Praktikumsbericht

über ein Praktikum vom 22. Februar bis 26. März 2010 beim

Deutschen Zentrum für Luft- und Raumfahrt
Simulations- und Softwaretechnik
Verteilte Systeme und Komponentensoftware

Betreuer: Dr.-Ing. Achim Basermann, Dr. Hans-Peter Kersken

Franziska Krüger

24. März 2010

Inhaltsverzeichnis

1 IPOPT **2**
1.1 Allgemeines zum Software-Paket IPOPT . . . 2
1.2 Algorithmus von IPOPT . . . 3

2 Einführung in die Mehrziel-Optimierung **4**

3 AutoOpti **6**
3.1 Softwaretechnischer Hintergrund von AutoOpti . . . 6
3.2 Algorithmus von AutoOpti . . . 6

4 DesParO **8**
4.1 Softwaretechnischer Hintergrund von DesParO . . . 8
4.2 Grundzüge des Algorithmus von DesParO . . . 9
4.3 Graphische Benutzeroberfläche von DesParO . . . 10

5 MOPS **10**
5.1 Softwaretechnischer Hintergrund von MOPS . . . 11
5.2 Algorithmus von MOPS . . . 12

Literaturverzeichnis **15**

Abbildungsverzeichnis **16**

A Modellierung eines Beispiels in IPOPT **17**
A.1 Das Beispiel . . . 17
A.2 Modellierung in AMPL . . . 17
A.2.1 hs71.mod . . . 17
A.3 Modellierung in C . . . 18
A.3.1 hs071_c.c . . . 18
A.4 Modellierung in C++ . . . 23
A.4.1 hs071_nlp.hpp . . . 23
A.4.2 hs071_main.cpp . . . 26
A.4.3 hs071_nlp.cpp . . . 27
A.5 Modellierung in Fortran . . . 33
A.5.1 hs071_f.f . . . 33

1 IPOPT

IPOPT (**I**nterior **P**oint **OP**Timizer, gesprochen I-P-Opt; IBM Research, Universität Lehigh, Universität Basel, vgl. [1]) ist ein Open-Source-Software-Paket zur hochparallelen Lösung großer, nichtlinearer Optimierungsprobleme mit einer Zielfunktion. Es wurde zur Lösung von allgemeinen nichtlinearen Problemen der Form

$$(1.1)\begin{cases} & \min_{x\in\mathbb{R}^n} f(x) \\ \text{mit} & g^L \leq g(x) \leq g^U \\ \text{und} & x^L \leq x \leq x^U \end{cases}$$

entwickelt. Hierbei ist $f : \mathbb{R}^n \to \mathbb{R}$ die zu minimierende Funktion und $g : \mathbb{R}^n \to \mathbb{R}^m$ die Nebenbedingung, wobei f und g nichtlinear, nichtkonvex und hinreichend glatt (mindestens einmal stetig differenzierbar, wünschenswert wäre zweimal stetig differenzierbar) sind. Gleichungsrestriktionen können durch $g^L = g^U$ realisiert werden.
IPOPT findet lokale Lösungen von (1.1).

IPOPT wurde vor allem für sehr große, nichtlineare Probleme entwickelt. Die Rede ist hier von bis zu Millionen von Variablen, wobei die Probleme auch einen großen Freiheitsgrad aufweisen dürfen. Im Rahmen seiner Doktorarbeit [2] hat der Entwickler von IPOPT, Andreas Wächter, Probleme mit 160.000 Variablen gelöst.

1.1 Allgemeines zum Software-Paket IPOPT

IPOPT ist in der Programmiersprache C++ geschrieben und wurde als Open-Source-Code unter der Common Public License (CPL) herausgegeben.
Es kann Optimierungsprobleme entweder durch Verwendung einer mathematischen Modellierungssprache (wie z.B. AMPL, vgl. [3, 4]) oder durch Formulierung des Problems als Code (in C++, C, Fortran 77 oder Matlab) verarbeiten. Beispiele zur konkreten Problemmodellierung in den verschiedenen Sprachen finden sich nach der Installation von IPOPT unter *examples*; eine Zusammenstellung ist zudem im Anhang beigefügt.

Zur Installation von IPOPT werden mehrere Software-Pakete benötigt (vgl. [5, 6]), angefangen mit IPOPT [7]. IPOPT kann sowohl auf Linux/UNIX- als auch auf Mac OS X- und Windows-Plattformen genutzt werden.

Zudem wird ein Löser für schwach besetzte, symmetrische, indefinite Matrizen benötigt. Mögliche Löser sind

- MA27 oder MA57 aus der Harwell Subroutine Library [8],
- PARDISO (**PAR**allel Sparse **DI**rect **SO**lver, siehe [9]) oder
- WSMP (**W**atson **S**parse **M**atrix **P**ackage, siehe [10]).

Im Rahmen meines Praktikums hab ich mich mit PARDISO beschäftigt. Dieser Löser ist für akademische Zwecke (Universitäten, Forschungszentren, Bundesagenturen) frei verfügbar; die kommerzielle Nutzung ist jedoch kostenpflichtig (einmalig 300,00 CHF). In beiden Fällen ist die Lizenz User- und Host-gebunden. PARDISO ist für 32bit- und 64bit-Architekturen verfügbar.

Desweiteren müssen LAPACK (Linear Algebra PACKage - eine Softwarebibliothek zur effizienten Lösung linearer Gleichungssysteme unter der BSD-Lizenz, siehe [11]) und BLAS (Basic Linear Algebra Subprograms - eine Softwarebibliothek zur Durchführung elementarer Operationen der linearen Algebra

wie Vektor- und Matrixmultiplikationen, siehe [12]) vorhanden sein. Beide Softwarebibliotheken sind als Open-Source-Code frei zugänglich.

Möchte man eine Modellierung mit AMPL (A Mathematical Programming Language) vornehmen, so ist auch hier ein Download erforderlich. Auch AMPL ist als Open-Source-Code freigegeben.

Zur Installation muss IPOPT mit entsprechenden Optionen (u.a. bzgl. des verwendeten Lösers) ausgeführt werden. Anschließend können die jeweiligen Programme (in AMPL oder in C, C++, Fortran) über die Konsole ausgeführt werden. Die Lösung des nichtlinearen Optimierungsproblems erscheint daraufhin als Ausgabe in der Konsole.

1.2 Algorithmus von IPOPT

Bei IPOPT handelt es sich um eine primal-duale Barrier-Methode, die eine Folge von Barrier-Problemen löst. Die Suchrichtungen können entweder durch Benutzung eines *Full-Space* oder eines *Reduced-Space*-Ansatzes berechnet werden, wobei letzterer die problemabhängige Struktur ausnutzen kann und Annäherung der Hessematrix ermöglicht. Globale Konvergenz wird durch den sogenannten *Line-Search*-Ansatz gesichert. Hierbei wird dem Nutzer die Auswahl zwischen mehreren Merit-Funktionen oder einer neuen Line-Search-Filter-Methode ermöglicht. Es hat sich herausgestellt, dass letztere in den meisten Anwendungsfällen die robustere und effizientere Methode ist.

IPOPT ersetzt intern alle Ungleichungsrestriktionen $g^L \leq g(x) \leq g^U$ durch Gleichungsrestriktionen und Restriktionen von neuen Schlupfvariablen s_i (z.B. $g_i(x) - s_i = 0$ mit $g_i^L \leq s_i \leq g_i^U$). Desweiteren nehmen wir für unsere Betrachtungen der Einfachheit halber an, dass x nach unten nur durch Null beschränkt ist. Es ergibt sich dann insgesamt folgendes vereinfachtes Optimierungsproblem

$$(1.2) \begin{cases} & \min\limits_{x \in \mathbb{R}^n} f(x) \\ \text{mit} & c(x) = 0 \\ \text{und} & x \geq 0 \end{cases} .$$

Als Innere-Punkte- (bzw. Barrier-) Methode betrachtet IPOPT dann folgende Hilfsprobleme

$$(1.3) \begin{cases} \min\limits_{x \in \mathbb{R}^n} \ \varphi_\mu(x) = f(x) - \mu \sum\limits_{i=1}^{n} ln(x_i) \\ \text{mit } c(x) = 0 \end{cases}$$

mit einem Barrier-Paramter $\mu > 0$. Nähert sich eines der x_i von oben der Null an, so geht $\varphi_\mu(x) \to \infty$. Demnach wird sich eine optimale Lösung im Inneren der zulässigen Mengen (d.h. $x_i > 0 \ \forall i \in \{1, \ldots, n\}$) befinden. Die Gewichtung des sogenannten Barrier-Terms ($-\mu \sum_{i=1}^{n} ln(x_i)$) richtet sich nach der Größenordnung von μ. Man kann unter gewissen Voraussetzungen zeigen, dass eine optimale Lösung $x^*(\mu)$ von (1.3) für $\mu \to 0$ gegen eine optimale Lösung des ursprünglichen Problems (1.2) konvergiert. Demnach löst IPOPT eine Folge von Hilfsproblemen (1.3). Gestartet wird mit einem $\mu > 0$, z.B. $\mu = 0.1$; das zugehörige Hilfsproblem (1.3) wird dann mit relativ geringer Genauigkeit gelöst. Danach wird μ weiter verkleinert, die geforderte Genauigkeit wird erhöht und das daraus entstehende Problem (1.3) wird gelöst. Dieses Verfahren wird solange wiederholt, bis die erhaltene Lösung die Optimalitätsbedingungen 1. Ordnung hinreichend gut erfüllt. Beachte, dass IPOPT hierbei nicht notwendigerweise gegen ein lokales Minimum konvergiert; lokale Maxima und Sattelpunkte sind nicht ausgeschlossen.

Noch zu klären ist die Frage, wie sich die Hilfsprobleme (1.3) für ein festes $\mu > 0$ lösen lassen. Die

Optimalitätsbedingungen 1. Ordnung von (1.3) sind gegeben durch

$$\begin{cases} \nabla f(x) + \nabla c(x)y - z = & 0 \\ c(x) = & 0 \\ XZe - \mu e = & 0 \\ x, z \geq & 0 \end{cases}$$

mit $y \in \mathbb{R}^m$, $z \in \mathbb{R}^n$ Lagrange'sche Multiplikatoren für die Gleichungsrestriktionen und Schranken, den Diagonalmatrizen $X = diag(x)$, $Z = diag(z)$ und $e = (1, \cdots, 1)^T$. Durch die ersten drei Gleichungen ergibt sich ein nichtlineares Gleichungssystem, auf welches sich nun ein Newton-Algorithmus anwenden lässt. Damit lässt sich eine Folge von Iterierten erzeugen, welche die Ungleichungsrestriktion $x, z \geq 0$ streng erfüllen. Nachdem somit der Newtonschritt berechnet wurde, muss noch eine geeignete Schrittweite gewählt werden. Hierzu wird vorerst eine maximale Schrittweite bestimmt, sodass die Ungleichungsrestriktionen immer noch strikt erfüllt sind. Anschließend wird eine sog. *Line Search* mit Testschrittweiten (diese erhält man durch Multiplikation der maximalen Schritte mit 2^{-l}, $l = 0, 1, \ldots$) durchgeführt, bis die Schrittweite einen „hinreichenden Fortschritt“ aufweist. Um zu entscheiden, ob ein so gefundener Punkt akzeptabel ist und man sich der optimalen Lösung weiter angenähert hat, wird die sog. Filter-Methode angewandt: Ein Punkt wird als besser erachtet als der vorherige, wenn entweder der Funktionswert $\varphi_\mu(x)$ oder die Norm der Restriktion $||c(x)||$ verringert wird. Diese Verbesserung muss allerdings nicht nur zur vorherigen Iterierten, sondern zu einer ganzen Liste von vorherigen Iterierten gewährleistet sein. Dadurch wird u.a. das sog. *Cycling* verhindert.

Unter Standardvoraussetzungen kann man zeigen, dass obiger Algorithmus global konvergent ist, d.h. dass für jeden beliebigen Startpunkt zumindest ein Grenzwert ein stationärer Punkt von (1.3) ist.

