TECHNISCHE UNIVERSITÄT CHEMNITZ

Arnd Meyer

Programmbeschreibung SPC-PM3-AdH-XX Teil 1

CSC/14-01

Chemnitz Scientific Computing Preprints

Chemnitz Scientific Computing Preprints – ISSN 1864-0087

Impressum:

Chemnitz Scientific Computing Preprints — ISSN 1864-0087
(1995–2005: Preprintreihe des Chemnitzer SFB393)

Herausgeber:
Professuren für
Numerische und Angewandte Mathematik
an der Fakultät für Mathematik
der Technischen Universität Chemnitz

Postanschrift:
TU Chemnitz, Fakultät für Mathematik
09107 Chemnitz
Sitz:
Reichenhainer Str. 41, 09126 Chemnitz

http://www.tu-chemnitz.de/mathematik/csc/

TECHNISCHE UNIVERSITÄT CHEMNITZ

Chemnitz Scientific Computing
Preprints

Arnd Meyer

Programmbeschreibung
SPC-PM3-AdH-XX
Teil 1

CSC/14-01

Zusammenfassung

Beschreibung der Finite Elemente Software-Familie

SPC – PM3 – AdH – XX

für: **S**cientific **P**arallel **C**omputing -
Programm–**M**odul **3D** – **ad**aptiv – **H**exaederelemente.

Für XX stehen die einzelnen Spezialvarianten, die in Teil 2 detailliert geschildert werden.

Stand: Ende 2013

CSC/14-01 ISSN 1864-0087 März 2014

Inhaltsverzeichnis

1 Allgemeine Vorbemerkungen **1**

2 Grundstruktur **2**

3 Datenstrukturen **3**
- 3.1 Gesamtüberblick . . . 3
- 3.2 Einzeldarstellung der Strukturen . . . 4
  - 3.2.1 Datenstruktur Knoten: `X(MvonX, *)` . . . 5
  - 3.2.2 Datenstruktur `Edge(MEdge, *)` . . . 5
  - 3.2.3 Datenstruktur `Face(MFace, *)` . . . 6
  - 3.2.4 Datenstruktur `Solid(MSol, *)` . . . 7

4 Gesamtablauf **8**

5 Parallelisierung **10**

6 Die Grundvariante A3D_Original und ihre Bibliotheken **11**
- 6.1 Netzeingabe- und Start-Routinen (0) . . . 11
- 6.2 Netzverfeinerung (1) . . . 12
- 6.3 Berechnung der Elementmatrix (2) . . . 12
- 6.4 Löser (3) . . . 15
- 6.5 Fehlerschätzer (4) und Post-Processing . . . 17

Author's addresses:

Arnd Meyer
TU Chemnitz, Fakultät für Mathematik
09107 Chemnitz, Germany

unter Mitwirkung von:
Janine Glänzel, Martina Weise, Dr. Roman Unger, Dr. Matthias Pester,
Michael Weise

`http://www.tu-chemnitz.de/mathematik/`

Some titles in this CSC and the former SFB393 preprint series:

11-01 P. Benner, M.-S. Hossain, T. Stykel. Low-rank iterative methods of periodic projected Lyapunov equations and their application in model reduction of periodic descriptor systems. February 2011.

11-02 G. Of, G. J. Rodin, O. Steinbach, M. Taus. Coupling Methods for Interior Penalty Discontinuous Galerkin Finite Element Methods and Boundary Element Methods. September 2011.

12-01 J. Rückert, A. Meyer. Kirchhhoff Plates and Large Deformation. April 2012.

12-02 A. Meyer. The Koiter shell equation in a coordinate free description. February 2012.

12-03 M. Balg, A. Meyer. Fast simulation of (nearly) incompressible nonlinear elastic material at large strain via adaptive mixed FEM. July 2012.

13-01 A. Meyer. The Koiter shell equation in a coordinate free description – extended. September 2013.

13-02 R. Schneider. With a new refinement paradigm towards anisotropic adaptive FEM on triangular meshes. September 2013.

13-03 A. Meyer. The linear Naghdi shell equation in a coordinate free description. November 2013.

The complete list of CSC and SFB393 preprints is available via
`http://www.tu-chemnitz.de/mathematik/csc/`

[15] M. Balg, A. Meyer, *Fast simulation of (nearly) incompressible nonlinear elastic material at large strain via adaptive mixed FEM*, Preprint CSC12-03 TUChemnitz (2012).

[16] M. Balg, J. Lang, A. Meyer, G. Rünger, *Array–based reduction operations for a parallel adaptive FEM*, R.Keller et al(Eds.): Facing the Multicore-Challenge III 2012 Lecture Notes in Comp.Sci. Vol.7686, pp.25-36,(2013).

1 Allgemeine Vorbemerkungen

In der vorliegenden Programmdokumentation soll ausführlich das Spezialwissen dargelegt werden, das bei der Implementierung der **SPC-PM3-AdH**–Software–Familie der TU Chemnitz (Fakultät für Mathematik) eingeflossen ist und für zukünftige Fortentwicklungen nötig ist. Die **SPC**-Software dient der näherungsweisen Lösung von Aufgabenstellungen, die mit ausgewählten partiellen Differentialgleichungen beschrieben sind. Hierzu kommt die adaptive Finite Elemente Methode zum Einsatz, wobei grundsätzlich auf besonders hohe Effizienz bezüglich Rechenzeit und Speicherbedarf Wert gelegt wurde.

- Die exakte Lösung des Differentialgleichungsproblems wird mit Hilfe einer Folge immer feinerer Netze, die sich mittels Fehlerindikatoren besonders an die Lösung anpassen, durch eine sehr genaue FE-Lösung approximiert.
- Die Netzverfeinerung basiert auf Teilung von Elementen, deshalb entsteht eine hierarchische Knotenanordnung.
- Die Knotenhierarchie wird besonders ausgenutzt, um in der Folge der einzelnen FE–Gleichungssysteme stets schnelle Löser parat zu haben. Hierzu wird wesentlich der MDS-BPX-Preconditioner [1, 2] eingebettet in **PCG** benutzt.
- Da die Steifigkeitsmatrizen in der adaptiven Abfolge nur sehr kurz benötigt würden, aber relativ aufwändig aufgebaut werden, wird auf eine Assemblierung der einzelnen Steifigkeitsmatrizen ganz verzichtet. Statt dessen enthält die Element–Datenstruktur die Element–Steifigkeitsmatrix (eventuell auch mehrere), die für lineare Probleme nur bei „neuen“ Elementen (aus der letzten Verfeinerung) neu berechnet werden müssen.

