Wissenschafliches Rechnen mit Hilfe der Programmiersprache Julia

Damian Belz (DB)
Albert Piwonski (AP)
Rodrigo Rezende (RR)
Stephanie Tchoumi (ST)

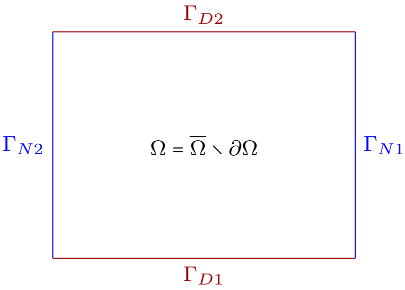
Wissenschaftliche Betreuer:
Mirsad Hadžiefendić
Marcus Christian Lehmann
TU Berlin

13. Februar 2019

Stand Zwischenvortrag

LAPLACE-Gleichung mit DIRICHLET- & NEUMANN-Bedingungen

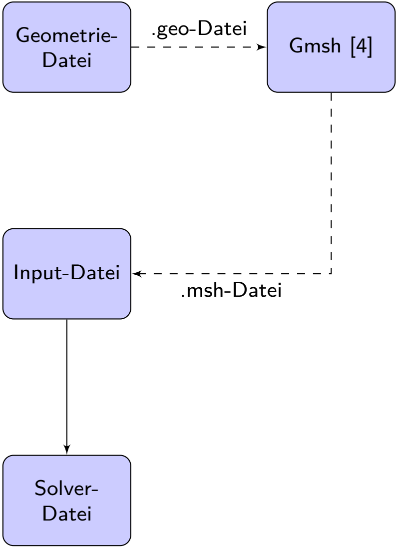
$$\text{(RWP)}\begin{cases} \Delta\phi = 0 & \text{in } \Omega \\ \partial\phi/\partial n = 0 & \text{auf } \Gamma_{N1} \cup \Gamma_{N2} \\ \phi = \phi_- & \text{auf } \Gamma_{D1} \\ \phi = \phi_+ & \text{auf } \Gamma_{D2} \end{cases}$$



FEM-Programmaufbau

- *händisches* Erstellen der Geometrie-Datei
- *händisches* Einlesen in Gmsh
- *händisches* Erstellen des Rechengitters
- *händischer* Export des Rechengitters
- *händische* Identifikation von $\partial\Omega$

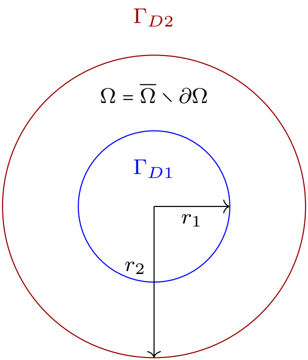
l: [1], r: [4].



LAPLACE-Gleichung mit DIRICHLET-Bedingungen

$$(\mathrm{RWP}) \begin{cases} \Delta\,\phi = 0 & \text{in } \Omega \\ \phi = \phi_- & \text{auf } \Gamma_{D1} \\ \phi = \phi_+ & \text{auf } \Gamma_{D2} \end{cases}$$

- *well-posed* (HADAMARD) [6]:
  - Existenz
  - Eindeutigkeit
  - Stabilität
- modelliert idealen Zylinderkondensator



1. Stand Zwischenvortrag

2. Modifiziertes Problem & Programmaufbau
   - Problemstellung
   - Programmaufbau
     - Rechengitter
     - Randbedingungen
     - Netzstruktur
     - Systemerzeugung
     - Lösung und Konvergenzstudie

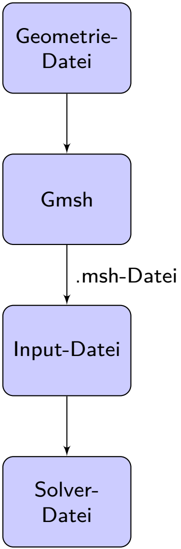
3. Besonderheiten der Implementierung
   - Typisierung
   - Paralleles Rechnen