Beachte: Damit es sich bei der Newtonrichtung tatsächlich um eine Abstiegsrichtung handelt, muss gewährleistet sein, dass die Matrix zur Berechnung der Newtonrichtung auf einem gewissen Unterraum positiv definit ist. Ist dies nicht der Fall, so addiert IPOPT ein positives Vielfaches der Einheitsmatrix zu der ursprünglichen Matrix, sodass die resultierende Matrix dann positiv definit auf dem geforderten Unterraum ist.
Desweiteren kann es passieren, dass keine akzeptable Schrittweite gefunden wird; wir sind also eventuell in jede Richtung unzulässig. In diesem Fall geht der Algorithmus in eine sogenannte *Zulässigkeits-Erholungs*-Phase (engl.: feasibility restoration phase) über, in welcher die eigentliche Zielfunktion kurzzeitig vollständig ignoriert wird und nur die ℓ_1-Norm der Restriktion $||c(x)||_1$ minimiert wird. Man versucht also, einen zulässigen Punkt in der Umgebung des Punktes zu finden, an dem die Zulässigkeits-Erholungs-Phase startete. Als Resultat der Minimierung erhält man entweder einen zulässigen Punkt (d.h. $||c(x)||_1$=0), sodass eine Rückkehr zum herkömmlichen IPOPT-Algorithmus möglich ist, oder es wird ein lokales Minimum von $||c(x)||_1$ gefunden, welches unzulässig ist (d.h. $||c(x)||_1 > 0$). Letzteres signalisiert dem Benutzer, dass das Problem (lokal) nicht zulässig ist (Konsolen-Ausgabe: `EXIT: Converged to a point of local infeasibility. Problem may be infeasible.`). In diesem Fall kann versucht werden, durch Änderung des Startpunktes ein anderes Ergebnis zu erhalten.

2 Einführung in die Mehrziel-Optimierung

Im folgenden soll die Mehrziel-Optimierung kurz eingeführt werden. Für weiterführende Literatur verweise ich auf [13].

Allgemein betrachtet man in der Mehrziel-Optimierung Probleme der Form

$$\text{„} \min_{x \in \mathscr{F}} \text{“} (f_1(x), \ldots, f_p(x)) \ . \tag{2.1}$$

$\mathscr{F} \subseteq \mathbb{R}^n$ ist die zulässige Menge, $f_i : \mathbb{R}^n \to \mathbb{R}$ für $i = 1, \ldots, p$ sind die verschiedenen Ziele (zu „minimierende“ Funktionen), wobei die Art der Minimierung noch zu klären ist. Eine eindeutige Form der Minimierung (wie es für Funktionen $f : \mathbb{R}^n \to \mathbb{R}$ der Fall ist) gibt es hier nicht mehr, da der $\mathbb{R}^p$ für $p > 1$ keine natürliche Ordnungsstruktur aufweist.

Bei unseren Mehrziel-Optimierungsaufgaben werden wir demnach als Lösung keinen einzelnen Punkt, sondern ein Menge an Kompromisslösungen, die sogenannte Pareto-Menge, erhalten. Bevor wir diesen Begriff näher erläutern, definieren wir die Relation *Dominanz* [14]:
Gegeben sei das Mehrziel-Optimierungsproblem (2.1). Ein Punkt $x \in \mathscr{F}$ dominiert $y \in \mathscr{F}$, wenn:

$$\begin{aligned} f_i(x) &\leq f_i(y) \qquad \forall i \in \{1, \ldots, p\} \text{ und} \\ f_j(x) &< f_j(y) \qquad \text{für ein } j \in \{1, \ldots, p\}. \end{aligned}$$

Zwei Lösungen x und y können also in drei möglichen Varianten in Beziehung zueinader stehen: x dominiert y, x wird durch y dominiert oder x und y dominieren sich nicht.

Mit Hilfe obiger Dominanzrelation lassen sich nun also die Begriffe *nichtdominiert*, *Pareto-Optimum* und *Pareto-Menge* erklären:
Ein Punkt $x \in \mathscr{F}$ heißt nichtdominiert genau dann, wenn kein Element $y \in \mathscr{F}$ existiert, welches x dominiert. x heißt dann Pareto-Optimum.
Eine Menge $X \subseteq \mathscr{F}$ heißt Paretomenge genau dann, wenn sie nur aus nichtdominierten Punkten besteht.

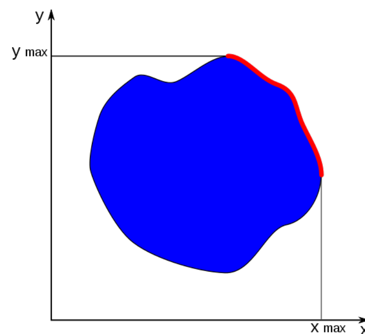

Abbildung 2.1: Pareto-Menge (rot) einer Mehrziel-Optimierung (hier Maximierung) über der Menge $\mathscr{F}$ der zulässigen Funktionswerte (blau); zwei Zielfunktionen (Quelle: [15]).

Ein Individuum befindet sich also in der Pareto-Menge, wenn die Minimierung eines Zielfunktionswerts nur noch durch gleichzeitige Erhöhung eines anderen stattfinden kann (siehe Abbildung 2.1).
Beachte: Ist ein Individuum nicht in der Pareto-Menge enthalten, so kann es keinesfalls eine Lösung unseres Mehrziel-Problems sein, da es möglich ist, einen Zielfunktionswert weiter zu minimieren, ohne dabei an anderer Stelle eine nicht gewünschte Erhöhung eines Wertes zu erhalten.

Unter dem *Pareto-Rang* eines Individuums verstehen wir die Anzahl der Individuen, die das Individuum dominieren.

Wie schon oben erwähnt ist die Art der Minimierung im Fall der Mehrziel-Optimierung nicht mehr eindeutig bestimmt. Um einen Eindruck der verschiedenen Möglichkeiten zu erhalten, wollen wir im Folgenden beispielhaft drei mögliche Arten der Minimierung näher betrachten:
Lexikographische Sortierung nach Wichtigkeit: Wir betrachten als Beispiel verschiedene Kriterien beim Autokauf (bspw. Anschaffungspreis, Benzinverbrauch, Leistung - schon nach Wichtigkeit sortiert). In

diesem Fall wird der Käufer das Auto mit dem geringsten Anschaffungspreis bevorzugen.
Min-Max-Optimierung: Angenommen, wir betrachten zwei Zielfunktionen, die verschiedene Umweltbelastungen darstellen sollen. In diesem Fall ist es wichtig, dass eine Minimierung des einen Zielfunktionswerts nicht auf Kosten des anderen stattfindet. Als mögliche Strategie könnte man über die maximale Zielfunktion minimieren, d.h. $\min_{x \in \mathscr{F}} \max_{i=1,2} f_i(x)$, wobei die Maximierung der Zielfunktionen als punktweise Maximierung in allen $x \in \mathscr{F}$ zu interpretieren ist.
Gewichtete-Summen-Methode: Hierbei wird nur eine einzige Zielfunktion minimiert. Man betrachtet dabei ein Optimierungsproblem vom Typ $\min_{x \in \mathscr{F}} \sum_{k=1}^{p} \lambda_k f_k(x)$ mit gewissen Gewichtungsfaktoren $\lambda_k \in \mathbb{R}$.

3 AutoOpti

AutoOpti ist ein am DLR-Institut für Antriebstechnik in Köln-Porz entwickelter automatischer Optimierer (vgl. [16, 17, 18]). Enwickelt wurde AutoOpti zur Beschleunigung des aerodynamischen Auslegungsprozesses von Verdichterschaufeln bzw. Verdichterprofilen und zur Entwicklung effizienterer Beschaufelungen. Da eine solche Auslegung immer einen Kompromiss zwischen verschiedensten Anforderungen darstellt, wurde bei AutoOpti besonderer Wert auf die Möglichkeit zur gleichzeitigen Optimierung mehrerer Zielfunktionen gelegt (sog. *Multi-Objective* bzw. *Multi-Criteria*-Optimierung).
Bisher findet AutoOpti Anwendung bei Problemen mit bis zu 250 freien Parametern (bei 1250 erzeugten Individuen betrug die Rechenzeit auf 130 Prozessoren ca. zwei Monate).
Eine weitere Schwierigkeit stellt die i.A. fehlende Glattheit der Zielfunktionen dar (die Zielfunktionen sind in der Regel weder differenzierbar noch stetig). Aus diesem Grund finden die bekannten klassischen Optimierungsverfahren hier keine Anwendung.

3.1 Softwaretechnischer Hintergrund von AutoOpti

AutoOpti ist in der Programmiersprache C geschrieben und wurde für Linux-Architekturen entwickelt. Die zu lösenden Optimierungsprobleme werden in C (und Python) formuliert. Ausgeführt wird AutoOpti über die Konsole. Als Lösung erhält der Benutzer wieder C-Dateien. Diese enthalten die mit Hilfe des Verfahrens konstruierten Punkte und deren Eigenschaften.
AutoOpti arbeitet hoch parallel. Die Kommunikation zwischen dem eigentlichen Optimierungsprozess und den Metamodellen sowie die Master-Slave-Kommunikation (siehe Abbildung 3.1) finden über Datei-Austausch statt. Eine Parallelisierung mittels MPI und OpenMP wurde innerhalb der einzelnen Prozesse realisiert. Zudem ist es möglich, die Anzahl der zur Optimierung benutzten Cluster zu erweitern oder runterzufahren. Somit wird eine optimale Ausnutzung der vorhandenen Ressourcen unterstützt.

3.2 Algorithmus von AutoOpti

Der Algorithmus von AutoOpti basiert grundsätzlich auf der Evolutionsstrategie. Nach dem Vorbild der Natur (*survival of the fittest*) wird die Menge der Individuen (d.h. hier eine spezielle Auslegung einer Verdichterschaufel) durch Mutation, Selektion und Vererbung stetig erweitert und verbessert.
Die Güte eines einzelnen Individuums wird anhand des Pareto-Rangs gemessen. Hierbei gehen wir folgendermaßen vor: Wir betrachten die Menge aller Individuen und suchen die Individuen mit Pareto-Rang Null. Diesen Pareto-Rang weisen wir ihnen zu. Anschließend entfernen wir die Pareto-Rang-Null-Individuen aus der Menge aller Individuen und suchen erneut nach neuen Pareto-Rang-Null-Individuen. Diesen weisen wir den Pareto-Rang Eins zu. Dies führen wir so lange fort, bis allen Individuen ein Pareto-Rang zugewiesen wurde. Individuen, die nichtdominiert sind haben also Pareto-Rang Null (je nach Konvention/Definition auch Eins). Ziel ist es eine Menge von nichtdominierten Individuen zu erhalten. Die Ausgabe von AutoOpti beinhaltet also Individuen aus der Pareto-Menge.
Durch die Verwendung eines genetischen Algorithmus müssen keine weiteren Voraussetzungen hinsichtlich der Glattheit der Zielfunktion gestellt werden.

Die Restriktionen an die Parameter gehen über sogenannte *Penalty*-Terme (Straf-Terme) in die Optimierung ein. Ist eine der Restriktionen verletzt, so erhöht sich der Wert der Zielfunktion. Dies möchte man vermeiden.

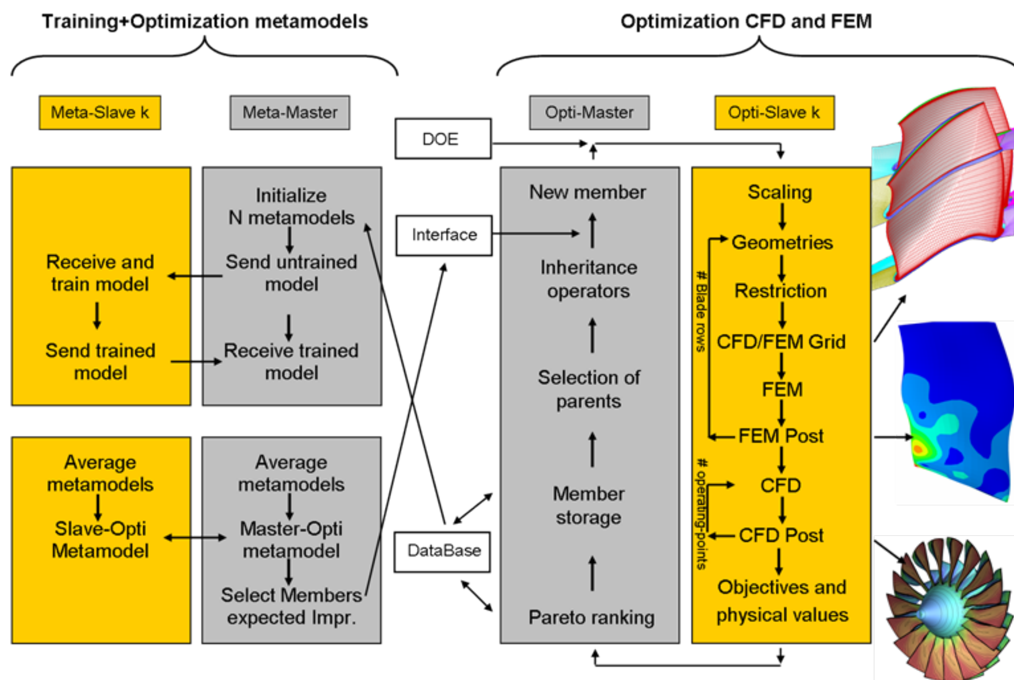

Abbildung 3.1: Prozesskette bei AutoOpti (Quelle: [17]).