Aus den bisherigen Ausführungen resultieren folgende grundlegenden Einschränkungen bei der Anwendbarkeit dieser Software-Familie:

1. Es werden ausschließlich symmetrische Problemstellungen betrachtet (dies sind in der Anwendung: Reaktions-Diffusions-Probleme als skalare Gleichungen bzw. Deformationsprobleme der Festkörpermechanik inKl. Temperatur–Elastitzität, nichtlineares Material bzw. gemischte Ansätze für (fast) Inkompressibilität u. ä.).
2. Die „matrixfreie“ Lösertechnik verbietet Vorkonditionierer, die die Matrixelemente direkt benötigen (z. B. die meisten Glätter bei Multigrid).
3. Eine vernünftige Simulation erfordert die Startvernetzung als „Grobnetz“, das von einem einzulesenden File stammt. Dies sollte Knotenzahlen bis wenige Hundert nicht übersteigen. Die Knotenzahlen in der adaptiven Simulation wachsen dann durchaus auf etwa 10^6 an, was bei hoher Genauigkeit eine

Verarbeitungszeit meist im Sekundenbereich (bis wenige Minuten) möglich macht.

2 Grundstruktur

Die **SPC-PM3-AdH**-Software-Familie besteht aus mehreren Problem–Directories, die die jeweilige Gesamtimplementierung für eine spezielle Problemvariante enthalten. Im einzelnen sind dies:

A3D_Original: die Startimplementierung für lineare Reaktions-Diffusions-Probleme und lineare Elastizität. Diese Variante enthält alle grundlegenden Moduln im Original, die bei den anderen Varianten ungeändert genutzt werden, hauptsächlich das gesamte "mesh handling" (Grobnetz-Eingabe, Initialisierung der Datenstrukturen, Netz-Verfeinerungen). Alle anderen Varianten nutzen hier Links auf diese Moduln.

A3D-LD: die Erweiterung von **A3D_Original** für nichtlineares Materialverhalten bei Deformationsproblemen ("**L**arge **D**eformations"). Insbesondere eine Newton–Iterationsschleife und die zugehörige Steuerung (im Hauptprogramm) sowie neue Moduln für die Materialgesetze kommen hinzu.

A3D-ThEl: Kopplung von Wärmeleitgleichung mit elastischer Deformation (Thermoelastizität).

A3D-inkLE: linear-elastische Deformationsprobleme mit inkompressiblen Materialien (als gemischte FEM behandelt).

A3D-inkLD: nichtlineare Materialien im Deformationsproblem mit (ev. teilweisen) Inkompressibilitäten, also ebenfalls gemischte FEM nach Newton-Linearisierung.

A3D-TraIso: lineare Deformationsprobleme mit transversal isotropem Materialverhalten. Besonders die Bereitstellung einer Funktion zur Beschreibung der örtlich variablen „Faser-Richtung" macht hier die Besonderheit aus.

A3D-TEAni: Vereinigung von Thermo-Elastizität mit Transversal-Isotropie.

Jede dieser Varianten ist im jeweiligen Directory **A3D-XX** zu finden, im wesentlichen mit dem Hauptprogramm zur gesamten Steuerung der Simulation. Dieses Hauptprogramm enthält neben den Aufrufen der wesentlichen Aktionen (Initialisierung, Verfeinerung, Berechnung der FE-Matrizen, Löser, Grafik, Post-Processing) hauptsächlich Speicherplatz-Management für Hilfsfelder, die diese Aktionen benötigen. Zusätzlich finden sich hier das Makefile, u. ä.

Die 4 Hauptaktionen basieren auf Moduln in den 4 Subdirectories

Literatur

[1] J. Bramble, J. Pasciak, and J. Xu, *Parallel multilevel preconditioners*, Math. Comput., 55:1-22, (1990).

[2] S. Zhang. *Multilevel schwarz methods* Numer. Math. 63:512-539,(1992).

[3] A. Meyer, *Projected PCGM for Handling Hanging Nodes in Adaptive Finite Element Procedures*, Preprint SFB393 99-25 TUChemnitz (1999).

[4] S. Beuchler, A. Meyer, M. Pester, *SPC-PM3AdH v 1.0 - Programmer's Manual*, Preprint SFB393 01-18 TUChemnitz (2001), (revised version, August 2003).

[5] A. Meyer, *Projection Techniques embedded in the PCGM for Handling Hanging Nodes and Boundary Restrictions* in: Engeneering Computational Technology B.H.V.Topping and Z.Bittnar, (Eds.), Saxe-Coburg Publ., Stirling, Scotland, 147-165, (2002).

[6] M. Pester, *Bibliotheken zur Entwicklung paralleler Algorithmen - Basisroutinen für Kommunikation und Grafik*, Preprint SFB393 02-01 TUChemnitz (2002).

[7] M. Pester, *Visualization Tools for 2D and 3D Finite Element Programs - Users Manual*, Preprint SFB393 02-02 TUChemnitz (2002).

[8] A. Meyer, R. Unger, *Projection methods for contact problems in elasticity*, Preprint SFB393 04-04 TUChemnitz (2004).

[9] R. Unger, *Unterraum-CG-Techniken zur Bearbeitung von Kontaktproblemen*, Dissertation TUChemnitz (2006).

[10] A. Meyer *Grundgleichungen und adaptive Finite-Elemente-Simulation bei „Großen Deformationen"*, Preprint CSC07-02 TUChemnitz (2007).

[11] R. Unger, *Obstacle Description with Radial Basis Functions for Contact Problems in Elasticity*, Preprint CSC09-01 TUChemnitz (2009).

[12] J. Glänzel, *Kurzvorstellung der 3D-FEM Software SPC-PM3AdH-XX*, Preprint CSC09-03 TUChemnitz (2009).

[13] M. Balg, A. Meyer *Numerische Simulation nahezu inkompressibler Materialien unter Verwendung von adaptiver, gemischter FEM*, Preprint CSC10-02 TUChemnitz (2010).

[14] M. Weise, A. Meyer, *Grundgleichungen für transversal isotropes Materialverhalten*, Preprint CSC10-03 TUChemnitz (2010).

wobei η_T der elementbasierte berechenbare Fehleranteil des Elements T ist. Im hier zu betrachtenden Fall wird für η_T^2 eine Näherung der Größe

$$(1/\lambda_T)(h^2||r_T||_T^2 \; + \sum_{F\subset\partial T} h||r_F||_F^2)$$

berechnet. Hierbei ist $||r_T||_T$ die L_2–Norm des Elementresiduums:

$$r_T = div\,\sigma(u_h) + f,$$

($\sigma(u_h)$ ist der Spannungsverlauf bei 3 Freiheitsgraden, bei 1 Freiheitsgrad sei im weiteren $\sigma(u_h) = A \cdot \nabla u_h$, die konormale Ableitung. Analog ist f der Lastvektor bzw. die Quellfunktion q bei einem Freiheitsgrad).

