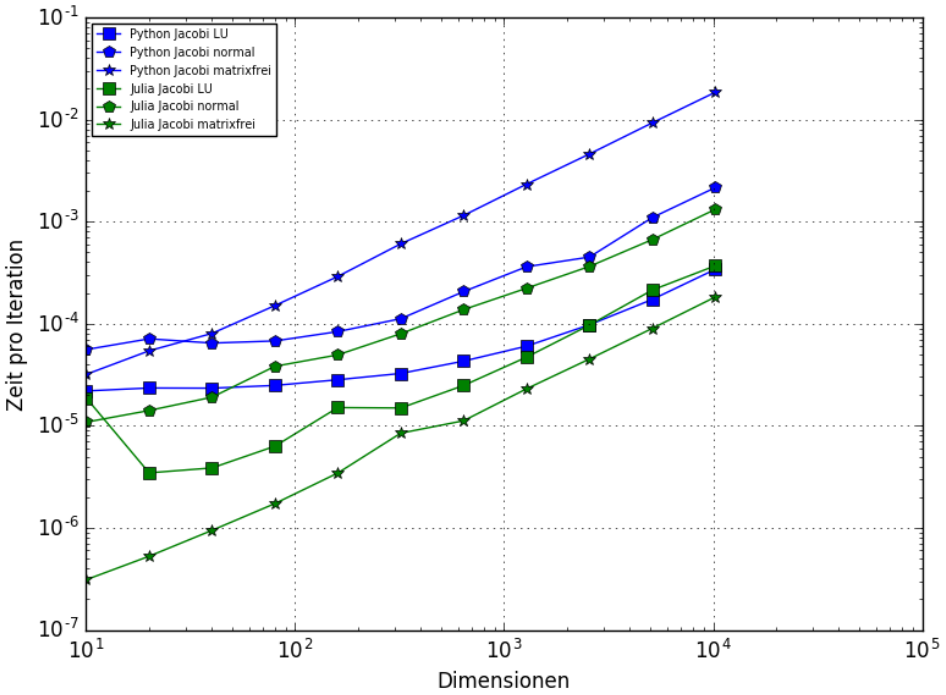

Abbildung 5: Vergleich Jacobi matrixfrei

10.4 SOR matrixfrei

Die Allgemeine Formel fuer das SOR Verfahren ist:

$$x_{k+1} = (1-w)x_k + \frac{w}{a_{ii}}(b - \sum_{j<i} a_{ij}x_{k+1j} - \sum_{j>i} a_{ij}x_{k+1j}) \qquad (19)$$

Aufgrund der Struktur des SOR Verfahren:
x[i]= (1-w)*x[i]+((w/matrix[1])*(b[i]-(matrix[0]*x[i-1]+matrix[2]*x[i+1])))
Ich teste das SOR Verfahren mithilfe der Fouriermode mit k=40% bei einem Residuum=0.01, damit ich dieses mit dem Jacobi Verfahren vergleichen kann. Auch hier teste ich in Julia und Python und benutze zum vergleichen die matrixbasierten Varianten. Ich muss hier den Parameter w=1 waehlen, da ich sonst keinen Trend in den Iterationszahlen feststellen kann. Damit ist das SOR Verfahren hier im Grunde das Gauss Seidel Verfahren ist. Trotzdem sind die Iterationszahlen hier nicht konstant deswegen ist hier vor allem die Zeit pro Iteration wichtig.
Man kann erkennen, dass das schnellste Verfahren die matrixfreie Implementierung in Julia ist und das langsamste Verfahren die Matrixfreie Implementierung in Python. Die Matrixbasierten Implementierungen liegen zwischen beiden matrixfreien Varianten.

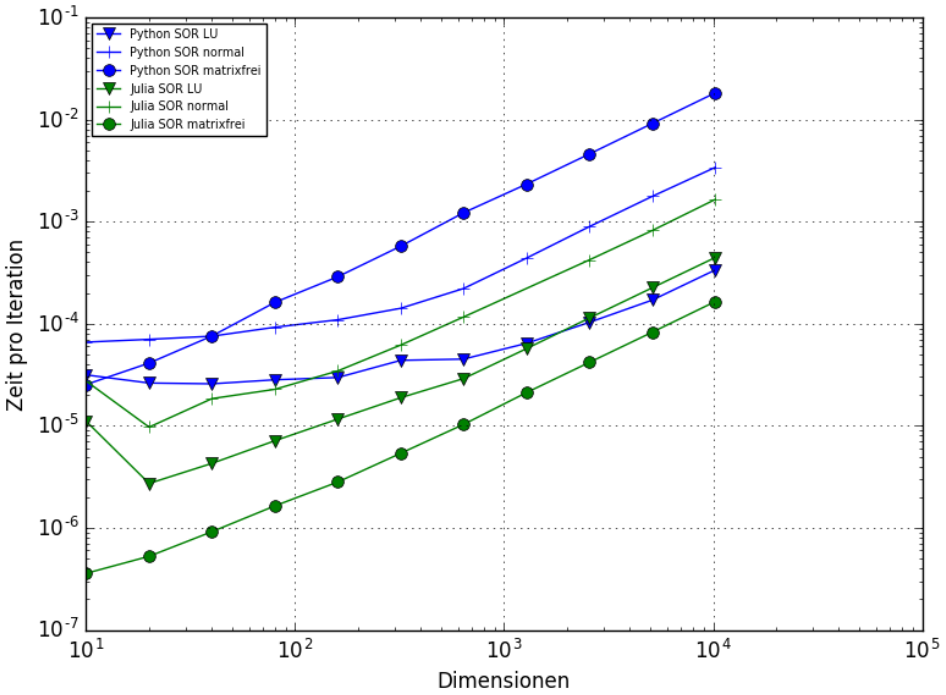

Abbildung 6: Vergleich SOR matrixfrei

10.5 Vergleich der Matrixfreien Varianten

Man kann erkennen, dass die Verfahren vor allem die Matrixbasierten Varianten in etwa gleich viel Zeit brauchen, was erstaunlich ist, da im Jacobi Verfahren ein Schritt mehr gemacht werden muss und ein Vektor kopiert werden muss. Dies kann aber nicht vermieden werden. Wenn mann die Gesamtzeiten betrachtet, kann man erkennen, das das Jacobi Verfahren an sich schneller ist, da dieses weniger Iterationen benoetigt.

11 Fazit

Schlussendlich kann ich sagen, dass die Matrixfreien Varianten von der Laufzeit her in Julia und in Python sehr unterschiedlich sind. In Julia ist diese Variante sehr schnell, wohingegen in Python diese Variante sehr langsam ist.