Abbildung 3.1 zeigt die Struktur des parallelisierten Mehrziel-Evolutionsalgorithmus AutoOpti. Der *Opti-Master* führt den Optimierungsprozess durch. Zur (meist teuren) Berechnung der Fitness eines erzeugten Individuums übergibt der Master das Individuum an einen *Opti-Slave*. Nach dem Prozessdurchlauf erhält das Individuum seinen Fitnesswert und wird in der Datenbank (*DataBase*) gespeichert. Daraufhin werden die Pareto-Ränge aller in der Datenbank gespeicherten Individuen angepasst. Nun werden aus der Datenbank einige Eltern aufgrund ihrer Fitness und des geringen Pareto-Rangs ausgewählt, um mit Hilfe obiger Evolutionsoperatoren neue Kinder zu erzeugen. Die Fitness dieser Kinder wird dann wiederum in einem Slave-Prozess berechnet. Diese Vorgehensweise wird so lange fortgeführt bis der Benutzer (Ingenieur) anhand seiner Fachkenntnisse erkennt, zu welchem Zeitpunkt er den Prozess abbrechen kann. Ein allgemeines Abbruchkriterium ist nicht implementiert.
Da sich im Fall von AutoOpti ein enormer numerischer Aufwand hinter der Prozesskette des Designs verbirgt (CFD und FEM in mehreren Punkten), wurde hier viel Wert darauf gelegt, dass möglichst wenige Individuen diese Prozesskette durchlaufen müssen. Um dies zu ermöglichen, sind einige Erweiterungen in AutoOpti integriert. Durch diese Erweiterungen unterscheidet sich AutoOpti von herkömmlichen Optimierungsprogrammen.

AutoOpti ist nicht populationsbasiert, d.h. die jeweiligen Eltern werden nicht aus der vorherigen Generation sondern aus der Menge aller bisher erzeugten Individuen ausgewählt. Somit ist eine asynchrone Kommunikation zwischen dem Opti-Master und den Slaves möglich (normalerweise muss der Opti-Master mit der Erzeugung einer neuen Population warten, bis auch der langsamste Slave seine Berechnung beendet hat).
Die Datenbank ist essentiell für die Restart-Option und die Erstellung von Metamodellen (siehe un-

ten). Durch diese Restart-Option wird dem Benutzer eine Änderung/Anpassung der Optionen wie auch der Fitnessfunktion *online* (d.h. während des Optimierungsprozesses) ermöglicht. Nach einer solchen Änderung kann AutoOpti auf die Datenbank zurückgreifen; die Berechnungen müssen also nicht von neuem starten. Ein weiterer Vorteil der Datenbank ist die Möglichkeit der Beobachtung des Optimierungsvorgangs durch den Benutzer. Unstimmigkeiten oder eine Optimierung „in die falsche Richtung“ können frühzeitig erkannt und durch oben erwähnte Anpassung bereinigt werden.
Ein großer Nachteil von Evolutionsalgorithmen ist deren langsame Konvergenz. Vor allem für Probleme mit teurer Funktionsauswertung resultiert daraus eine große Rechenzeit. Um dieses Problem zu reduzieren (siehe nächster Absatz) und den Optimierungsprozess aus Sicht des Benutzers kontrollierbarer zu gestalten, wurde ein Interface (siehe Abbildung 3.1) implementiert. Dadurch ist es während des Optimierungsprozesses möglich, neue Individuen, welche dann vorrangig behandelt werden, in die Optimierung zu integrieren. Diese neuen Individuen können z.B. durch externe Algorithmen (siehe unten) oder durch den Benutzer (mit Hilfe seiner Fachkenntnisse) konstruiert worden sein.
Zur Beschleunigung und Verbesserung des Optimierungsprozesses werden in AutoOpti einige Approximations-Modelle, sogenannte Metamodelle (siehe Abbildung 3.1), verwendet. Während der ursprüngliche Optimierungsprozess (Abbildung 3.1, rechte Seite) arbeitet, läuft parallel dazu ein zweiter Prozess (linke Seite). Dieser beschäftigt sich mit dem Training von Metamodellen und der Optimierung auf diesen. Eine Kommunikation beider Prozesse findet über die Datenbank und das Interface statt. Ein Metamodell wird mit Hilfe vorher berechneter Lösungen (aus der Datenbank) erstellt und dazu verwendet, die Fitness eines beliebigen Individuums vorherzusagen. Bei der Optimierung auf den Metamodellen findet wiederum ein Evolutionsalgorithmus Anwendung. Die mit Hilfe der Metamodelle erzeugten Individuen können nun über das Interface in den ursprünglichen Optimierungsalgorithmus integriert werden.
Die noch verbleibende Frage beschäftigt sich mit dem Training der Metamodelle. Aufgrund der hohen Dimension des Suchraums (großen Anzahl an freien Parametern) und der geringen Anzahl an verfügbaren Test-Punkten gestaltet sich eine Interpolation der Werte schwierig. Die hierfür in AutoOpti verwendeten Modelle sind Kriging-Modelle und Neuronale Netze. Nähere Informationen hierzu finden sich in [18, 19].

4 DesParO

DesParO (**Des**ign **Par**ameters **O**ptimization, siehe [20, 21]) ist eine am Fraunhofer-Institut für Algorithmen und Wissenschaftliches Rechnen (SCAI) entwickelte Software zur interaktiven und robusten Mehrziel-Optimierung. Die Gruppe *Robust Design* unter der Leitung von Dr. Tanja Clees beschäftigt sich mit der Weiterentwicklung und Wartung von DesParO.
DesParO bietet neben einer Sensibilitätsanalyse und robuster Mehrzieloptimierung auch eine intuitiv handhabbare graphische Benutzeroberfläche zur interaktiven Erkundung des Designraumes (vgl. Abbildung 4.1). Um dies in angemessener Zeit realisieren zu können, wird eine Approximation mit Metamodellen vorgenommen. Ein weiterer Vorteil von DesParO ist die Möglichkeit, mit einer geringen Anzahl an Simulationen sehr gute Konfigurationen zu erzeugen. Daher eignet sich diese Software insbesondere für Probleme mit teurer Funktionsauswertung, wie dies beispielsweise bei Simulationen der Fall ist. DesParO stellt, genau wie AutoOpti, keine weiteren Voraussetzungen an die Zielfunktionen.

Da DesParO mit verschiedensten Simulationsprogrammen und Datensätzen aus physikalischen Experimenten arbeiten kann, ist es in den unterschiedlichsten Bereichen einsetzbar. Ursprünglich wurde DesParO zur Optimierung in der Automobilindustrie (z.B. für Crash-Simulationen) eingesetzt. Heutzutage beschäftigt sich die Gruppe Robust Design u.a. auch mit Anwendungen aus den Bereichen Strömungslehre, Elektrochemie, Halbleitertechnologie sowie Öl- und Gas-Reservoirs.

4.1 Softwaretechnischer Hintergrund von DesParO

DesParO ist in C++ geschrieben; die graphische Benutzeroberfläche wurde in QT (und OpenGL) programmiert. Erhältlich ist DesParO als alleinstehende Anwendung mit graphischer Benutzeroberfläche

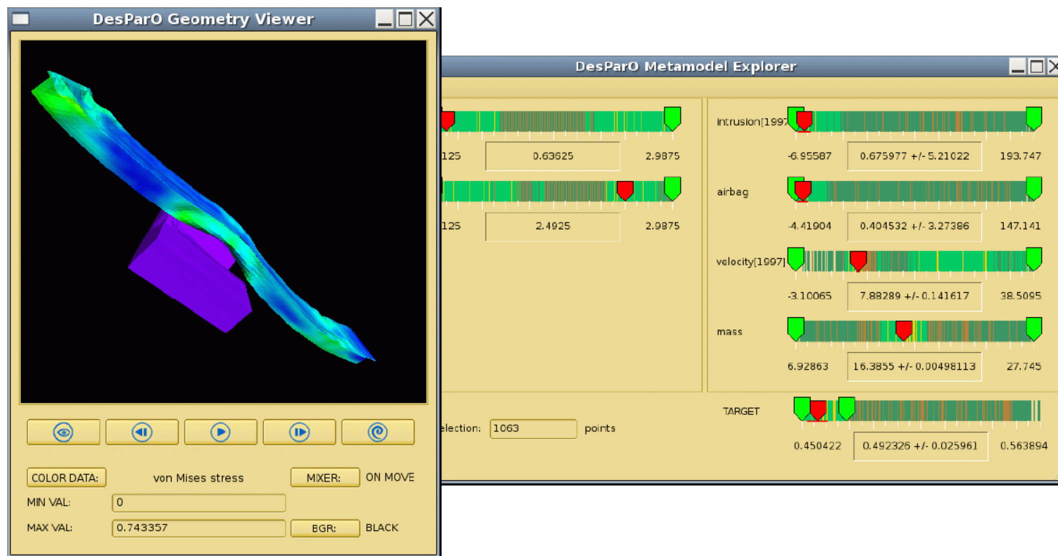

Abbildung 4.1: DesParO GUI nach einer Optimierung auf Metamodellen und anschließender Vorschau des neuen Designs (Quelle: [21]).

für Windows- und Linux-Architekturen oder als Software Development Kit (SDK) inklusive Dokumentation. Letzteres ist zur Integration in Workflow-Tools, andere Optimierer, eigene Benutzer-Software oder Simulatoren gedacht. Hierbei ist DesParO über die Kommandozeile und Steuerdateien ansprechbar. Als Input erhält der Optimierer dann Dateien im Tabellenformat. DesParO arbeitet nicht parallel. Die Lizensierung von DesParO wird mit Hilfe von *FLEXlm* gesteuert.
Da aufgrund der Beschaffenheit der Anwendungsprobleme mit sehr großen Datenmengen gearbeitet wird, findet eine Kompression mit FEMZIP [22] statt. Diese Art der Kompression wurde intern noch erweitert, da bei den Berechnungen viele ähnliche Datensätze entstehen und diese sich gut kombinieren und komprimieren lassen.

4.2 Grundzüge des Algorithmus von DesParO

DesParO basiert auf dem Prinzip der globalen Suche mit lokaler Nachoptimierung. Globale Suche bedeutet, dass der gesamte Designraum nach guten Kandidaten durchsucht wird. Durch die gewählte Implementierung ist diese globale Suche jedoch nicht teuer.
Die anschließende Optimierung wird zur Beschleunigung auf Metamodellen durchgeführt. Die hier verwendeten Metamodelle entstehen durch Interpolation mittels Radialer-Basis-Funktionen [23]. Deren Potenz ist nicht, wie sonst teilweise üblich, beschränkt. Somit sind beliebige Interpolationen ohne Einschränkung durchführbar.
Somit ist es DesParO möglich, eine Menge von globalen Optima zu finden. Als Ergebnis erhält der Benutzer, analog zu AutoOpti, eine Menge von Punkten, welche Teilmenge der Pareto-Menge ist. Diese Ausgabe kann bei wenigen Parametern visualisiert werden; ansonsten erhält der Benutzer eine Liste mit Pareto-Punkten. Desweiteren kann eine Visualisierung auch bei einer großen Anzahl von Parametern stattfinden, wenn viele dieser Paramter *fest* sind oder nur die wichtigsten Designparameter visualisiert werden sollen.