Der wichtigste Bestandteil ist der Flächensprungterm:

$$r_F = \begin{cases} \frac{1}{2}(\sigma(u_h)|_{T_1} - \sigma(u_h)|_{T_2}) \cdot \mathfrak{n} & \text{bei inneren Faces } F = T_1 \cap T_2 \\ \sigma(u_h)|_T \cdot \mathfrak{n} - g & \text{bei Randfaces } F \text{ am Neumann-Rand} \\ 0 & \text{bei Randfaces } F \text{ am Dirichlet-Rand} \end{cases}$$

(analog ist r_F ein Vektor bei 3 Freiheitsgraden, sonst skalar). Die Skalierungsgröße h steht für die Ausdehnung des Elements orthogonal zu F und wird als $measT/measF$ angesetzt. Im elementorientierten Teil des Fehlerschätzers, wo obige Summe über alle Faces gebildet wird, benutzen wir λ_T als typische Materialgröße des Elements, also $\lambda_T = 2\mu$ bei linearer Elastizität und $\lambda_T = \min(\alpha_1, \alpha_2, \alpha_3)$ bei einem Freiheitsgrad.

Mit dieser Vorbereitung ergibt die Berechnung der L_2-Norm $||r_F||_F^2$ über eine 1-Punkt-Integration:

$$h||r_F||_F^2 = \frac{measT}{measF} \cdot measF \cdot |r_{F|_M}|^2$$

mit $r_{F|_M}$ am Mittelpunkt des Faces. Wegen $|\sigma \cdot \mathfrak{n}|^2 = \frac{1}{||c||^2}|\sigma \cdot c|^2$ wird eindgültig folgende Formel für η_T^2 benutzt:

$$\eta_T^2 = |\sigma \cdot c|_{|_M}^2 \cdot \frac{measT}{measF^2},$$

weil das Kreuzprodukt c zur Bildung der Normalenrichtung gerade die Norm $||c|| = measF$ hat. Es wird also der Sprungterm durch Addition von

$$\frac{\sigma \cdot c|_M}{measF}$$

über alle Faces und über alle Elemente gebildet (c zeigt stets nach außen) und danach das Normquadrat berechnet.

`aNetzQ:`	Initialisierung/Startvernetzung
`aFineQ:`	Netzverfeinerung
`aElemQ:`	Berechung der Element-Matrizen und rechten Seiten des Gleichungssystems mit problemabhängigen Teilroutinen sowie wesentliche Routinen des FE–Post–Processings. Bestandteile des Post–Processings sind die Berechnung von Ableitungen bzw. Spannungen und deren Verwendung in den Fehlerschätzern.
`aSolvQ:`	Vorbereitung und Lösung der Gleichungssysteme mit BPX–Vorkonditionierer

In den meisten Varianten sind die Routinen in `aNetzQ` und `aFineQ` keine eigenständigen Files, sondern symbolische Links auf die gleichnamigen Files in der Grundvariante **A3D_Original**. Die wesentlichen Unterschiede der Programmvarianten dokumentieren sich also in den Subdirectories `aElemQ` und `aSolvQ`, die sich von Variante zu Variante sicher unterscheiden (meist gleiche UP-Namen mit angepasstem anderen Inhalt).

Im Hauptprogramm findet sich noch der Aufruf einer Grafikroutine mit der die aktuell berechneten Näherungsfunktionen und eventuell bereitgestellte abgeleitete Größen visualisiert werden können. Dieses Paket ist beim Linken bereitgestellt (Makefile) und nicht in den lokalen Bibliotheken zu finden. Eine ausführliche Beschreibung dieser Visualisierungssoftware (Autor: M. Pester) findet sich in [6, 7].

Desweiteren nutzen alle Unterprogramme konsequent einige Grundroutinen, die ebenfalls in einer Bibliothek beim Link-Prozess bereitgestellt werden. Diese Grundroutinen sind etwa die Linear-Algebra-Basisroutinen ("saxpy" o. ä.), die eventuell durch entsprechende **BLAS**-Routinen ersetzt sein können (abhängig vom Link-Prozess). Genaueres siehe [6].

3 Datenstrukturen

3.1 Gesamtüberblick

Die angestrebte hohe Effizienz der FE-Berechnungsmoduln wird nicht nur durch die adaptive Vernetzung erreicht, vielmehr kann man durch Ignorieren von Hardware-basierten Restriktionen drastische Verluste erzeugen. Deshalb sind alle Datenstrukturen als eindimensionale "array of structure" ausgelegt. Die meisten Grundaktionen sind vom Typ einer Schleife:
"`for` each Element `do` ..." oder "`for` each Edge `do` ..." o. ä.
Durch die eindimensionale Anordnung aller Elemente, Faces, Edges etc. ergibt sich eine weitgehend cacheoptimale Verarbeitung. Die grundlegenden Datenstrukturen wurden in [4] beschrieben. Eventuell haben sich seither geringe Modifi-

kationen notwendig gemacht, weshalb hier noch einmal ein Überblick über die Datenstrukturen gegeben wird.

Wir unterscheiden:

a) feste Datenfelder, die beim Einlesen des Files *xx.std* festgelegt werden für:

- Geometriedefinitionen der Faces: `GEOMETRY(DIMGEOM,nGeo)` – feste Anzahl `nGeo` von Face-Geometrien, z. Zt. `DIMGEOM=9`
- Materialdaten von Materialbereichen `RMate(NMaxIn+2,NMat)` – feste Anzahl von `NMat` Materialbereichen mit 2 INTEGER- und maximal `NMaxIn` REAL-Informationen.
- Randbedingungsinformationen als Dirichlet- und Neumann-Daten: `DIR(1+MDirNeu,nDir), Neum(1+MDirNeu,nNeum)` – feste Anzahl von `nDir` bzw. `nNeum` verschiedenen Randbedingungsinformationen, die den eingelesenen Faces des Grobnetzes zugeordnet sind. Mit der Verfeinerung vererben sich die Zeiger (Feldindices k für `DIR(1,k)`) bzw. `Neum(1,k)` von geteilten Faces auf die Tochterfaces.

b) variabel anwachsende Datenfelder, die mit der Verfeinerung wachsen und das aktuelle Netz repräsentieren:

`X(MvonX, Nn)`	Knoten / Lösungswerte in den Knoten
`Edge(MEdge, NE)`	Kanten–Infos
`Face(MFace, NF)`	Face–Infos
`Solid(MSol, NVol)`	Solid–(Volume)–Infos, über die Hexaederelemente

Die 4 Anzahlen `Nn` (Anzahl der Knoten), `NE` (Anzahl der Edges), `NF` (Anzahl der Faces) und `NVol` (Anzahl der Solids/Elemente) wachsen im Verlauf der adaptiven Netzverfeinerung kontinuierlich an. Deshalb wird die Startadresse dieser Felder am Anfang im großen Workspace-Feld auf sinnvolle Positionen gesetzt (Modul *Set_Pointers*).

c) weitere variabel anwachsende Datenstrukturen, die bei der Implementierung des Lösers helfen, vor allem eine Informationsliste `Hier(NHier)` mit den Vater-Sohn-Informationen über Knoten zur Implementierung des **BPX**-Vorkonditionierers. Dies ist in [4] geschildert.