4. Fazit & Ausblick

FEM-Programmaufbau: Erstellung des Rechengitters

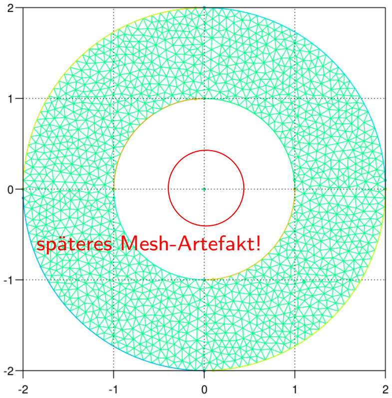
- Erstellen des Rechengitters durch .jl-Datei → Gmsh API [2]
- Definition von ***physical groups***
- automatisierte Identifikation von $\partial\Omega = \Gamma_{D1} \cup \Gamma_{D2}$

l: [1], r: [4].



Erstellung des Rechengitters: *Eigene* Implementierung

1. `calcPosition(posX, posY)`
   - berechnet Stützpunkt-Koordinaten
2. `createGeometry(posX, posY)`
   - erstellt Konstruktionspunkte, Kreisbögen, Kurven, Flächen, ***physical groups*** $\rightarrow \partial\Omega = \Gamma_{D1} \cup \Gamma_{D2}$
3. Erzeugung eines Rechengitters für Ansatzfunktionen 1. Ordnung



1. Stand Zwischenvortrag

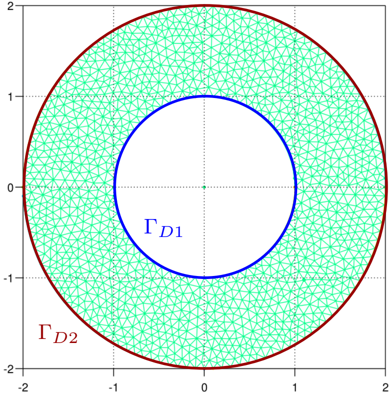
2. Modifiziertes Problem & Programmaufbau
   - Problemstellung
   - Programmaufbau
     - Rechengitter
     - Randbedingungen
     - Netzstruktur
     - Systemerzeugung
     - Lösung und Konvergenzstudie

3. Besonderheiten der Implementierung
   - Typisierung
   - Paralleles Rechnen

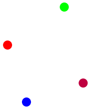
4. Fazit & Ausblick

Aufstellen der Randbedingungen

- diskrete Lösung : $A\backslash b = \phi_d$
- für DIRICHLET RB wird ϕ_d an den Randknoten definiert:
  1. ϕ_d[inner_B]$=u_{d1}$[inner_B]
  2. ϕ_d[outer_B]$=u_{d2}$[outer_B]



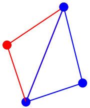
1. `inner_B=.getNodesForPhysicalGroup(1,2)`
2. `outer_B=.getNodesForPhysicalGroup(1,3)`



Koordinaten 2D

FLOAT	FLOAT
FLOAT	FLOAT
…	…

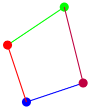
`.getNodes()`



Flächenelemente

INT	INT	⋯	
INT	INT	⋯	
…	…		

`.getElements(2,1)`



Randelemente

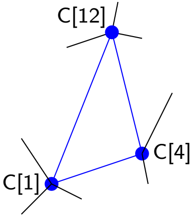
INT	INT
INT	INT
…	…

`.getElements(1,1)`

Funktionsargumente

Über `dim` und `tag` kann die Rückgabe der Funktionen spezifiziert werden.

Lokale Steifigkeitsmatrix

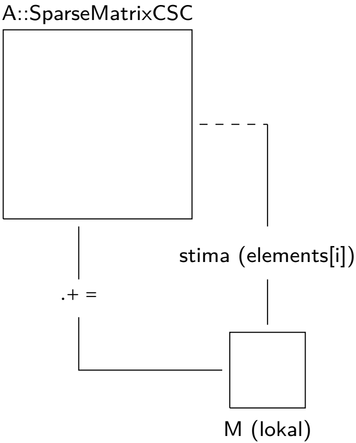


`elements[1]::element` = (I_1, I_4, I_{12})

`M::Array{Float}` $=$ `stima`$((I_1, I_4, I_{12}))$

Funktion: `stima(element)`

- gibt `M::Array{Float}` für ein Element zurück
- abhängig von Ansatzfunktion und Knotenanzahl
- für jede Elementart muss eine Methode der `stima()` Funktion implementiert werden