DesParO ist für ca. zwei bis 25 Designparameter ausgelegt (üblich für die Automobilindustrie sind z.Z. etwa 10 Parameter). Je mehr Designparameter vorhanden sind, desto hochdimensionaler ist der

Suchraum. Die Anzahl der benötigten Simulationen für eine gute Approximation mittels Metamodellen erhöht sich somit erheblich. Hier ist es sinnvoll, die Anzahl der Paramter z.B. durch Zusammenfassen vor der Optimierung zu reduzieren.
Zudem verwendet DesParO eine Korrelationsanalyse zur Entdeckung von Zusammenhängen zwischen den Parametern und den Restriktionen. Dies verkleinert den Suchraum.
Bei der Programmierung von DesParO wurde großer Wert auf Robustheit gelegt. Die gefundenen Pareto-Optima sollen keine „Spitzen“ sein, sondern auch noch in einer gewissen Umgebung sehr gute Funktionswerte annehmen (*soft spheres*). Dies ist vor allem in der Praxis wichtig, da die Herstellung eines Produktes nie exakt möglich sein wird. Realisiert wird diese robuste Optimierung, indem nicht, wie üblich, erst optimiert und anschließend die Umgebung betrachtet wird, sondern dieses Kriterium direkt in das Optimierungsproblem eingebaut wird.

Bei DesParO ist es dem Benutzer nicht möglich, für einzelne Parameter des Optimierungsalgorithmus Einstellungen vorzunehmen; es wird mit Defaulteinstellungen gearbeitet. Somit vereinfacht sich die Benutzung des Programms, allerdings gehen auch Möglichkeiten der Anpassung des Algorithmus an die spezielle Problemstellung verloren.

4.3 Graphische Benutzeroberfläche von DesParO

Abbildung 4.1 zeigt die graphische Benutzeroberfläche von DesParO. Hier sind die Parameter und die Restriktionen an die Zielfunktionen per Schieberegler einstellbar. Nach einer solchen Änderung in der GUI findet sofort eine Angleichung der möglichen Bereiche der Parameter statt (aufgrund der Restriktionen nicht mehr erreichbare Bereiche sind dann grau unterlegt). Anschließend findet eine interaktive Funktionsauswertung (mit Hilfe der oben erwähnten Metamodelle) statt. Somit erhält der Benutzer nach der Änderung seiner Einstellungen sofort ein Rückmeldung des Programms.

Der Optimierungsprozess in DesParO durchläuft bei Verwendung der graphischen Benutzeroberfläche die folgenden drei Schritte (vgl. Abbildung 4.2):
Im ersten Schritt werden die gewünschten Bereiche für die Paramter und Restriktionen festgelegt. Anschließend können verschiedene Alternativen durch Verschieben der Regler getestet werden. So kann der Parameterraum erkundet werden. Im letzten Schritt werden dann die verschiedenen Alternativen verglichen, und man entscheidet sich für das bestmögliche Design.

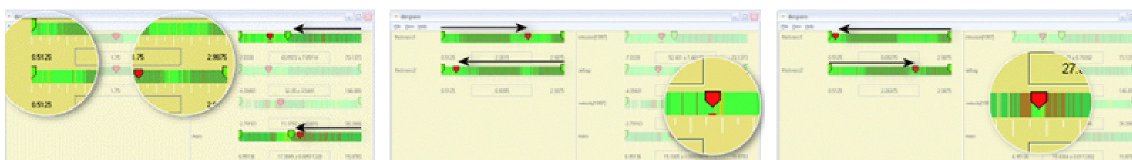

Abbildung 4.2: Lösung eines Optimierungsproblems mit Hilfe der DesParO GUI in drei Schritten (Quelle: [20]).

5 MOPS

MOPS (**M**ulti-**O**bjective **P**arameter **S**ynthesis, vgl. [24, 25]) ist eine am DLR-Institut für Robotik und Mechatronik in Oberpfaffenhofen entwickelte integrierte optimierungsbasierte Entwurfsumgebung für mehrzielige, parametrische Analyse und Synthese. Sie wurde für sogenannte *multi-objective*/*multi-model*/*multi-case*-Design-Probleme entworfen. Nähere Informationen hierzu finden sich im Abschnitt 5.2.

MOPS bietet eine grundlegende Kriterien-Bibliothek, eine allgemeine Mehr-Modell-Struktur (*multi-model*) für mehrzielige Probleme und eine allgemeine Mehr-Fall-Struktur (*multi-case*) für robustes Design. Abschätzungen über die Robustheit des Problems werden über eine *worst-case-search*-Optimierung, eine Monte-Carlo-Analyse oder einfache Parameter-Studien vorgenommen.
Zur Lösung des zugrundeliegenden Optimierungsproblems kann der Benutzer auf verschiedene leistungsstarke Algorithmen zurückgreifen. Darüber hinaus ist eine Visualisierung des Design-Prozesses möglich.

Zur Zeit wird MOPS für unterschiedliche Design- und Auswertungsprobleme im DLR und der Industrie genutzt. Das Anwendungsspektrum erstreckt sich von industriellen Robotern und der Flugkontrolle über leistungsoptimierte Flugsysteme bis hin zur Fahrzeugdynamik.

5.1 Softwaretechnischer Hintergrund von MOPS

MOPS ist vollständig in Matlab programmiert. Dem Benutzer wird ein Interface zur Programmanwendung und eine graphische Benutzeroberfläche (siehe Abbildung 5.1) zur Verfügung gestellt. Zudem bietet MOPS eine Online-Visualisierung, die eine Durchsicht aller Optimierungsiterationen und eine Auswahl einzelner Optimierungsergebnisse ermöglicht.
Eine Parallelisierung findet basierend auf PVM (Parallel Virtual Machine) und einer freien PM-Toolbox (Parallel-Matlab-Toolbox) statt. Dabei sichert MOPS automatisch die Synchronisation der parallelen Prozesse.

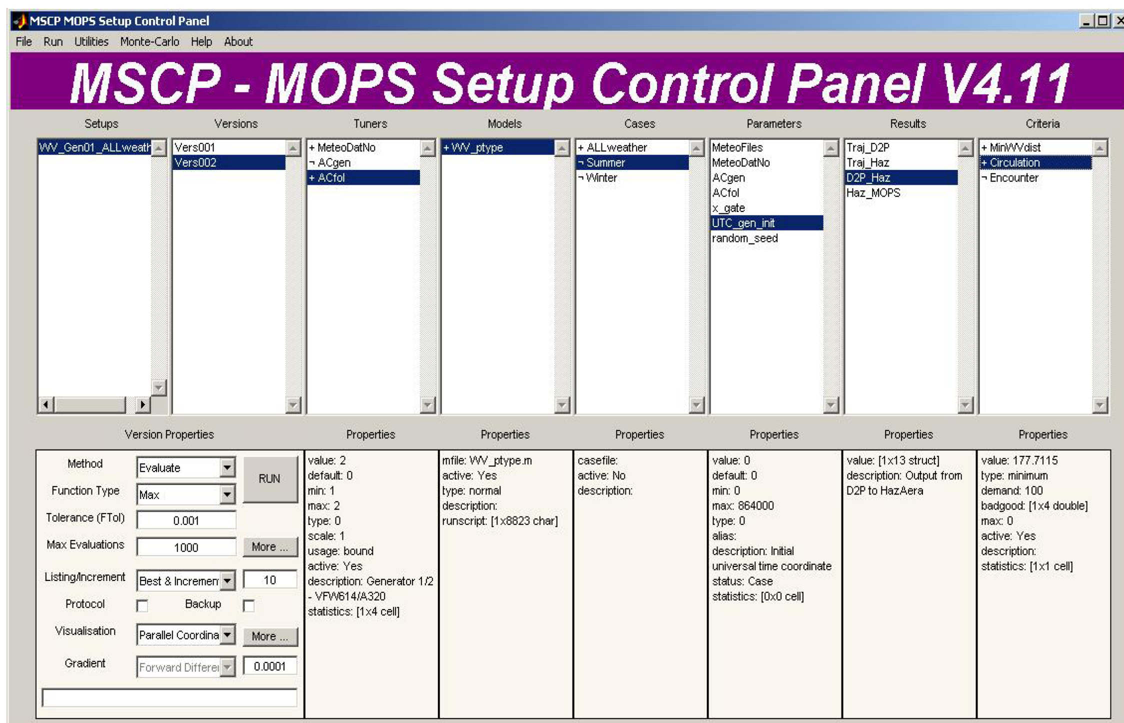

Abbildung 5.1: Graphische Benutzerschnittstelle von MOPS (Quelle: [26]).

Abbildung 5.2 zeigt die MOPS-Software-Architektur, bestehend aus verschiedenen Software-Ebenen. Zur Lösung des eigentlichen Optimierungsproblems (4.1) in Abschnitt 5.2 kommunizieren die Löser- und die MEX-Interface-Ebene über IPC (*Inter-Process Communication*). Die API-Ebene (*Application Program Interface*) stellt ein Benutzer-Interface zu den zwei grundlegegenden Ebenen dar. Hier können die Probleme dann einfach definiert und gelöst werden.

Die Löser-Ebene enthält eine Kollektion von Lösern für nichtlineare Probleme. Weitere Löser können hier einfach eingebunden werden. In der MEX-Interface-Ebene wird das *Min-Max*-Optimierungsproblem durch Aufruf eines Lösers gelöst. Die API-Ebene bietet eine Menge von Basisfunktionen, die in Matlab-Skripten und *m*-files sowie für die Kommandoeingabe zur Definition und Lösung des Problems genutzt werden können (siehe [27]). Die Auswertung der Kriterien muss in sogenannten *model-run*-Skripten formuliert werden. Diese m-file-Skripte definieren das Interface zwischen MOPS und dem speziellen Modell zur Funktionsauswertung.

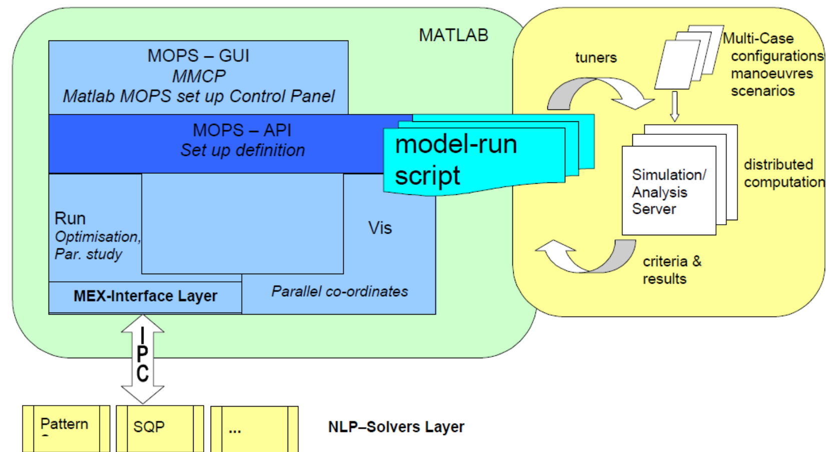

Abbildung 5.2: Software-Architektur von MOPS (Quelle: [27]).

5.2 Algorithmus von MOPS

Wie schon erwähnt wurde MOPS zur Lösung von *multi-objective*/*multi-model*/*multi-case*-Design-Problemen entwickelt. Um welche Art von Problemen es sich hierbei handelt, werden wir anhand eines Beispiels näher erläutern (vgl. [26]). Wir betrachten als Anwendung von MOPS lastabmindernde Regelungssysteme für Großraumflugzeuge (*Active Loads Control*, ALC; vgl. Abbildung 5.3). Belastungen der Struktur treten hier u.a. durch Manöver, Böen und Fehlerfälle (bspw. durch einen Triebwerksausfall) auf. Ziel ist eine Reduktion der Belastung unter Beibehaltung der Flugeigenschaften. Dabei soll das Biegemoment an der Wurzel des Höhenleitwerks um mindestens 15% reduziert werden; die „Dutch-Roll"-Starrkörperbewegung soll nur minimal verändert werden.

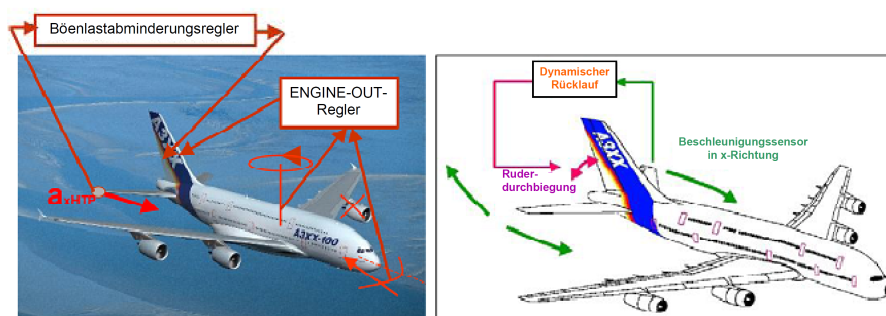

Abbildung 5.3: MOPS: ALC-Entwurf (Quelle: [26]).