3.2 Einzeldarstellung der Strukturen

Jedes Einzelelement der array-of-structure obiger Datenstrukturen, also `X(1,i)`, `Edge(1,i)`, `Face(1,i)`, `Solid(1,i)` für das jeweils i-te Element dieser Strukturen stellt eine Informationsstruktur aus INTEGER– und REAL–Werten (ev. auch REAL*8–Werten) dar:

Diese Matrix K wäre in jedem Fall singulär, es wird aber das projizierte Gleichungssystem

$$P^T K \underline{u} = P^T \underline{b}$$

gelöst, wobei einige Komponenten in $\underline{u}$ schon festgelegt sind (z. B. inhomogene Dirichlet-Randbedingungen). Die Matrix $P^T K P$ ist dann regulär und die wirkende Systemmatrix im Löser, vgl. [3, 5].

Der Projektor P hat mehrere Funktionen. Die wichtigsten sind:

- Erfüllung der Dirichlet-Randbedingungen
- zulässige Funktionswerte bei "hanging nodes"
- Erfüllung der „Rutsch-Randbedingungen"

usw.

Für den Fall, dass die rechte Seite orthogonal zum Nullraum von K steht, kann ohne jede Schwierigkeit auch ein singuläres Gleichungssystem gelöst werden. In diesem Fall muss natürlich der Grobgitterlöser abgeschaltet sein (`Ivar`=4, s. 6.1).

6.5 Fehlerschätzer (4) und Post-Processing

Nach Berechnung der aktuellen FE-Lösung kann die grafische Darstellung dieser erfolgen (*CALL s3Dgraph*). Dies ist in der neueren Version eine elementorientierte Ausgabe aller Lösungskomponenten (als stetige Felder) und von abgeleiteten Größen (Gradient oder Spannungen, unstetig über die Elementgrenzen). Die Aufbereitung der Lösung (aus `X`) und die Berechnung von abgeleiteten Größen erfolgt von der Vorbereitungsroutine *GetStress*, die eine (elementorientierte) Ausgabe dieser Größen in die Struktur `Solid(,.,)` vornimmt. Danach wird die eigentliche Funktionalität des Schrittes (4) folgen.

Im Falle von Kontaktsimulationen (bei 3 oder mehr Freiheitsgraden) wird vorher die Berechnung des Kontaktdruckes auf das Hindernis vorgenommen (Routine *ContTract*). Hiermit kann man Knoten detektieren, die zu Unrecht als Kontaktknoten markiert wurden (negativer Kontaktdruck). Diese werden im folgenden Durchlauf wieder frei sein, so dass sich die richtige Kontaktzone in der adaptiven Schleife „nebenbei" einspielt.

Der eigentliche Fehlerschätzer wird durch die Routine *EfromU* vorbereitet und in *ErrorEst* zur Markierung der Solids ausgewertet.

Die grundlegende Formel für den benutzten Residuen-Fehlerschätzer ist die Abschätzung

$$a(\underline{u} - \underline{u}_h, \underline{u} - \underline{u}_h) \;\leq\; \sum_{\forall T} \eta_T^2,$$

Elementmatrizen des Grobnetzes einmalig aufgebaut, sofort Cholesky-zerlegt und bei jedem Verfeinerungsschritt unverändert als „Grobgitter-Löser“ benutzt.

Dies ist nur dann möglich, wenn durch Einbau von Dirichlet-Randbedingungen in C_0 eine nichtsinguläre Grobgittermatrix entsteht. Wenn am Start keine Dirichlet-Randbedingungen existieren, also C_0 singulär entsteht, muss ohne Grobgitterlöser weitergearbeitet werden.

Vor dem Aufruf von PPCGM steht im Hauptprogramm noch die wichtige Routine *stave* (Quelle: *aSolvQ/stave.f*), die der Festlegung des Startvektors für die PPCG-Iteration dient.

Grundsätzlich werden zuerst alle Funktionswerte aus dem vorhergehenden Netz (und interpoliert auf das aktuelle) von der Datenstruktur `X` auf den Startvektor übertragen. (Dies zieht den sogenannten „Kaskade“-Effekt nach sich: der Startfehler enthält nur „hochfrequente“ Anteile.)

Zum Zweiten sind die eventuell vorgegebenen Dirichlet-Randbedingungen korrekt auf den Startvektor zu übertragen, weil im Löser an diesen Stellen ausschließlich Null-Korrekturen erfolgen.

Für den Fall von Kontaktsimulationen bei 3 Freiheitsgraden müssen nunmehr einige Komponenten des Startvektors (gehörend zu Randknoten) auf ihre Zulässigkeit überprüft und eventuell entsprechend korrigiert werden.

Für den Randknoten X mit dem vorläufigen Lösungsvektor $U(X)$ heißt dies:
$x = X + U(X)$ hat in einer zulässigen Menge $\mathcal{Z}$ zu sein.

Dies sind z. B. alle Punkte $x = X + U$ außerhalb eines gegebenen Hindernisses, dessen konkrete Definition im Dirichlet-Feld `Dir`(.,k) abgelegt ist (hierzu wird `Dir`$(2, k) \geq 210$ sein).

Die Kontaktroutinen *obstacle210/obstacle230* liefern im wesentlichen eine Skalierung α, so dass $X + \alpha U \in \mathcal{Z}$ (falls vorher verletzt) und die Information, dass von nun an dieser Knoten ein „Kontaktknoten“ ist. Dies bedeutet, dass im Löser eine Korrektur an diesen Komponenten (αU) nur noch tangential (s. „Rutsch-Randbedingungen“) zum Hindernis erlaubt ist. Weitere Informationen zur Kontaktsimulation sind in [8] und [9] zu finden.

Der eigentliche Lösungsprozess in *PPCGM* hat folgende Besonderheit. Auf der Datenstruktur `Solid`$(., k)$ $\forall k$ sind die Elementsteifigkeitsmatrizen K_{el} gehalten. Die Gesamtsteifigkeitsmatrix K ist die Assemblierung aller Elementmatrizen

$$K = \sum_{el} H_{el} K_{el} H_{el}^T$$

(H_{el} steht für eine Boolesche Matrix der Abbildung:
globale Freiheitsgrade $\rightarrow$ lokale Freiheitsgrade, als $\underline{u}_{el} := H_{el}\, \underline{u}$).

3.2.1 Datenstruktur Knoten: `X(MvonX, *)`

`MvonX` sowie die Bestandteile `MNode=3`, `NDof` (Anzahl der Freiheitsgrade pro Knoten) und `JXtoDof`, `JXContactinfo` als Zeiger werden beim Einlesen des Files *xx.std* festgelegt als `MvonX=MNode+NDof+2` für

`MNode=3`	x-, y-, z-Koordinate des Knotens, Typ REAL*4
`NDof`	Lösungswerte am Knoten, Typ REAL*4
1	INTEGER-Information als Zeiger zur Position der Knotenfreiheitsgrade im Lösungsvektor
1	INTEGER-Information zur Steuerung bei Kontaktberechnungen

Somit ist `JXtoDof=MNode+NDof+1` und `JXcontactInfo=MNode+NDof+2`.
Die INTEGER-Information `X(JXtoDof,i)` dient der Abbildung von Knotennummer i auf die Position der Freiheitsgrade in den Lösungsvektoren. Bei trilinearen Elementen sind Seitenmittenknoten/Face-Mittenknoten schon vorhanden (für gewisse i), tragen aber vorerst keinen Freiheitsgrad `X(JXtoDof,i)=0`, weil nur die Eckknoten im trilinearen Element relevant sind. Deshalb ist die Dimension des Lösungsvektors nicht `Nn*NDof`, sondern viel kleiner. Diese Abbildung (i $\longrightarrow$ Position im Lösungsvektor) ist `X(JXtoDof,i)`. Nach Berechnung der Lösung werden auf allen Knoten die Funktionswerte bereitgestellt (entweder als Kopie REAL*8 $\rightarrow$ REAL*4 oder als Interpolation für die aktuell nichtbenutzten Knoten). Deshalb steht bei Netzverfeinerung stets der richtige Wert auch für neue Knoten zur Verfügung.