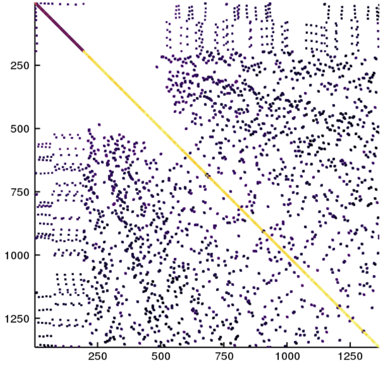


Assambly-Schleife

1. Initialisieren von A.
   n:= Knotenanzahl
2. Lokale Steifigkeitsmatrix für jedes Flächenelement.
   - Methodenauswahl durch Typ des Elements := Elementart
3. Übertragen von M in A[elements[i],elements[i]]

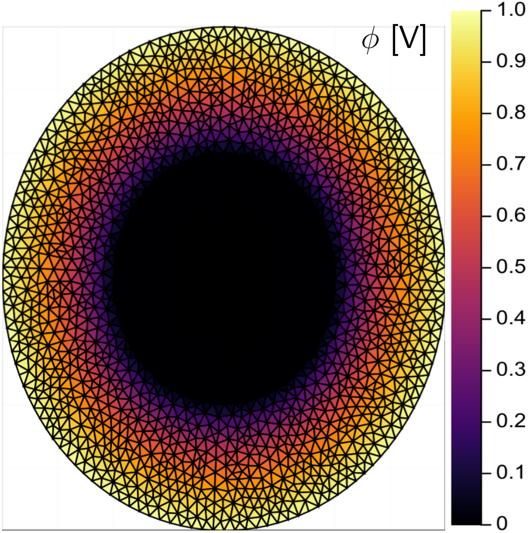
Beobachtung

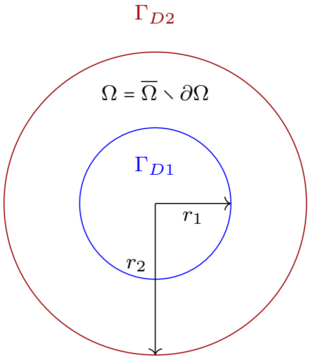
- DIRICHLET RB: $b = b - A \cdot \phi_d$
- $A \backslash b = \phi_d \rightarrow$ `SingularExeption`: $det(A) = 0$ bei 0-Eintrag in der Hauptdiagonalen
- Nicht alle Knoten sind Teil des Netzes: `A[FreeNodes ∩ Ele_Indicies]`



`spy(A)`

FEM-Lösung des RWP's

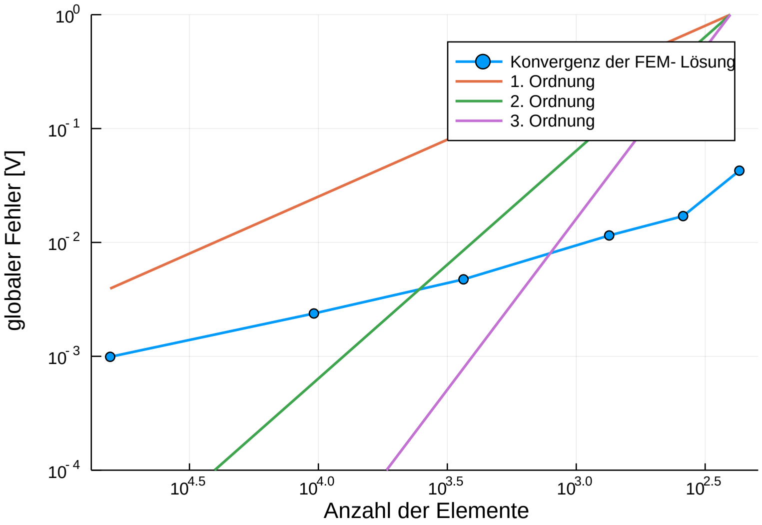




$$\text{(RWP)} \begin{cases} \Delta\phi = 0 & \text{in } \Omega \\ \phi = \phi_- & \text{auf } \Gamma_{D1} \\ \phi = \phi_+ & \text{auf } \Gamma_{D2} \end{cases}$$