Die zu beachtenden Kriterien (*multi-criteria*) sind hier u.a. die auftretenden Kräfte und Momente an verschiedenen Punkten, Manövrierfähigkeit und Stabilität. Hierbei muss das Problem in unterschiedlichen Fällen (*multi-cases*), z.B. unter verschiedenen Flugbedingungen und Beladungen, betrachtet werden. Die Kriterien sind hier jedoch in jedem Fall gleich. Nun stehen mehrere Berechnungsmodelle zur Funktionsauswertung (*multi-model*) zur Auswahl: Die Berechnung der entstehenden Frequenz mit instationären Kräften für Belastungskriterien oder eine stationäre lineare Approximation und Eigenwertberechnung für Stabilitätskriterien. Für die verschiedenen Berechnungsmodelle können hier unterschiedliche Kriterien betrachtet werden.
MOPS unterstützt gerade diese *multi-model*- und *multi-case*-Ansätze.

Gelöst werden die auftretenden *multi-objective*/*multi-model*/*multi-case*-Design-Probleme durch Betrachtung eines gewichteten *Min-Max*-Optimierungsproblems. Hierzu ist eine vorherige Normalisierung der Kriterien (z.B. durch Skalierung und Verschiebung) notwendig. Diese wird von MOPS automatisch durchgeführt. Jedes Kriterium c wird durch einen zugehörigen Wert d gewichtet. Wir erhalten das normalisierte Kriterium q mit $q := c/d$. Der Wert d skaliert die Kriterien, sodass für $q = 1$ das Kriterium befriedigend erfüllt ist ($q \leq 1$: Kriterium ist befriedigend erfüllt; $q > 1$: Kriterium ist nicht befriedigend erfüllt).
Abbildung 5.4 zeigt am Beispiel eines Problems mit zwei Kriterien, wann ein Kriterium befriedigend erfüllt ist (*set of satisfactory solutions*) und wie d_1 und d_2 gewählt werden. Als Lösung erhalten wir ein Pareto-Optimum c^*, welches einen Kompromiss zwischen den beiden Kriterien darstellt.

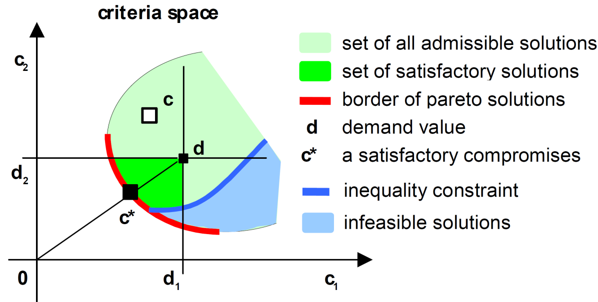

Abbildung 5.4: Min-Max-Mehrziel-Optimierung mit zwei Kriterien (Quelle: [26]).

Wir können also unser Optimierungsproblem, wie schon oben angedeutet, schreiben als

$$(4.1)\quad \begin{cases} \min\limits_{T} \max\limits_{ijk \in S_m} \frac{c_{ijk}(T, p_{ij})}{d_{ijk}} & \\ c_{ijk}(T, p_{ij}) \leq d_{ijk}, & ijk \in S_i \\ c_{ijk}(T, p_{ij}) = d_{ijk}, & ijk \in S_e \\ T_{min,l} \leq T_l \leq T_{max,l} & \end{cases} .$$

Hierbei bezeichnen wir mit ijk das k-te Kriterium im j-ten Fall des i-ten Modells. Weiter sei S_m die Menge der zu minimierenden Kriterien, S_i die Menge der Kriterien mit Ungleichungsrestriktionen und analog S_e die Menge der Kriterien mit Gleichungsrestriktionen. T bezeichnet die Menge der zu optimierenden Parameter und p_{ij} die Menge der Parameter des i-ten Modells, welches den j-ten Fall definiert. Die Zugehörigkeit eines Kriteriums zu der jeweiligen Menge S_m, S_i, S_e kann in jedem Schritt variiert werden. Dies liefert hohe Flexibilität während des Designprozesses.

(4.1) lässt sich nun leicht in ein *normales* nichtlineares Problem umformulieren. Zur Lösung dieses nichtlinearen Problems bietet MOPS eine Vielzahl von Algorithmen an. Neben effizienten gradientenba-

sierten Lösern (gut geeignet für glatte Probleme), stehen weniger effiziente, aber robuste gradienten-freie Löser (basierend auf direkter Suche) zur Verfügung. Um das Problem der lokalen Minimierung zu umgehen, hat MOPS u.a. auch statistische Methoden und genetische Algorithmen implementiert.

Beachte, dass MOPS (im Gegensatz zu AutoOpti oder DesParO) aufgrund der obigen Umformulierung des Problems immer nur ein einziges Pareto-Optimum, welches einen Kompromiss zwischen allen Kriterien darstellt, als Lösung ausgibt. Dies stellt einen Vorteil für unerfahrene Benutzer dar, die hier nicht zwischen verschiedenen Pareto-Optima *ihr* Optimum wählen müssen. Andererseits schränkt es vor allem erfahrene Benutzer in ihrer Freiheit, aus allen berechneten Pareto-Optima wählen zu können, ein.

Literatur

[1] Wächter, Andreas ; Biegler, Lorenz T.: On the Implementation of a Primal-Dual Interior Point Filter Line Search Algorithm for Large-Scale Nonlinear Programming. In: *Mathematical Programming* 106 (2006), S. 25–57

[2] Wächter, Andreas: *An Interior Point Algorithm for Large-Scale Nonlinear Optimization with Applications in Process Engineering*, Carnegie Mellon University, Diss., 2002

[3] Fourer, Robert ; Gay, David M. ; Kernighan, Brian W.: *AMPL: A Modeling Language for Mathematical Programming*. 2. Auflage. Brooks/Cole Publishing Company, 2002

[4] *AMPL Home Page*. `http://www.ampl.com/`, Abruf: 17. März 2010

[5] Laird, Carl ; Wächter, Andreas: *Introduction to IPOPT: A tutorial for downloading, installing, and using IPOPT*. November 2006. – Tutorial Revision Number 799

[6] Wächter, Andreas: *Short Tutorial: Getting Started With Ipopt in 90 Minutes*. April 2009. – IBM Research Report

[7] *IPOPT Home Page*. `http://www.coin-or.org/Ipopt/`, Abruf: 17. März 2010

[8] *Download HSL Subroutines*. `http://www.coin-or.org/Ipopt/documentation/node16.html`, Abruf: 17. März 2010

[9] *PARDISO Home Page*. `http://www.pardiso-project.org/`, Abruf: 17. März 2010

[10] *WSMP Home Page*. `http://www-users.cs.umn.edu/~agupta/wsmp.html`, Abruf: 17. März 2010

[11] *LAPACK Home Page*. `http://www.netlib.org/lapack/`, Abruf: 17. März 2010

[12] *BLAS Home Page*. `http://www.netlib.org/blas/`, Abruf: 17. März 2010

[13] Ehrgott, Matthias: *Multicriteria Optimization*. 2. Auflage. Springer Verlag, 2005

[14] Stöcker, Martin: *Untersuchung von Optimierungsverfahren für rechenzeitaufwändige technische Anwendungen in der Motorenentwicklung*, Technischen Universität Chemnitz, Diplomarbeit, 2007

[15] *Wikipedia: Pareto-Optimum*. `http://de.wikipedia.org/wiki/Pareto-Optimum`, Abruf: 22. März 2010

[16] *AutoOpti Home Page*. `http://www.dlr.de/at/desktopdefault.aspx/tabid-1529/2164_read-4771/`, Abruf: 19. März 2010

[17] Voss, Christian: *AutoOpti: Overview*. März 2010. – Vortragsfolien

[18] Siller, Ulrich ; Voss, Christian ; Nicke, Eberhard: Automated Multidisciplinary Optimization of a Transonic Axial Compressor. In: *47th AIAA Aerospace Sciences Meeting Including The New Horizons Forum and Aerospace Exposition*. Orlando, Florida, Januar 2009

[19] MacKay, David J. C.: *Bayesian Methods for Adaptive Models*, California Institute of Technology, Diss., 1992

[20] *DesParO Home Page*. `http://www.scai.fraunhofer.de/en/business-research-areas/numerical-software/products/desparo.html`, Abruf: 19. März 2010

[21] Clees, Tanja: *Robust Design - Overview*. März 2010. – Vortragsfolien zu DesParO

[22] *FEMZIP Home Page.* http://www.scai.fraunhofer.de/geschaeftsfelder/numerische-software/produkte/femzip.html, Abruf: 22. März 2010

[23] Kürz, Christoph: *Radiale-Basis-Funktionen.* http://www.scai.fraunhofer.de/fileadmin/ArbeitsgruppeTrottenberg/WS0809/seminar/Kuerz.pdf, Abruf: 17. März 2010. – Vortragsfolien

[24] Joos, Hans-Dieter: *MOPS - Multi-Objective Parameter Synthesis.* – User's Guide Version 5.5

[25] *MOPS Home Page.* http://www.dlr.de/rm/en/desktopdefault.aspx/tabid-3842/6343_read-9099/, Abruf: 19. März 2010

[26] Joos, Hans-Dieter ; Bals, Johann ; Looye, Gertjan ; Schnepper, Klaus ; Varga, Andras: *MOPS: Eine integrierte optimierungsbasierte Entwurfsumgebung für mehrzielige, parametrische Analyse und Synthese.* Version: 2008. http://www.dglr.de/veranstaltungen/archiv/2005_T5_1/Joos-Varga-MOPS_Entwurfsumgebung.pdf, Abruf: 19. März 2010. – Vortragsfolien

[27] Joos, Hans-Dieter ; Bals, Johann ; Looye, Gertjan ; Schnepper, Klaus ; Varga, Andras: A multi-objective optimisation-based software environment for control systems design. In: *IEEE International Symposium on Computer Aided Control System Design Proceedings.* Glasgow, Scotland, U.K., September 2002

Abbildungsverzeichnis

2.1 Pareto-Menge 5
3.1 Prozesskette bei AutoOpti 7
4.1 DesParO GUI 9
4.2 Optimierung mit Hilfe der DesParO GUI 10
5.1 MOPS GUI 11
5.2 Software-Architektur von MOPS 12
5.3 MOPS: ALC-Entwurf 12
5.4 Min-Max-Mehrziel-Optimierung mit zwei Kriterien 13

A Modellierung eines Beispiels in IPOPT

A.1 Das Beispiel

Wir betrachten ein einfaches Beispiel einer nichtlinearen Optimierungsaufgabe, um die verschiedenen Möglichkeiten der Problemmodellierung in IPOPT darzustellen. Bei dem Beispiel handelt es sich um das Testbeispiel aus dem IPOPT-Package [7]. Im Rahmen des Praktikums wurde das Beispiel in AMPL, C und C++ getestet. Desweiteren wurden Tests mit kleineren Aufgaben der nichlinearen Optimierung, wie in etwa der Rosenbrock-Funktion, mit Erfolg durchgeführt.