Bemerkung: Der `COMMON`-Block/`X_Dof/` enthält alle zugehörigen Informationen.

3.2.2 Datenstruktur `Edge(MEdge, *)`

Die Gesamtheit der Kanteninformation bildet einen Baum, weil auch geteilte Kanten aus gröberen Netzen gespeichert bleiben, was für alle hierarchischen Lösertechniken von hoher Bedeutung ist. Die einzelne Kante `Edge(1,i)` enthält folgende `(Medge=6)` Informationen vom Typ INTEGER:

$$Nmid \mid N1 \mid N2 \mid K1 \mid K2 \mid \text{Polynomgrad}$$

Hierbei sind die ersten 3 Werte die Indices der 3 Knoten dieser Kante (Mittelknoten, Startknoten, Endknoten), die nächsten beiden sind:

bei $K1 > 0$:	$K1, K2$ die beiden Kanten-Nummern der Tochter-Kanten einer geteilten Kante,
sonst:	ungeteilte Kante des feinsten Levels, auf dem Wert $K2$ ist die wesentliche Geometrieinformation der Kante gespeichert.

Diese Geometrieinformation $k = K2$ ist:

$k = 0$:	bei inneren Kanten bzw. sonst. geraden Kanten
$k > 0$:	Bei Randkanten, die von <u>einem</u> Face mit besonderer Geometrie stammen ist k diese Geometrie-Nummer ($k < 2^{16}$).
$k > 0$:	Bei Randkanten, die der Durchschnitt von 2 Faces mit unterschiedlicher Geometrie sind, ist $k = (kg1) * 2^{16} + (kg2)$ und $kg1$ bzw. $kg2$ sind die beiden Geometrienummern dieser Faces.

Der Fall $K1 < 0$ codiert den Fakt eines möglichen "hanging nodes", also diese Kante ist nicht der Durchschnitt aller Elemente ringsum.

3.2.3 Datenstruktur `Face(MFace, *)`

Dient der Speicherung aller Außenflächen der Elemente (z. Zt. stets Vierecke). Mit `MFace=12` sind z. Zt. die Indices der 4 Kanten ($e1, \ldots e4$), der 4 Sohn-Faces (falls geteilt) $s1, \ldots s4$, sowie der Geometrie/Randbedingungsinformationen (*type*), eine Markierungsinformation, der Index des Face-Mittenknotens (*mid*) und der Polynomgrad gespeichert:

$$e1 \mid \ldots \mid e4 \mid s1 \mid \ldots \mid s4 \mid type \mid mark \mid mid \mid \text{Polynomgrad}$$

Die Information

$type = b*\texttt{RandBit}+kDir*\texttt{DirDiv}+kNeum*\texttt{NeumDiv}+kGeo$

codiert:

Randface	(ja bei $b = 1$, sonst 0)
$kDir$	Nummer der Dirichlet-Randbedingung.
$kNeum$	Nummer der Neumann-Randbedingung.
$kGeo$	Nummer der Geometrie dieses Faces.

(z. Zt.: `NeumDiv=64` (also $kGeo \leq 63$), `DirDiv=64*1024`, `Randbit=64*256`^{3}, also max. jeweils 1023 Neumann- und Dirichlet-Randbedingungen)

Für Spezialanwendungen mit z. B. sehr viel mehr Dirichlet-Randbedingungen, aber weniger Neumann (oder umgekehrt) können die Platzhalter-Variablen:
`NeumByte`/`NeumDiv` bzw. `DirByte/DirDiv` bzw. `RandBit`
im Programmteil <u>*adapmesh.inc*</u> abgeändert werden, was nach vollständigem Neucompilieren wirksam wird. Die benutzten Standartwerte sind als sinnvoller Kompromiss zu verstehen.

Analog zu `Edge(.,.)` enthält $s1$ die Anzeige, ob das aktuelle Face geteilt ist ($s1 > 0$ und $s1, \ldots, s4$ die Indices der Sohn-Faces) oder nicht ($s1 \leq 0$). Wieder codiert $s1 < 0$ die Tatsache "hanging nodes", d. h. von 2 benachbarten Elementen

Hier wird in der Schleife über alle Elemente zuerst anhand der Elementknotennummern überprüft, ob das Element neu entstanden ist oder schon im vorherigen Netz vorhanden war. Im Falle von linearen Rechnungen (wie hier bei **A3D_Original**) muss nur für die neuen Elemente die Berechnung der Elementmatrizen durchgeführt werden (Routine *element*). Vorher werden deren Elementknotennummern bereitgestellt (Routine *GetNode*) und die Abbildungen der gültigen Knoten auf die Freiheitsgrade berechnet (Routine *nodeinit*).

Die symmetrische Elementmatrix wird „sparsam" als obere Hälfte auf der Datenstruktur `Solid(JELMat,i)` abgelegt. Hiernach erfolgt die Assemblierung der rechten Seite des FE-Gleichungssystems (auf Vektor `F`) sowie der Hauptdiagonalen der Steifigkeitsmatrix (auf Vektor `K`), danach die Ergänzung der Neumann-Randbedingung auf `F`. Zur Markierung der Dirichlet-Randbedingungen wird danach die große Zahl `Oxer`$=10^{40}$ auf die entsprechenden Positionen von `K` addiert. Im Löser wird stets die Multiplikation mit der inversen Matrix $J = diag(K)^{-1}$ vorgenommen, wodurch dann an allen Dirichlet-Knoten nur Korrekturen von "0" möglich sind.

Für eine weitere Sorte von „Dirichlet-artigen" Randbedingungen (nur bei 3 oder mehr Freiheitsgraden) wird der Vektor `K` als doppelt lang (also `K(2*N)`)) gewählt.

Dies ist für sogenannte „Rutsch-Randbedingungen" vorgesehen: $\underline{u}$ am Knoten ist nicht frei ($\in \mathbb{R}^3$) sondern darf nur in einer 2-dimensionlen Ebene (gegeben durch Normalvektor $\mathbf{n}$ im Dirichlet-Feld) liegen. Hierzu wird im Löser der Knotenprojektor ($I - \mathbf{n}\mathbf{n}^T$) benötigt und $\mathbf{n}$ wird auf die entsprechende Position von `K(N+...)` übertragen.