Konvergenzstudie

- Global Nodal Error: $\delta \coloneqq \sqrt{\sum_{i=1}^{N}(\phi_{FEM}(x_i,y_i) - \phi(x_i,y_i))^2)}$



Konvergenzstudie: Kommentare

- Global Nodal Error: $\delta \coloneqq \sqrt{\sum_{i=1}^{N}(\phi_{FEM}(x_i, y_i) - \phi(x_i, y_i))^2)}$
- darin entspricht N der Anzahl der Elemente in dem jeweiligen Mesh
- ebenfalls möglich wäre die Betrachtung eines *effektiven Fehlers pro Element*: $\delta_{eff} \coloneqq \frac{1}{N}\sqrt{\sum_{i=1}^{N}(\phi_{FEM}(x_i, y_i) - \phi(x_i, y_i))^2)}$

Dynamische Typisierung

- Variablen sind nicht an Objekte eines bestimmten Typs gebunden
- die selbe Variable kann zu verschieden Zeitpunkten unterschiedliche, zueinander inkompatible Objekte referenzieren
- Typprüfungen während Laufzeit eines Programms
- Performance-Verlust durch Typprüfungen während Laufzeit
- Beispiel: `a = 4`

Statische Typisierung

- Typen werden während der Kompilierung geprüft
- Variablen werden zusammen mit ihrem Datentyp im Quellcode deklariiert
- Typen der Variablen sind zur Übersetzungszeit bekannt und sind nicht veränderbar
- höhere Performance, da Typen schon zur Laufzeit bekannt sind
- Beispiel: `a = 4::Int64`

Typisierung

[1]

Typisierung in Julia

- Vorteile von dynamisch und statisch getypten Sprachen
- parametrische Typdefinitionen möglich
- **struct** Definition möglich
- *bekannte* konkrete Typen, z.B.: `Int64, Float64`
- Methoden-Dispatch für abstrakte und konkrete Typen – Methoden werden über den Argumenttyp ausgewählt

```julia
#point2d = Array{Float64}
element = Tuple{Vararg{Int}} #Defined for various size of elements
# problem input mesh type
struct mesh
    coordinates::Array{Float64}  # coordinates of domainpoints {Float64}
    elements::Array{element}   # elements: indices point to coordinates {Int64}
    boundaries::Array{element}  #boundary: indices point to coordinates {Int64}
end

# creating mesh from gmsh data
msh = mesh( coordinates[:,1:2],
            [tuple(elements[i,:]...) for i in 1:size(elements,1) ],
            [tuple(boundary[k,:]...) for k in 1:size(boundary,1) ] )

```

- Wichtige Überlegungen für die Leistung:
  1. Ist das Problem *parallelisierbar*?
     - AMDAHLsches Gesetz
  2. *Datenbewegung*:
     - Die genutzte Zeit um die Daten zu einem Core zu bringen
  3. Die *Größe* des Problems
     - Bei größeren Probleme kann man größeren Zeitgewinn mit der Parallelisierung erreichen

- Um unseren Code parallelisieren zu können, haben wir die Zeit von verschiedenen Teilen des Programms gemessen (für Gitter mit 1365 Knotenpunkten): [1] [2]
  1. **Aufstellung der Steifigkeitsmatrix: 3.59 s**
  2. Lösung des Gleichungssystems: 1.95 s
  3. Ploten des Ergebnisses: 4.99 s

  Gesamte Simulationszeit: 25.92 s

[1] Hardware: Intel Core i7-4702MQ CPU @ 2.2 GHz,16 GB RAM

[2] unter Verwendung des `@time` Makro's nach zweitem Funktionsaufruf

- Es sind in Julia 3 verschiedene Niveaus von Parallelismus vorhanden [5]:

  1. **Julia Coroutines** (Green Threading)
  2. Multi-Threading (Testphase)
  3. **Multi-Core** oder **Distributed Processing**
  4. **Konkurrenz***

Julia Coroutines

- Erlaubt die Abwechselung von Rechnungen
- Koordinierung von der Reihefolge von Befehlen (*Tasks*)