Betrachte also

$$\begin{cases} \min\limits_{x\in\mathbb{R}^4} & x^{(1)}x^{(4)}(x^{(1)}+x^{(2)}+x^{(3)})+x^{(3)} \\ \text{mit} & x^{(1)}x^{(2)}x^{(3)}x^{(4)} \geq 25 \\ & (x^{(1)})^2+(x^{(2)})^2+(x^{(3)})^2+(x^{(4)})^2 = 40 \\ & 1 \leq x \leq 5 \end{cases}$$

A.2 Modellierung in AMPL

A.2.1 hs71.mod

```
# Copyright (C) 2009, International Business Machines
#
# This file is part of the Ipopt open source package, published under
# the Common Public License
#
# Author:  Andreas Waechter         IBM       2009-04-03

# This is a model of Example 71 from
#
# Hock, W, and Schittkowski, K,
# Test Examples for Nonlinear Programming Codes,
# Lecture Notes in Economics and Mathematical Systems.
# Springer Verlag, 1981.

##############################################################################

# Definition of the variables with bounds
var x {i in 1..4}, >= 1, <= 5;

# objective function
minimize obj: x[1]*x[4]*(x[1] + x[2] + x[3]) + x[3];

# and the constraints
subject to c1: x[1]*x[2]*x[3]*x[4] >= 25;
subject to c2: x[1]^2+x[2]^2+x[3]^2+x[4]^2 = 40;

# Now we set the starting point:

let x[1] := 1;
let x[2] := 5;
let x[3] := 5;
let x[4] := 1;
```

A.3 Modellierung in C

A.3.1 hs071_c.c

```
/* Copyright (C) 2005, 2006 International Business Machines and others.
 * All Rights Reserved.
 * This code is published under the Common Public License.
 *
 * $Id: hs071_c.c 1324 2008-09-16 14:19:26Z andreasw $
 *
 * Authors:  Carl Laird, Andreas Waechter     IBM    2005-08-17
 */

#include "IpStdCInterface.h"
#include <stdlib.h>
#include <assert.h>
#include <stdio.h>

/* Function Declarations */
Bool eval_f(Index n, Number* x, Bool new_x,
            Number* obj_value, UserDataPtr user_data);

Bool eval_grad_f(Index n, Number* x, Bool new_x,
                 Number* grad_f, UserDataPtr user_data);

Bool eval_g(Index n, Number* x, Bool new_x,
            Index m, Number* g, UserDataPtr user_data);

Bool eval_jac_g(Index n, Number *x, Bool new_x,
                Index m, Index nele_jac,
                Index *iRow, Index *jCol, Number *values,
                UserDataPtr user_data);

Bool eval_h(Index n, Number *x, Bool new_x, Number obj_factor,
            Index m, Number *lambda, Bool new_lambda,
            Index nele_hess, Index *iRow, Index *jCol,
            Number *values, UserDataPtr user_data);

/* Main Program */
int main()
{
  Index n=-1;                          /* number of variables */
  Index m=-1;                          /* number of constraints */
  Number* x_L = NULL;                  /* lower bounds on x */
  Number* x_U = NULL;                  /* upper bounds on x */
  Number* g_L = NULL;                  /* lower bounds on g */
  Number* g_U = NULL;                  /* upper bounds on g */
  IpoptProblem nlp = NULL;             /* IpoptProblem */
  enum ApplicationReturnStatus status; /* Solve return code */
  Number* x = NULL;                    /* starting point and solution
   vector */
  Number* mult_x_L = NULL;             /* lower bound multipliers
      at the solution */
```

```
Number* mult_x_U = NULL;              /* upper bound multipliers
    at the solution */
Number obj;                           /* objective value */
Index i;                              /* generic counter */

/* Number of nonzeros in the Jacobian of the constraints */
Index nele_jac = 8;
/* Number of nonzeros in the Hessian of the Lagrangian (lower or
   upper triangual part only) */
Index nele_hess = 10;
/* indexing style for matrices */
Index index_style = 0; /* C-style; start counting of rows and column
     indices at 0 */

/* set the number of variables and allocate space for the bounds */
n=4;
x_L = (Number*)malloc(sizeof(Number)*n);
x_U = (Number*)malloc(sizeof(Number)*n);
/* set the values for the variable bounds */
for (i=0; i<n; i++) {
  x_L[i] = 1.0;
  x_U[i] = 5.0;
}

/* set the number of constraints and allocate space for the bounds */
m=2;
g_L = (Number*)malloc(sizeof(Number)*m);
g_U = (Number*)malloc(sizeof(Number)*m);
/* set the values of the constraint bounds */
g_L[0] = 25;
g_U[0] = 2e19;
g_L[1] = 40;
g_U[1] = 40;

/* create the IpoptProblem */
nlp = CreateIpoptProblem(n, x_L, x_U, m, g_L, g_U, nele_jac, nele_hess,
                         index_style, &eval_f, &eval_g, &eval_grad_f,
                         &eval_jac_g, &eval_h);

/* We can free the memory now - the values for the bounds have been
   copied internally in CreateIpoptProblem */
free(x_L);
free(x_U);
free(g_L);
free(g_U);

/* Set some options.  Note the following ones are only examples,
   they might not be suitable for your problem. */
AddIpoptNumOption(nlp, "tol", 1e-7);
AddIpoptStrOption(nlp, "mu_strategy", "adaptive");
AddIpoptStrOption(nlp, "output_file", "ipopt.out");
```

```
  /* allocate space for the initial point and set the values */
  x = (Number*)malloc(sizeof(Number)*n);
  x[0] = 1.0;
  x[1] = 5.0;
  x[2] = 5.0;
  x[3] = 1.0;

  /* allocate space to store the bound multipliers at the solution */
  mult_x_L = (Number*)malloc(sizeof(Number)*n);
  mult_x_U = (Number*)malloc(sizeof(Number)*n);

  /* solve the problem */
  status = IpoptSolve(nlp, x, NULL, &obj, NULL, mult_x_L, mult_x_U, NULL);

  if (status == Solve_Succeeded) {
    printf("\n\nSolution of the primal variables, x\n");
    for (i=0; i<n; i++) {
      printf("x[%d] = %e\n", i, x[i]);
    }

    printf("\n\nSolution of the bound multipliers, z_L and z_U\n");
    for (i=0; i<n; i++) {
      printf("z_L[%d] = %e\n", i, mult_x_L[i]);
    }
    for (i=0; i<n; i++) {
      printf("z_U[%d] = %e\n", i, mult_x_U[i]);
    }

    printf("\n\nObjective value\n");
    printf("f(x*) = %e\n", obj);
  }

  /* free allocated memory */
  FreeIpoptProblem(nlp);
  free(x);
  free(mult_x_L);
  free(mult_x_U);

  return 0;
}


/* Function Implementations */
Bool eval_f(Index n, Number* x, Bool new_x,
            Number* obj_value, UserDataPtr user_data)
{
  assert(n == 4);

  *obj_value = x[0] * x[3] * (x[0] + x[1] + x[2]) + x[2];

  return TRUE;
}
```

```
Bool eval_grad_f(Index n, Number* x, Bool new_x,
                 Number* grad_f, UserDataPtr user_data)
{
  assert(n == 4);

  grad_f[0] = x[0] * x[3] + x[3] * (x[0] + x[1] + x[2]);
  grad_f[1] = x[0] * x[3];
  grad_f[2] = x[0] * x[3] + 1;
  grad_f[3] = x[0] * (x[0] + x[1] + x[2]);

  return TRUE;
}

Bool eval_g(Index n, Number* x, Bool new_x,
            Index m, Number* g, UserDataPtr user_data)
{
  assert(n == 4);
  assert(m == 2);

  g[0] = x[0] * x[1] * x[2] * x[3];
  g[1] = x[0]*x[0] + x[1]*x[1] + x[2]*x[2] + x[3]*x[3];

  return TRUE;
}

Bool eval_jac_g(Index n, Number *x, Bool new_x,
                Index m, Index nele_jac,
                Index *iRow, Index *jCol, Number *values,
                UserDataPtr user_data)
{
  if (values == NULL) {
    /* return the structure of the jacobian */

    /* this particular jacobian is dense */
    iRow[0] = 0;
    jCol[0] = 0;
    iRow[1] = 0;
    jCol[1] = 1;
    iRow[2] = 0;
    jCol[2] = 2;
    iRow[3] = 0;
    jCol[3] = 3;
    iRow[4] = 1;
    jCol[4] = 0;
    iRow[5] = 1;
    jCol[5] = 1;
    iRow[6] = 1;
    jCol[6] = 2;
    iRow[7] = 1;
    jCol[7] = 3;
  }
```

```
  else {
    /* return the values of the jacobian of the constraints */

    values[0] = x[1]*x[2]*x[3]; /* 0,0 */
    values[1] = x[0]*x[2]*x[3]; /* 0,1 */
    values[2] = x[0]*x[1]*x[3]; /* 0,2 */
    values[3] = x[0]*x[1]*x[2]; /* 0,3 */

    values[4] = 2*x[0];         /* 1,0 */
    values[5] = 2*x[1];         /* 1,1 */
    values[6] = 2*x[2];         /* 1,2 */
    values[7] = 2*x[3];         /* 1,3 */
  }

  return TRUE;
}

Bool eval_h(Index n, Number *x, Bool new_x, Number obj_factor,
            Index m, Number *lambda, Bool new_lambda,
            Index nele_hess, Index *iRow, Index *jCol,
            Number *values, UserDataPtr user_data)
{
  Index idx = 0; /* nonzero element counter */
  Index row = 0; /* row counter for loop */
  Index col = 0; /* col counter for loop */
  if (values == NULL) {
    /* return the structure. This is a symmetric matrix, fill the lower left
     * triangle only. */

    /* the hessian for this problem is actually dense */
    idx=0;
    for (row = 0; row < 4; row++) {
      for (col = 0; col <= row; col++) {
        iRow[idx] = row;
        jCol[idx] = col;
        idx++;
      }
    }

    assert(idx == nele_hess);
  }
  else {
    /* return the values. This is a symmetric matrix, fill the lower left
     * triangle only */

    /* fill the objective portion */
    values[0] = obj_factor * (2*x[3]);                /* 0,0 */

    values[1] = obj_factor * (x[3]);                  /* 1,0 */
    values[2] = 0;                                    /* 1,1 */

    values[3] = obj_factor * (x[3]);                  /* 2,0 */
```

```
    values[4] = 0;                                   /* 2,1 */
    values[5] = 0;                                   /* 2,2 */

    values[6] = obj_factor * (2*x[0] + x[1] + x[2]); /* 3,0 */
    values[7] = obj_factor * (x[0]);                 /* 3,1 */
    values[8] = obj_factor * (x[0]);                 /* 3,2 */
    values[9] = 0;                                   /* 3,3 */


    /* add the portion for the first constraint */
    values[1] += lambda[0] * (x[2] * x[3]);          /* 1,0 */

    values[3] += lambda[0] * (x[1] * x[3]);          /* 2,0 */
    values[4] += lambda[0] * (x[0] * x[3]);          /* 2,1 */

    values[6] += lambda[0] * (x[1] * x[2]);          /* 3,0 */
    values[7] += lambda[0] * (x[0] * x[2]);          /* 3,1 */
    values[8] += lambda[0] * (x[0] * x[1]);          /* 3,2 */

    /* add the portion for the second constraint */
    values[0] += lambda[1] * 2;                      /* 0,0 */

    values[2] += lambda[1] * 2;                      /* 1,1 */

    values[5] += lambda[1] * 2;                      /* 2,2 */

    values[9] += lambda[1] * 2;                      /* 3,3 */
  }

  return TRUE;
}
```

A.4 Modellierung in C++

A.4.1 hs071_nlp.hpp

```
// Copyright (C) 2005, 2007 International Business Machines and others.
// All Rights Reserved.
// This code is published under the Common Public License.
//
// $Id: hs071_nlp.hpp 1324 2008-09-16 14:19:26Z andreasw $
//
// Authors:  Carl Laird, Andreas Waechter     IBM    2005-08-09

#ifndef __HS071_NLP_HPP__
#define __HS071_NLP_HPP__

#include "IpTNLP.hpp"

using namespace Ipopt;

/** C++ Example NLP for interfacing a problem with IPOPT.
```

```
 *  HS071_NLP implements a C++ example of problem 71 of the
 *  Hock-Schittkowski test suite. This example is designed to go
 *  along with the tutorial document and show how to interface
 *  with IPOPT through the TNLP interface.
 *
 * Problem hs071 looks like this
 *
 *     min   x1*x4*(x1 + x2 + x3)  +  x3
 *     s.t.  x1*x2*x3*x4                   >=  25
 *           x1**2 + x2**2 + x3**2 + x4**2  =  40
 *           1 <=  x1,x2,x3,x4  <= 5
 *
 *     Starting point:
 *        x = (1, 5, 5, 1)
 *
 *     Optimal solution:
 *        x = (1.00000000, 4.74299963, 3.82114998, 1.37940829)
 *
 *
 */
class HS071_NLP : public TNLP
{
public:
  /** default constructor */
  HS071_NLP();

  /** default destructor */
  virtual ~HS071_NLP();

  /**@name Overloaded from TNLP */
  //@{
  /** Method to return some info about the nlp */
  virtual bool get_nlp_info(Index& n, Index& m, Index& nnz_jac_g,
                            Index& nnz_h_lag,
                            IndexStyleEnum& index_style);

  /** Method to return the bounds for my problem */
  virtual bool get_bounds_info(Index n, Number* x_l, Number* x_u,
                               Index m, Number* g_l, Number* g_u);

  /** Method to return the starting point for the algorithm */
  virtual bool get_starting_point(Index n, bool init_x, Number* x,
                                  bool init_z, Number* z_L, Number* z_U,
                                  Index m, bool init_lambda,
                                  Number* lambda);

  /** Method to return the objective value */
  virtual bool eval_f(Index n, const Number* x, bool new_x,
   Number& obj_value);

  /** Method to return the gradient of the objective */
  virtual bool eval_grad_f(Index n, const Number* x, bool new_x,
```

```
    Number* grad_f);

  /** Method to return the constraint residuals */
  virtual bool eval_g(Index n, const Number* x, bool new_x,
   Index m, Number* g);

  /** Method to return:
   *   1) The structure of the jacobian (if "values" is NULL)
   *   2) The values of the jacobian (if "values" is not NULL)
   */
  virtual bool eval_jac_g(Index n, const Number* x, bool new_x,
                          Index m, Index nele_jac, Index* iRow,
                          Index *jCol, Number* values);

  /** Method to return:
   *   1) The structure of the hessian of the lagrangian
   * (if "values" is NULL)
   *   2) The values of the hessian of the lagrangian
   * (if "values" is not NULL)
   */
  virtual bool eval_h(Index n, const Number* x, bool new_x,
                      Number obj_factor, Index m, const Number* lambda,
                      bool new_lambda, Index nele_hess, Index* iRow,
                      Index* jCol, Number* values);

  //@}

  /** @name Solution Methods */
  //@{
  /** This method is called when the algorithm is complete so the TNLP
  */ can store/write the solution */
  virtual void finalize_solution(SolverReturn status,
          Index n, const Number* x, const Number* z_L, const Number* z_U,
            Index m, const Number* g, const Number* lambda,
            Number obj_value,
 const IpoptData* ip_data,
 IpoptCalculatedQuantities* ip_cq);
  //@}

private:
  /**@name Methods to block default compiler methods.
   * The compiler automatically generates the following three methods.
   *  Since the default compiler implementation is generally not what
   *  you want (for all but the most simple classes), we usually
   *  put the declarations of these methods in the private section
   *  and never implement them. This prevents the compiler from
   *  implementing an incorrect "default" behavior without us
   *  knowing. (See Scott Meyers book, "Effective C++")
   *
   */
  //@{
  //  HS071_NLP();
```

```
  HS071_NLP(const HS071_NLP&);
  HS071_NLP& operator=(const HS071_NLP&);
  //@}
};


#endif
```