6.4 Löser (3)

Zum endgültigen Lösen des FE-Gleichungssystems gehören mehrere Vorbereitungsroutinen und die eigentliche Lösungsroutine *PPCGM* für die iterative Auflösung des Gleichungssystems mit der vorkonditionierten Methode der konjugierten Gradienten. In der Regel besteht der Vorkonditionierer formal aus 3 Teilen

$$C^{-1} = QD^{-1}Q^T.$$

Die Multiplikationen mit Q^T und Q stellen den grundlegenden BPX-Vorkonditionierer dar und sind in den beiden Routinen *NHSTmulBPX* bzw. *NHiSmulBPX* implementiert. Hierzu ist der BPX-Informationsvektor `Hier` (in *PPCGM*) zu verwenden, der bei jeder Netzverfeinerung entsprechend aktualisiert wird.

Die Matrix D^{-1} steht für die Blockdiagonalmatrix aus C_0^{-1} (Grobgittermatrix) fortgesetzt durch eine Diagonalmatrix aus den Hauptdiagonalelementen der Steifigkeitsmatrix. Die Grobgittermatrix C_0 wird nach der ersten Berechnung aller

Es ist das einfachste St.Venant–Kirchhoff–Materialgesetz:

$$\sigma = 2\mu\,\epsilon(\underline{u}) + \lambda\, I(div\ \underline{u})$$

vorgesehen, hier bezeichnen ϵ den Verzerrungstensor und σ den Spannungstensor. Zur Implementierung wird die Voigtsche Notation verwendet. Mit

$$\underline{\epsilon} = (\epsilon_{11}, \epsilon_{22}, \epsilon_{33}, 2\epsilon_{12}, 2\epsilon_{23}, 2\epsilon_{31})^T$$
$$\underline{\sigma} = (\sigma_{11}, \sigma_{22}, \sigma_{33}, \sigma_{12}, \sigma_{23}, \sigma_{31})^T$$

und der Matrix

$$D(x) = \begin{pmatrix} x_1 & 0 & 0 \\ 0 & x_2 & 0 \\ 0 & 0 & x_3 \\ x_2 & x_1 & 0 \\ 0 & x_3 & x_2 \\ x_3 & 0 & x_1 \end{pmatrix}$$

ergibt sich

$$\underline{\epsilon} = D(\nabla)\underline{u}$$

und die schwache Formulierung der Lamégleichung ist:
finde $\underline{u} \in (\mathbb{H}^1_D(\Omega))^3$ mit

$$a(\underline{u}, \underline{v}) = f(\underline{v}) \quad \forall \underline{v} \in (\mathbb{H}^1_0(\Omega))^3.$$

Hierbei ergibt sich

$$a(\underline{u}, \underline{v}) = \int_\Omega (D(\nabla)\underline{v})^T C (D(\nabla)\underline{u})\, d\Omega$$

und

$$f(\underline{v}) = \int_\Omega \underline{v}^T f\, d\Omega + \int_{\Gamma_N} \underline{v}^T g\, d\mathcal{S}.$$

Die (6×6)-Materialmatrix ist mit obigem Materialgesetz einfach

$$C = \mu\ diag(2,2,2,1,1,1)\ +\ \lambda\, ee^T, \qquad e = (1,1,1,0,0,0)^T.$$

Die FE-Diskretisierung beider Aufgabenstellungen erfolgt mit 8-Knoten (trilinearen) Elementen, 20-Knoten (reduziert triquadratischen) Elementen oder 27-Knoten (triquadratischen) Elementen. Die Berechnung aller Elementmatrizen (und Element-rechten Seiten) des FE-Gleichungssystems übernimmt die Routine *assem*.

ist nur eines geteilt. Das Vater–Face zeigt auf die 4 Söhne. Diese 4 Faces werden als "hanging" codiert.

3.2.4 Datenstruktur `Solid(MSol, *)`

Die Struktur Solid enthält alle nötigen Informationen zu den 3-dimensionalen Finiten Elementen des aktuellen Netzes. Jedes Solid entspricht z. Zt. einem Hexaeder-Element. Anders als bei `Edge` und `Face` ist die Gesamtstruktur kein Baum, sondern die Liste der aktuell gültigen Elemente. Pro Solid sind gespeichert: die 6 Faces, Materialnummer, ein Mittelknoten, der Polynomgrad, seine (maximal 27) Knoten, die (eine oder mehrere) Element-Steifigkeitsmatrix (Typ REAL*8) und die Element-rechte-Seite.

Ebenfalls anders als bei `Edge/Face` ist die Gesamtlänge `MSol` der Einzelstruktur nicht zur Compile-Zeit festzulegen, sondern wird beispielabhänging beim Initialisieren aller Datenstrukturen berechnet, bleibt dann (COMMON-Block/X_Dof/) für einen Gesamtdurchlauf aber fest. Diese Festlegung übernimmt der Modul `SetMdims`. Hier wird zuerst die Anzahl der Knoten pro Element berechnet `(Nodes_In_El)`, die zwar schon als Starteingabe mit 8, 20 oder 27 festgelegt war, allerdings ist eine Erweiterung auf „p-Methode“ denkbar, wo die Gesamtzahl der gültigen Elementknoten aus den 3 Polynomgraden der Edge-, Face- und Solid-Formfunktionen berechnet werden könnte.

Mit der Festlegung von `(Nodes_In_El)` ist dann die sparsame Planung des Speicherbedarfs für eine Element-Steifigkeitsmatrix samt rechte Seite möglich:
`(NDof*Nodes_In_El)*(NDof*Nodes_In_El+1)/2` Doppelworte (REAL*8)
würde die Matrix unter Ausnutzung ihrer Symmetrie benötigen. Also Matrix + rechte Seite benötigen:
`MelMat=(NDof*Nodes_In_El)*(NDof*Nodes_In_El+3)` Worte (INTEGER*4)
zur Speicherplatzrechnung.