Multi-Cores

- Standard Bibliothek von Julia
- Multi processing Umgebung

- Wichtiger Befehl: **@distributed**
- Dieses Makro enthält Eigenschaften der *Coroutines*, der *Multi-Core* und *Konkurrenz*

```
a = zeros(1000000000);
@distributed for i = 1:1000000000
    a[i] = i;
end
a
```

- Befehl um *parallele* for Schleifen zu programieren
- Iteration werden nicht in einer bestimten Reihefolge durchgeführt
- Umschreibung von Variablen werden lokal im Core gesehen

- Anwendung in dem Code:
  1. Parallele Eigenschaft
  2. Großes Problem
  3. Möglichst einfache Berechnung

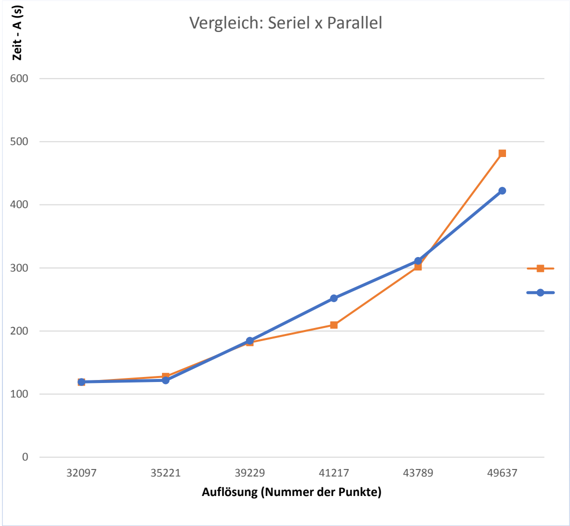
- Originaler Code

```
#A Assambly alternative
@time for i in 1:size(msh.elements,1)    #hier parallelisieren!
        A[collect(msh.elements[i]),collect(msh.elements[i])]+=stima(msh,msh.elements[i])
end
```

- Paralleler Code

```
#A Assambly alternative
#@parallel
@time @distributed for i in 1:size(msh.elements,1)    # @distributed macro to parallelise
        A[collect(msh.elements[i]),collect(msh.elements[i])]+=stima(msh,msh.elements[i])
end
```

- Vergleich von der Zeit:



- Vergleich von der Zeit:

Seriel

Mesh Size	Zeit - A (s)	Gesamte Zeit (s)
32097	118,938688	150,315614
35221	128,04001	161,83766
39229	182,043374	213,222984
41217	209,478964	240,516532
43789	302,002495	378,614366
49637	481,744498	569,334007

Parallel

Mesh Size	Zeit - A (s)	Gesamte Zeit (s)
32097	119,396473	151,773994
35221	121,806232	155,603882
39229	184,730799	215,910409
41217	251,858964	282,9
43789	311,288495	387,902476
49637	422,394498	509,980535

- Out of Memory ab 50.000 Gitterpunkten
- Andere Vorgehensweise wäre hier mit Parallelismus möglich

- Vergleich von der Zeit (unterschiedliche Simulationszeit bei Wiederholung):

```
Info    : 5201 vertices 10409 elements
Info    : Writing 'nice_mesh_v3.msh'...
Info    : Done writing 'nice_mesh_v3.msh'
  6.039364 seconds (7.60 M allocations: 4.507 GiB, 12.44% gc time)
  1.927692 seconds (4.60 M allocations: 226.599 MiB, 6.54% gc time)
  4.972373 seconds (5.15 M allocations: 256.448 MiB, 2.61% gc time)
 28.767019 seconds (62.22 M allocations: 7.366 GiB, 8.85% gc time)
```

```
Info    : 5201 vertices 10409 elements
Info    : Writing 'nice_mesh_v3.msh'...
Info    : Done writing 'nice_mesh_v3.msh'
  5.403396 seconds (7.36 M allocations: 4.494 GiB, 11.95% gc time)
  1.761284 seconds (4.38 M allocations: 214.800 MiB, 5.56% gc time)
  2.280496 seconds (4.32 M allocations: 215.104 MiB, 4.62% gc time)
 12.022765 seconds (20.15 M allocations: 5.317 GiB, 7.99% gc time)
```

Zusammenfassung & Ausblick

Zusammenfassung

- *komplizierteres* RWP ✓
- Kopplung von Julia und Gmsh über Gmsh API ✓
- Definition eines Element- & Mesh-Typs in Julia ✓
- paralleles Berechnen der Steifigkeitsmatrix ✓
- Versionierungssystem Git: ✓
  → Git Repository

Ausblick

- Grad der Parallelisierung erhöhen
- Ansatzfunktionen als Polynome höherer Ordnung (> 1)
- 3D-Problemstellungen

Vielen Dank für Ihre Aufmerksamkeit!