A.4.2 hs071_main.cpp

```
// Copyright (C) 2005, 2009 International Business Machines and others.
// All Rights Reserved.
// This code is published under the Common Public License.
//
// $Id: hs071_main.cpp 1597 2009-10-29 15:23:18Z andreasw $
//
// Authors:  Carl Laird, Andreas Waechter     IBM    2005-08-10

#include "IpIpoptApplication.hpp"
#include "hs071_nlp.hpp"

// for printf
#ifdef HAVE_CSTDIO
# include <cstdio>
#else
# ifdef HAVE_STDIO_H
#  include <stdio.h>
# else
#  error "don't have header file for stdio"
# endif
#endif

using namespace Ipopt;

int main(int argv, char* argc[])
{
  // Create a new instance of your nlp
  //  (use a SmartPtr, not raw)
  SmartPtr<TNLP> mynlp = new HS071_NLP();

  // Create a new instance of IpoptApplication
  //  (use a SmartPtr, not raw)
  // We are using the factory, since this allows us to compile this
  // example with an Ipopt Windows DLL
  SmartPtr<IpoptApplication> app = IpoptApplicationFactory();

  // Change some options
  // Note: The following choices are only examples, they might not be
  //       suitable for your optimization problem.
  app->Options()->SetNumericValue("tol", 1e-7);
  app->Options()->SetStringValue("mu_strategy", "adaptive");
  app->Options()->SetStringValue("output_file", "ipopt.out");
```

```
  // The following overwrites the default name (ipopt.opt) of the
  // options file
  // app->Options()->SetStringValue("option_file_name", "hs071.opt");

  // Intialize the IpoptApplication and process the options
  ApplicationReturnStatus status;
  status = app->Initialize();
  if (status != Solve_Succeeded) {
    printf("\n\n*** Error during initialization!\n");
    return (int) status;
  }

  // Ask Ipopt to solve the problem
  status = app->OptimizeTNLP(mynlp);

  if (status == Solve_Succeeded) {
    printf("\n\n*** The problem solved!\n");
  }
  else {
    printf("\n\n*** The problem FAILED!\n");
  }

  // As the SmartPtrs go out of scope, the reference count
  // will be decremented and the objects will automatically
  // be deleted.

  return (int) status;
}
```

A.4.3 hs071_nlp.cpp

```
// Copyright (C) 2005, 2006 International Business Machines and others.
// All Rights Reserved.
// This code is published under the Common Public License.
//
// $Id: hs071_nlp.cpp 1324 2008-09-16 14:19:26Z andreasw $
//
// Authors:  Carl Laird, Andreas Waechter     IBM    2005-08-16

#include "hs071_nlp.hpp"

// for printf
#ifdef HAVE_CSTDIO
# include <cstdio>
#else
# ifdef HAVE_STDIO_H
#  include <stdio.h>
# else
#  error "don't have header file for stdio"
# endif
#endif
```

```
using namespace Ipopt;

// constructor
HS071_NLP::HS071_NLP()
{}

//destructor
HS071_NLP::~HS071_NLP()
{}

// returns the size of the problem
bool HS071_NLP::get_nlp_info(Index& n, Index& m, Index& nnz_jac_g,
                             Index& nnz_h_lag,
                             IndexStyleEnum& index_style)
{
  // The problem described in HS071_NLP.hpp has 4 variables,
  // x[0] through x[3]
  n = 4;

  // one equality constraint and one inequality constraint
  m = 2;

  // in this example the jacobian is dense and contains 8 nonzeros
  nnz_jac_g = 8;

  // the hessian is also dense and has 16 total nonzeros, but we
  // only need the lower left corner (since it is symmetric)
  nnz_h_lag = 10;

  // use the C style indexing (0-based)
  index_style = TNLP::C_STYLE;

  return true;
}

// returns the variable bounds
bool HS071_NLP::get_bounds_info(Index n, Number* x_l, Number* x_u,
                                Index m, Number* g_l, Number* g_u)
{
  // here, the n and m we gave IPOPT in get_nlp_info are passed
  // back to us.
  // If desired, we could assert to make sure they are what we
  // think they are.
  assert(n == 4);
  assert(m == 2);

  // the variables have lower bounds of 1
  for (Index i=0; i<4; i++) {
    x_l[i] = 1.0;
  }

  // the variables have upper bounds of 5
```

```
  for (Index i=0; i<4; i++) {
    x_u[i] = 5.0;
  }

  // the first constraint g1 has a lower bound of 25
  g_l[0] = 25;
  // the first constraint g1 has NO upper bound, here we set it
  // to 2e19.
  // Ipopt interprets any number greater than
  // nlp_upper_bound_inf as infinity. The default value of
  // nlp_upper_bound_inf and nlp_lower_bound_inf
  // is 1e19 and can be changed through ipopt options.
  g_u[0] = 2e19;

  // the second constraint g2 is an equality constraint, so we set the
  // upper and lower bound to the same value
  g_l[1] = g_u[1] = 40.0;

  return true;
}

// returns the initial point for the problem
bool HS071_NLP::get_starting_point(Index n, bool init_x, Number* x,
                                   bool init_z, Number* z_L, Number* z_U,
                                   Index m, bool init_lambda,
                                   Number* lambda)
{
  // Here, we assume we only have starting values for x, if you code
  // your own NLP, you can provide starting values for the dual variables
  // if you wish
  assert(init_x == true);
  assert(init_z == false);
  assert(init_lambda == false);

  // initialize to the given starting point
  x[0] = 1.0;
  x[1] = 5.0;
  x[2] = 5.0;
  x[3] = 1.0;

  return true;
}

// returns the value of the objective function
bool HS071_NLP::eval_f(Index n, const Number* x, bool new_x,
 Number& obj_value)
{
  assert(n == 4);

  obj_value = x[0] * x[3] * (x[0] + x[1] + x[2]) + x[2];

  return true;
```

```
}

// return the gradient of the objective function grad_{x} f(x)
bool HS071_NLP::eval_grad_f(Index n, const Number* x, bool new_x,
Number* grad_f)
{
  assert(n == 4);

  grad_f[0] = x[0] * x[3] + x[3] * (x[0] + x[1] + x[2]);
  grad_f[1] = x[0] * x[3];
  grad_f[2] = x[0] * x[3] + 1;
  grad_f[3] = x[0] * (x[0] + x[1] + x[2]);

  return true;
}

// return the value of the constraints: g(x)
bool HS071_NLP::eval_g(Index n, const Number* x, bool new_x,
 Index m, Number* g)
{
  assert(n == 4);
  assert(m == 2);

  g[0] = x[0] * x[1] * x[2] * x[3];
  g[1] = x[0]*x[0] + x[1]*x[1] + x[2]*x[2] + x[3]*x[3];

  return true;
}

// return the structure or values of the jacobian
bool HS071_NLP::eval_jac_g(Index n, const Number* x, bool new_x,
                           Index m, Index nele_jac, Index* iRow,
                           Index *jCol, Number* values)
{
  if (values == NULL) {
    // return the structure of the jacobian

    // this particular jacobian is dense
    iRow[0] = 0;
    jCol[0] = 0;
    iRow[1] = 0;
    jCol[1] = 1;
    iRow[2] = 0;
    jCol[2] = 2;
    iRow[3] = 0;
    jCol[3] = 3;
    iRow[4] = 1;
    jCol[4] = 0;
    iRow[5] = 1;
    jCol[5] = 1;
    iRow[6] = 1;
    jCol[6] = 2;
```

```
    iRow[7] = 1;
    jCol[7] = 3;
  }
  else {
    // return the values of the jacobian of the constraints

    values[0] = x[1]*x[2]*x[3]; // 0,0
    values[1] = x[0]*x[2]*x[3]; // 0,1
    values[2] = x[0]*x[1]*x[3]; // 0,2
    values[3] = x[0]*x[1]*x[2]; // 0,3

    values[4] = 2*x[0]; // 1,0
    values[5] = 2*x[1]; // 1,1
    values[6] = 2*x[2]; // 1,2
    values[7] = 2*x[3]; // 1,3
  }

  return true;
}

//return the structure or values of the hessian
bool HS071_NLP::eval_h(Index n, const Number* x, bool new_x,
                       Number obj_factor, Index m, const Number* lambda,
                       bool new_lambda, Index nele_hess, Index* iRow,
                       Index* jCol, Number* values)
{
  if (values == NULL) {
    // return the structure. This is a symmetric matrix,
    // fill the lower left triangle only.

    // the hessian for this problem is actually dense
    Index idx=0;
    for (Index row = 0; row < 4; row++) {
      for (Index col = 0; col <= row; col++) {
        iRow[idx] = row;
        jCol[idx] = col;
        idx++;
      }
    }

    assert(idx == nele_hess);
  }
  else {
    // return the values. This is a symmetric matrix, fill the lower left
    // triangle only

    // fill the objective portion
    values[0] = obj_factor * (2*x[3]); // 0,0

    values[1] = obj_factor * (x[3]);   // 1,0
    values[2] = 0.;                    // 1,1
```

```
    values[3] = obj_factor * (x[3]);   // 2,0
    values[4] = 0.;                    // 2,1
    values[5] = 0.;                    // 2,2

    values[6] = obj_factor * (2*x[0] + x[1] + x[2]); // 3,0
    values[7] = obj_factor * (x[0]);                 // 3,1
    values[8] = obj_factor * (x[0]);                 // 3,2
    values[9] = 0.;                                  // 3,3


    // add the portion for the first constraint
    values[1] += lambda[0] * (x[2] * x[3]); // 1,0

    values[3] += lambda[0] * (x[1] * x[3]); // 2,0
    values[4] += lambda[0] * (x[0] * x[3]); // 2,1

    values[6] += lambda[0] * (x[1] * x[2]); // 3,0
    values[7] += lambda[0] * (x[0] * x[2]); // 3,1
    values[8] += lambda[0] * (x[0] * x[1]); // 3,2

    // add the portion for the second constraint
    values[0] += lambda[1] * 2; // 0,0

    values[2] += lambda[1] * 2; // 1,1

    values[5] += lambda[1] * 2; // 2,2

    values[9] += lambda[1] * 2; // 3,3
  }

  return true;
}

void HS071_NLP::finalize_solution(SolverReturn status,
                  Index n, const Number* x, const Number* z_L,
                  const Number* z_U, Index m, const Number* g,
                  const Number* lambda, Number obj_value,
                  const IpoptData* ip_data,
                  IpoptCalculatedQuantities* ip_cq)
{
  // here is where we would store the solution to variables, or
  // write to a file, etc
  // so we could use the solution.

  // For this example, we write the solution to the console
  printf("\n\nSolution of the primal variables, x\n");
  for (Index i=0; i<n; i++) {
    printf("x[%d] = %e\n", i, x[i]);
  }

  printf("\n\nSolution of the bound multipliers, z_L and z_U\n");
  for (Index i=0; i<n; i++) {
```

```
    printf("z_L[%d] = %e\n", i, z_L[i]);
  }
  for (Index i=0; i<n; i++) {
    printf("z_U[%d] = %e\n", i, z_U[i]);
  }

  printf("\n\nObjective value\n");
  printf("f(x*) = %e\n", obj_value);

  printf("\nFinal value of the constraints:\n");
  for (Index i=0; i<m ;i++) {
    printf("g(%d) = %e\n", i, g[i]);
  }
}
```