Zusätzlich wurde für ein ausführliches Post-Processing die Möglichkeit der Speicherung von element-basierten Größen (Ableitungen in den Knoten oder Spannungswerte) vorgesehen, insgesamt `mElUS` Worte. Im Falle nichtlinearer Aufgaben wird die Elementmatrix stets neu berechnet (da lösungsabhängig), weshalb ihr Speicherplatz beim Post-Processing genutzt werden kann. Also wird dann `mElUS=0` sein, aber bei linearen Aufgaben muss ein zusätzlicher Speicher für die Belange des Post-Processing vorgesehen werden. Mit der Festlegung `MelFac=9` (für 6 Face-Nummern, 1 Materialbereichsnummer, 1 Mittelknoten-Nummer, Polynomgrad) und `MelNod=27` (Platz für die max. 27 Knotennummern des Elementes) ergibt sich insgesamt

```
MSol = MelFac + MelNod + mElUS + MelMat .
```

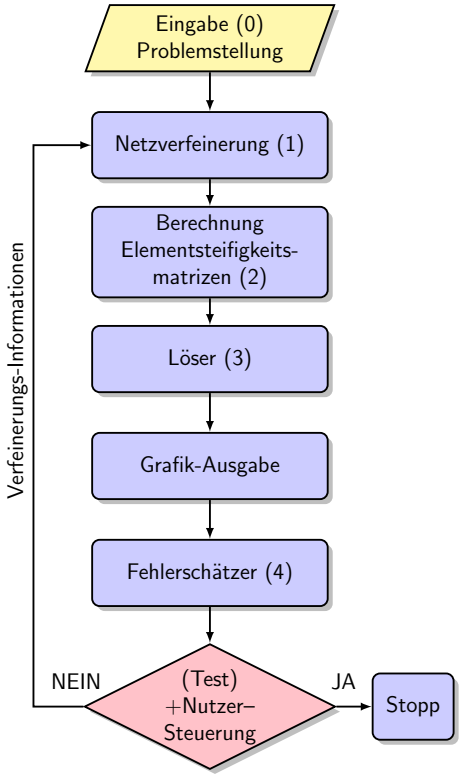
Bemerkung: Da die Elementmatrix aus REAL*8 Einträgen besteht, ist die Einhaltung der Doppelwortgrenzen für die Matrixstartadresse

`Solid(jMat, i), jMat=MelFac+MelNod+mElUs+1, ∀ i`

notwendig. Hierzu ist `MelFac + MelNod + mElUS` als gerade Zahl garantiert.

4 Gesamtablauf

Der Gesamtablauf aller Programmvarianten geht aus folgendem Ablaufdiagramm hervor.



Bei linearen Aufgaben wird im Test lediglich über die Benutzung der markierten Elemente zur Verfeinerung entschieden. Bei nichtlinearen Rechnungen wird häufig

(a) 1 Freiheitsgrad (im File *xxx.std* angegeben)

Lösung einer Reaktions-Diffusions-Gleichung vom Typ

$$\alpha_1 u_{,xx} + \alpha_2 u_{,yy} + \alpha_3 u_{,zz} + \gamma u = q$$

im 3D-Gebiet Ω ($u = u(x, y, z)$ ist skalare Funktion). Als Randbedingungen kommen
Dirichlet-Randbedingungen ($(u(x, y, z)$ auf Randflächen $\subset \Gamma_D$ gegeben) oder
Neumann-Randbedingungen ($\frac{\partial u}{\partial \mathfrak{n}} = g$ auf Randflächen $\subset \Gamma_N$ gegeben, $\mathfrak{n}$ ist deren Außennormale) in Betracht.
Es ist denkbar, dass für Rand-Faces im Eingabe-File überhaupt keine Randbedingungen codiert sind, dies zieht homogene Neumann-Randbedingungen ($\frac{\partial u}{\partial \mathfrak{n}} = 0$) nach sich. Die Konstanten α_i, γ in der Differentialgleichung können verschieden in den einzelnen "Solids" des Eingabenetzes sein (Materialbereiche im Eingabefile).

Die Funktion $q(x, y, z)$ der rechten Seite ist im Eingabefile als konstant pro Solid vorgebbar, für andere Fälle muss eine Programmanpassung vorgenommen werden (etwa: *bsp.f*).

Die Darstellung der Differentialgleichung als:
finde $u \in \mathbb{H}^1_D(\Omega)$ mit

$$a(u, v) = f(v) \quad \forall v \in \mathbb{H}^1_0(\Omega)^{\ddagger}$$

ist Ausgangspunkt für die FE-Formulierung mit

$$a(u, v) = \int_\Omega (\nabla v)^T A(\nabla u) + \gamma uv \, d\Omega, \quad A = diag(\alpha_1, \alpha_2, \alpha_3),$$
$$f(v) = \int_\Omega fv \, d\Omega + \int_{\Gamma_N} gv \, d\mathcal{S}.$$

(b) 3 Freiheitsgrade (im File *xxx.std* angegeben)

Lösung der linearen Elastitätsgleichung (Lamé-Gleichung) für die gesuchte Vektorfunktion $\underline{u}$

$$div\, \sigma(\underline{u}) + f = 0 \text{ in } \Omega.$$

Als Randbedingungen fungieren wieder
- Dirichlet-Randbedingungen: $\underline{u}$ auf Randflächen $\subset \Gamma_D$ gegeben oder
- Neumann-Randbedingungen: $\sigma(\underline{u}) \cdot \mathfrak{n} = g$ auf Randflächen $\subset \Gamma_N$ gegeben.

‡ dies bezeichnet den Sobolev-Raum der Funktionen mit erster verallgemeinerter Ableitung, der Index 0 steht für: $v = 0$ auf Γ_D, wo u vorgegeben ist.

'i': Änderung der maximalen Iterationszahl des PCG-Lösers

'e': Änderung des relativen Abbruchkriteriums beim PCG-Löser

'r': eine Startverfeinerung anstelle Start mit Grobnetz

's': Eingabe eines Vektors (3 Komponenten), der als Startvektor anstelle (0,0,0) für alle Komponenten des Verschiebungsvektors zu benutzen ist. Damit kann z. B. bei Kontaktrestriktionen für einen gültigen Kontakt mindestens eines Knotens mit dem Hindernis gesorgt werden.

'v': Steuerung des Vorkonditionierers, Standard: Ivar=5 für BPX mit Grobgitterlöser, Ivar=4 heißt ohne Grobgitterlöser, weitere Wahlen (0...3) sind möglich, aber selten sinnvoll

'z': Zwischenausgabensteuerung

'T': für Erweiterungen bei nichtlinearer Rechnung: Eingabe einer Anzahl von Inkrementierungsschritten

6.2 Netzverfeinerung (1)

Alle Routinen, die der adaptiven Netzverfeinerung dienen, sind im Directory *mylibs/aFineQ* zusammengefasst. Die Netzverfeinerung wird im Hauptprogramm durch den Aufruf *CALL netfine (...)* gestartet.

Die Routine *netfine* (in *divsolid.f*) besteht im wesentlichen aus der Schleife über alle Elemente (Solids), wo getestet wird, ob das jeweilige Element geteilt werden sollte und ruft in diesem Fall die eigentliche Teilungsroutine *div8solid* auf. Die Markierung am i-ten Element (`Solid (1,i)`), die der Fehlerschätzer bereitgestellt hat, ist die mit (-1) multiplizierte Kantennummer seiner ersten Kante (also: `Solid (1,i)`$< 0 \Rightarrow$ teile dieses Element). Daneben gibt es noch weitere Gründe ein Element zu teilen, so etwa zu viele "hanging nodes" an seinen Außenflächen. Dies wird in *teiltest* untersucht.

Die Teilung eines Hexaeder-Elements erfordert danach viele Hilfsoperationen zu Kanten- und Flächenteilungen, damit die neuen 8 Sohn-Elemente korrekt gebildet werden. Dies ist ausführlich in [4] dargelegt.