Quellen I

[1] Bezanson, J. ; Edelman, A. ; Karpinski, S. ; Shah, V.: Julia: A Fresh Approach to Numerical Computing. In: SIAM Review 59 (2017), Nr. 1, 65-98. http://dx.doi.org/10.1137/141000671. – DOI 10.1137/141000671

[2] Christophe Geuzaine, Jean-François R.: Gmsh. http://gmsh.info/

[3] Daniele Boffi, Michel F. Franco Brezzi B. Franco Brezzi: Mixed Finite Element Methods and Applications. Springer, 2013

[4] Geuzaine, Christophe ; Remacle, Jean-François: Gmsh: A 3-D finite element mesh generator with built-in pre- and post-processing facilities. In: International Journal for Numerical Methods in Engineering 79 (2009), Nr. 11, S. 1309–1331. http://dx.doi.org/10.1002/nme.2579. – DOI 10.1002/nme.2579

[5] Language, The J.: Parallel Computing. https://docs.julialang.org/en/v1/manual/parallel-computing/#Parallel-Computing-1

[6] Schmidt, Kersten: Numerics of Partial Differential Equations. TU Berlin, 2015

[7] Sebastian Schöps, Herbert De G.: PASIROM - Summer School. TU Darmstadt, 2018

Potentialansatz

E-Statik: Differentialgleichung

- FARADAY'sches Induktionsgesetz im statischen Limit:

$$\mathbf{rot}\,\mathbf{E} \equiv \mathbf{0}$$

- elektrisches Feld $\mathbf{E}$ als Grandientenfeld vom Skalarpotential ϕ:

$$\mathbf{E} = -\mathbf{grad}\,\phi$$

- lin., homog., isot., ladungsf. Material $\Rightarrow$ LAPLACE-Gleichung:

$$\mathrm{div}\,\mathbf{grad}\,\phi = \Delta\,\phi = 0$$

Finite Elemente Methode (FEM): Hauptideen [3]

- Diskretisierung von Ω in *einfache* Sub-Domänen (Dreiecke)
- Approximation des Funktionenraums der Lösung ϕ [7]:

$$\phi \in H^1(\Omega) : (\text{Sobolev-Raum 2-fach integrierbarer Funktionen}),$$

$$H^1(\Omega) \coloneqq \left\{ u \in L_2(\Omega) \middle| \, \mathbf{grad}\,(u) \in L_2(\Omega)^3 \right\}$$

- (Knoten)-Ansatzfunktionen mit **finiter** Basis & *Support*:
  oft Polynome niedriger Ordnung, müssen Differentiationsklasse von ϕ genügen
-

$$\text{LGS} \Leftarrow \begin{cases} \text{variationelle Formulierung des RWP} \\ \text{Ritz-Galerkin-Methode} \\ \ldots \end{cases}$$

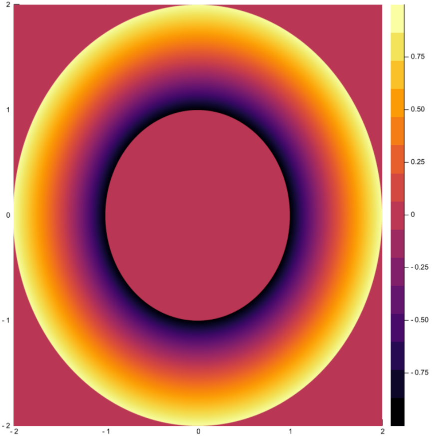


Abbildung: Analytische Lösung ϕ

Analytische Lösung ϕ

- $\phi(x,y) \coloneqq A_1 + A_2 \ln(\varrho)$
- $A_1 \coloneqq \frac{\phi_+ - \phi_+}{\ln(\frac{r_2}{r_1})}$
- $A_2 \coloneqq \phi_- - A_1 \ln(r_1)$