A.5 Modellierung in Fortran

A.5.1 hs071_f.f

```
C Copyright (C) 2002, 2007 Carnegie Mellon University and others.
C All Rights Reserved.
C This code is published under the Common Public License.
C
C    $Id: hs071_f.f.in 991 2007-06-09 07:20:55Z andreasw $
C
C =============================================================================
C
C     This is an example for the usage of IPOPT.
C     It implements problem 71 from the Hock-Schittkowski test suite:
C
C     min   x1*x4*(x1 + x2 + x3)  +  x3
C     s.t.  x1*x2*x3*x4                   >=  25
C           x1**2 + x2**2 + x3**2 + x4**2  =  40
C           1 <=  x1,x2,x3,x4  <= 5
C
C     Starting point:
C        x = (1, 5, 5, 1)
C
C     Optimal solution:
C        x = (1.00000000, 4.74299963, 3.82114998, 1.37940829)
C
C =============================================================================
C
C
C =============================================================================
C
C                            Main driver program
C
C =============================================================================
C
      program example
C
```

```
      implicit none
C
C     include the Ipopt return codes
C
      include 'IpReturnCodes.inc'
C
C     Size of the problem (number of variables and equality constraints)
C
      integer     N,     M,     NELE_JAC,     NELE_HESS,      IDX_STY
      parameter  (N = 4, M = 2, NELE_JAC = 8, NELE_HESS = 10)
      parameter  (IDX_STY = 1 )
C
C     Space for multipliers and constraints
C
      double precision LAM(M)
      double precision G(M)
C
C     Vector of variables
C
      double precision X(N)
C
C     Vector of lower and upper bounds
C
      double precision X_L(N), X_U(N), Z_L(N), Z_U(N)
      double precision G_L(M), G_U(M)
C
C     Private data for evaluation routines
C     This could be used to pass double precision and integer arrays
C untouched to the evaluation subroutines EVAL_*
C
      double precision DAT(2)
      integer IDAT(1)
C
C     Place for storing the Ipopt Problem Handle
C
CC     for 32 bit platforms
C      integer IPROBLEM
C      integer IPCREATE
C     for 64 bit platforms:
      integer*8 IPROBLEM
      integer*8 IPCREATE
C
      integer IERR
      integer IPSOLVE, IPADDSTROPTION
      integer IPADDNUMOPTION, IPADDINTOPTION
      integer IPOPENOUTPUTFILE
C
      double precision F
      integer i
C
C     The following are the Fortran routines for computing the model
C     functions and their derivatives - their code can be found furhter
```

```
C     down in this file.
C
      external EV_F, EV_G, EV_GRAD_F, EV_JAC_G, EV_HESS
C
C     Set initial point and bounds:
C
      data X   / 1d0, 5d0, 5d0, 1d0/
      data X_L / 1d0, 1d0, 1d0, 1d0 /
      data X_U / 5d0, 5d0, 5d0, 5d0 /
C
C     Set bounds for the constraints
C
      data G_L / 25d0, 40d0 /
      data G_U / 1d40, 40d0 /
C
C     First create a handle for the Ipopt problem (and read the options
C     file)
C
      IPROBLEM = IPCREATE(N, X_L, X_U, M, G_L, G_U, NELE_JAC, NELE_HESS,
     1     IDX_STY, EV_F, EV_G, EV_GRAD_F, EV_JAC_G, EV_HESS)
      if (IPROBLEM.eq.0) then
         write(*,*) 'Error creating an Ipopt Problem handle.'
         stop
      endif
C
C     Open an output file
C
      IERR = IPOPENOUTPUTFILE(IPROBLEM, 'IPOPT.OUT', 5)
      if (IERR.ne.0 ) then
         write(*,*) 'Error opening the Ipopt output file.'
         goto 9000
      endif
C
C     Note: The following options are only examples, they might not be
C           suitable for your optimization problem.
C
C     Set a string option
C
      IERR = IPADDSTROPTION(IPROBLEM, 'mu_strategy', 'adaptive')
      if (IERR.ne.0 ) goto 9990
C
C     Set an integer option
C
      IERR = IPADDINTOPTION(IPROBLEM, 'max_iter', 3000)
      if (IERR.ne.0 ) goto 9990
C
C     Set a double precision option
C
      IERR = IPADDNUMOPTION(IPROBLEM, 'tol', 1.d-7)
      if (IERR.ne.0 ) goto 9990
C
C     As a simple example, we pass the constants in the constraints to
```

```
C     the EVAL_C routine via the "private" DAT array.
C
      DAT(1) = 0.d0
      DAT(2) = 0.d0
C
C     Call optimization routine
C
      IERR = IPSOLVE(IPROBLEM, X, G, F, LAM, Z_L, Z_U, IDAT, DAT)
C
C     Output:
C
      if( IERR.eq.IP_SOLVE_SUCCEEDED ) then
         write(*,*)
         write(*,*) 'The solution was found.'
         write(*,*)
         write(*,*) 'The final value of the objective function is ',F
         write(*,*)
         write(*,*) 'The optimal values of X are:'
         write(*,*)
         do i = 1, N
            write(*,*) 'X  (',i,') = ',X(i)
         enddo
         write(*,*)
         write(*,*) 'The multipliers for the lower bounds are:'
         write(*,*)
         do i = 1, N
            write(*,*) 'Z_L(',i,') = ',Z_L(i)
         enddo
         write(*,*)
         write(*,*) 'The multipliers for the upper bounds are:'
         write(*,*)
         do i = 1, N
            write(*,*) 'Z_U(',i,') = ',Z_U(i)
         enddo
         write(*,*)
         write(*,*) 'The multipliers for the equality constraints are:'
         write(*,*)
         do i = 1, M
            write(*,*) 'LAM(',i,') = ',LAM(i)
         enddo
         write(*,*)
      else
         write(*,*)
         write(*,*) 'An error occoured.'
         write(*,*) 'The error code is ',IERR
         write(*,*)
      endif
C
 9000 continue
C
C     Clean up
C
```

```
      call IPFREE(IPROBLEM)
      stop
C
 9990 continue
      write(*,*) 'Error setting an option'
      goto 9000
      end
C
C =============================================================================
C
C                    Computation of objective function
C
C =============================================================================
C
      subroutine EV_F(N, X, NEW_X, F, IDAT, DAT, IERR)
      implicit none
      integer N, NEW_X
      double precision F, X(N)
      double precision DAT(*)
      integer IDAT(*)
      integer IERR
      F = X(1)*X(4)*(X(1)+X(2)+X(3)) + X(3)
      IERR = 0
      return
      end
C
C =============================================================================
C
C                Computation of gradient of objective function
C
C =============================================================================
C
      subroutine EV_GRAD_F(N, X, NEW_X, GRAD, IDAT, DAT, IERR)
      implicit none
      integer N, NEW_X
      double precision GRAD(N), X(N)
      double precision DAT(*)
      integer IDAT(*)
      integer IERR
      GRAD(1) = X(4)*(2d0*X(1)+X(2)+X(3))
      GRAD(2) = X(1)*X(4)
      GRAD(3) = X(1)*X(4) + 1d0
      GRAD(4) = X(1)*(X(1)+X(2)+X(3))
      IERR = 0
      return
      end
C
C =============================================================================
C
C                     Computation of equality constraints
C
C =============================================================================
```

```
C
      subroutine EV_G(N, X, NEW_X, M, G, IDAT, DAT, IERR)
      implicit none
      integer N, NEW_X, M
      double precision G(M), X(N)
      double precision DAT(*)
      integer IDAT(*)
      integer IERR
      G(1) = X(1)*X(2)*X(3)*X(4) - DAT(1)
      G(2) = X(1)**2 + X(2)**2 + X(3)**2 + X(4)**2 - DAT(2)
      IERR = 0
      return
      end
C
C =============================================================================
C
C                Computation of Jacobian of equality constraints
C
C =============================================================================
C
      subroutine EV_JAC_G(TASK, N, X, NEW_X, M, NZ, ACON, AVAR, A,
     1     IDAT, DAT, IERR)
      integer TASK, N, NEW_X, M, NZ
      double precision X(N), A(NZ)
      integer ACON(NZ), AVAR(NZ), I
      double precision DAT(*)
      integer IDAT(*)
      integer IERR
C
C     structure of Jacobian:
C
      integer AVAR1(8), ACON1(8)
      data  AVAR1 /1, 2, 3, 4, 1, 2, 3, 4/
      data  ACON1 /1, 1, 1, 1, 2, 2, 2, 2/
      save  AVAR1, ACON1
C
      if( TASK.eq.0 ) then
        do I = 1, 8
          AVAR(I) = AVAR1(I)
          ACON(I) = ACON1(I)
        enddo
      else
        A(1) = X(2)*X(3)*X(4)
        A(2) = X(1)*X(3)*X(4)
        A(3) = X(1)*X(2)*X(4)
        A(4) = X(1)*X(2)*X(3)
        A(5) = 2d0*X(1)
        A(6) = 2d0*X(2)
        A(7) = 2d0*X(3)
        A(8) = 2d0*X(4)
      endif
      IERR = 0
```

```
      return
      end
C
C =============================================================================
C
C                Computation of Hessian of Lagrangian
C
C =============================================================================
C
      subroutine EV_HESS(TASK, N, X, NEW_X, OBJFACT, M, LAM, NEW_LAM,
     1     NNZH, IRNH, ICNH, HESS, IDAT, DAT, IERR)
      implicit none
      integer TASK, N, NEW_X, M, NEW_LAM, NNZH, i
      double precision X(N), OBJFACT, LAM(M), HESS(NNZH)
      integer IRNH(NNZH), ICNH(NNZH)
      double precision DAT(*)
      integer IDAT(*)
      integer IERR
C
C     structure of Hessian:
C
      integer IRNH1(10), ICNH1(10)
      data  IRNH1 /1, 2, 2, 3, 3, 3, 4, 4, 4, 4/
      data  ICNH1 /1, 1, 2, 1, 2, 3, 1, 2, 3, 4/
      save  IRNH1, ICNH1

      if( TASK.eq.0 ) then
         do i = 1, 10
            IRNH(i) = IRNH1(i)
            ICNH(i) = ICNH1(i)
         enddo
      else
         do i = 1, 10
            HESS(i) = 0d0
         enddo
C
C     objective function
C
         HESS(1) = OBJFACT * 2d0*X(4)
         HESS(2) = OBJFACT * X(4)
         HESS(4) = OBJFACT * X(4)
         HESS(7) = OBJFACT * (2d0*X(1) + X(2) + X(3))
         HESS(8) = OBJFACT * X(1)
         HESS(9) = OBJFACT * X(1)
C
C     first constraint
C
         HESS(2) = HESS(2) + LAM(1) * X(3)*X(4)
         HESS(4) = HESS(4) + LAM(1) * X(2)*X(4)
         HESS(5) = HESS(5) + LAM(1) * X(1)*X(4)
         HESS(7) = HESS(7) + LAM(1) * X(2)*X(3)
         HESS(8) = HESS(8) + LAM(1) * X(1)*X(3)
```

```
          HESS(9) = HESS(9) + LAM(1) * X(1)*X(2)
C
C     second constraint
C
          HESS(1) = HESS(1) + LAM(2) * 2d0
          HESS(3) = HESS(3) + LAM(2) * 2d0
          HESS(6) = HESS(6) + LAM(2) * 2d0
          HESS(10)= HESS(10)+ LAM(2) * 2d0
      endif
      IERR = 0
      return
      end
```