6.3 Berechnung der Elementmatrix (2)

Zu Anfang muss der Leistungsumfang der Grundvariante **A3D-Original** aufgezeigt werden. Diese Programmvariante dient der FE-Approximation folgender beider Lösungen von Differentialgleichungen:

die Netzverfeinerung innerhalb der inkrementellen Newtonschritte verboten, um auf festem (sinnvollerweise relativ grobem) Netz die Newton-Konvergenz abzuwarten. Erst dann liegt die Lösung auf dem aktuellen Netz auskonvergiert vor und der Fehlerschätzer steuert eine sinnvolle weitere Verfeinerung, eingebettet in jeweils einen Newtonschritt.

Die Hauptaktionen (0) bis (4) sind im jeweiligen Hauptprogramm durch die folgenden UP-Rufe realisiert:

(0) CALL *SetFile*
CALL *SetPointers*
CALL *GrobnetzRead*
CALL *GrobnetzKorr*

(1) CALL *NetFine*

(2) CALL *Assem*

(3) CALL *Stave*
CALL *PPCGM* (oder ein anderer Löser)

(4) CALL *EfromU*
CALL *ErrEst*

Diese Hauptunterprogramme finden sich in den Quellfiles unter `./mylibs/` in

(0) `aNetzQ/*`

(1) `aFineQ/*` , Hauptroutine in *divsolid.f*

(2) `aElemQ/*`

(3) `aSolvQ/*`

(4) `aElemQ/*`

Bei (Test) findet die wesentliche Beeinflussung des Bedieners statt. Im Normalfall wird hier ein Steuerbyte über die Standarteingabe gelesen, das die Art der Fortsetzung zu (1) oder die Beendigung der Simulation nachsichzieht. Ebenso können zwei „Automatik"–Regimes geschaltet werden, die die sofortige Schleifenfortsetzung ohne Nutzereingabe bis zu einer Endsituation ermöglichen.
Die einzugebenden Steuerbytes haben z. Zt. folgende Codierung:

'0' Fortsetzung bei (1) ohne jede Netzverfeinerung
(alle Verfeinerungsmarkierungen werden gelöscht).

't' Fortsetzung bei (1) mit Totalverfeinerung
(jedes Element wird vollständig geteilt).

' ' (oder `Enter`) - übliche Fortsetzung bei (1) mit Verfeinerung der markierten Elemente

'r' Stop der aktuellen Simulation und Fortsetzung bei (0) mit gleichem Inputfile

'n' Stop der aktuellen Simulation und Fortsetzung bei (0) mit neuem Inputfile

'e' Beendigung des Programms

'a' "automatic": automatische Fortführung mit der adaptiven Netzverfeinerung entsprechend der markierten Elemente ohne Stop bei (Test) bis zu einer maximalen Anzahl von Knoten.

'A' "automatic" für nichtlineare Rechnungen: Fortführung ohne jede Netzverfeinerung, dafür Newton-Schleife für das aktuelle Inkrement und Erhöhung der Inkrementierung bis 1. Dies setzt das vorherige Einschalten der Inkrementierung (s. folgendes) voraus.

'1' Einschalten/Ausschalten der Erhöhung der Inkrementierung nach jedem konvergierten Newton-Prozess

'?' ergibt eine Ausgabe aller gültigen Steuerbytes, weitere hier nicht spezifizierte können für die grafische Ausgabe vorgesehen sein.

5 Parallelisierung

Durch die Verbreitung von Mehrprozessor–Architekturen ist die Parallelisierung von wesentlichen Teilschritten einfacher via `OpenMP` möglich und stimmt nicht mehr mit der Schilderung in [12] überein.

Es ist jetzt lediglich eine einfache Parallelausführung der Schleife über alle Elemente in *Assem*:

`for each` Element `do`:
    Berechne die Elementmatrix

und bei der Matrix-Vektor-Multiplikation im Löser:
`for each` Element `do`:

- Multipliziere $y_{el} := A_{el} * x_{el}$ mit der Elementmatrix
- Akkumuliere y_{el} zum Gesamtvektor.

vorzusehen.

Während im ersten Fall jedes Element völlig unabhängig verarbeitet wird, muss im zweiten Fall beim Schritt „Akkumuliere“ ein Schutzmechanismus vorgesehen

werden, die Addition der Werte von y_{el} (privat in verschiedenen Prozessoren) auf den Ergebnisvektor als „atomare Operation“ auszuführen. Da diese Grundoperation zur Akkumulation von Elementvektoren in einen globalen Vektor vollständig vom Unterprogramm *Fakku* übernommen wird, ist lediglich darin die besondere Compilerdirective "`!$omp atomic`" eingefügt, die ohne OpenMP einfach ignoriert wird. Weitere Informationen sind in [16] zu finden.

6 Die Grundvariante A3D_Original und ihre Bibliotheken

Entsprechend des Gesamtdurchlaufs (s. obigen Ablaufplan) werden die wesentlichsten Implementierungsdetails innerhalb der Schritte (0) bis (4) und vor allem die grundlegenden Aufgaben der beteiligten Routinen geschildert.

6.1 Netzeingabe- und Start-Routinen (0)

Die eigentliche Leseroutine zum Einlesen und Decodieren der Files vom Typ *xxx.std* ist ein seit längerem existierender C-Code, der im Aufruf CALL *data_read*(...) zu sehen ist (Routine *Set_Pointers* und *grobnetzread*). Die Routine *SetFile* legt den zu benutzenden Filenamen *xxx* fest, danach wird über die Routine *ReadElem* die Art der zu benutzenden Finite Elemente (etwa 8-, 20-, 27-Knoten-Elemente) festgelegt. In *Set_Pointers* werden schon die Datenstrukturen fester Länge eingelesen, dann aber die Speicherplatzvorausplanung, v. a. Setzen der Startadressen der variablen Datenstrukturen vorgenommen. Die endgültige Eingabe dieser Daten erfolgt in *grobnetzread* (schreiben auf die vorberechneten Positionen) und deren Korrektur auf interne Datenstruktur in *grobnetzkorr*. Hier wird vor allem für die Generierung aller Kantenmittenknoten und Face-Mittenknoten gesorgt, weshalb zuerst eine ev. besondere Geometrieinformation eines Faces auf seine Kanten übertragen wird. Die Korrektur eines vorläufig berechneten Knotens auf die wahre Geometrie wird von *pcorect* geleistet.

Die Routine *read_rbfinfos* dient einer Besonderheit in der Definition von Kontaktrestriktionen für den Fall, dass eine Hindernisformulierung keine einfache Geometrie hat, sondern als Isofläche (basierend auf radialen Basisfunktionen [11]) in einem gesonderten File *xxx.rbf* abgelegt ist.

Danach komplettieren die Routinen:
hieranf (Initialisierung der BPX-Informationen), *StartControl* und *StartXinit* die Vorbereitung. In *StartControl* können durch Eingabe eines Steuerbytes Änderungen an Standardeinstellungen vorgenommen werden. Die wichtigsten sind: