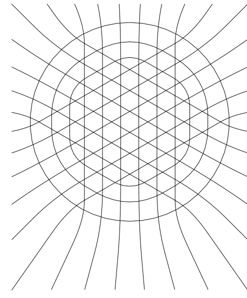

Konrad-Zuse-Zentrum f r Informationstechnik Berlin

U. Nowak U. Pöhle R. Roitzsch R. Werk

ZGUI Handbuch

Technical Report TR 96–10 (Dezember 1996)

`ZGUI` Handbuch

U. Nowak, U. Pöhle, R. Roitzsch, R. Werk

13. Februar 1997

Zusammenfassung

In diesem Handbuch werden die Bausteine zum Aufbau einer graphischen Benutzeroberfläche mit `ZGUI` beschrieben. Auf der einen Seite stehen die Tcl/Tk–Prozeduren, die die graphischen Elemente definieren. Die Beschreibung der Anwendung der Prozeduren und der Interaktionen der Elemente bildet den ersten Teil des Handbuches.

Auf der anderen Seite stehen die Anforderungen an Anwendungen, die mit einer `ZGUI`–Benutzeroberlfäche gesteuert werden sollen. Hier findet man im Handbuch die Beschreibung der Anwendungsprogrammierschnittstelle (application programming interface, API).

Inhaltsverzeichnis

1 `ZGUI`: Rahmen **2**
- 1.1 Initialisierung von `ZGUI` 3
- 1.2 Ressourcen 3
- 1.3 Ereignisse 4
- 1.4 Fensterverwaltung 4
- 1.5 Einrichten des Hauptfensters 4

2 `ZGUI`: Menüs **6**
- 2.1 Einrichten der Menüleiste 6
- 2.2 Hilfsroutine für das Example-Menü 7

3 `ZGUI`: Textfenster **8**

4 `ZGUI`: Optionen **9**
- 4.1 Die drei Ebenen 9
- 4.2 Optionsfenster 10
- 4.3 Hilfsprozeduren zum Aufbau von Widgets 11

5 ZGUI: Applikation starten **14**
5.1 tcl/tk–Schnittstelle 14
5.2 Ein Widget zur Kontrolle des Laufes eines Anwendungsprogramms 16

6 ZGUI: Datenvisualiserung **17**

7 Applikation: Parameter–Verwaltung **17**
7.1 Interne Datenstrukturen 18
7.2 Dateiformat 18
7.3 C Schnittstelle 18
7.4 C++ Schnittstelle 24
7.5 Fortran–Schnittstelle 25
7.6 Tcl Schnittstelle 27

8 Applikation: Ausgabeformatierung **28**
8.1 Haltepunkte 28
8.2 Ausgabe spezieller Daten 29
8.2.1 C Schnittstelle 30

1 ZGUI: Rahmen

Zum Aufbau einer graphischen Benutzeroberfläche für eine Anwendung werden in einer Tcl–Datei alle wichtigen Daten und die Aufrufe der Initialsierungsprozeduren zusammen gestellt. Diese Datei wird dem `ZGUI` als Parameter übergeben.

```
zgui -tcl kaskade.tcl
```

Die Initialisierungsprozedur hat folgende “Verantwortungen”:

- Festlegung des Pfades, nach dem Tcl–Prozeduren gesucht werden;
- Definition des Hauptfensters;
- Definition der Hauptmenüs, z.B. Example, Options;
- Kreation des Programmstart–Widgets;
- Initalisierung benutzereigener Funktionen.

Für die Programmierung der Initialisierungsdatei stehen Hilfsprogramme zu Verfügung, die im Folgendem beschrieben werden.

1.1 Initialisierung von ZGUI

`zuInitGUI pwdDir addDirs libDir pictDir` initialisiert `ZGUI`. Parameter sind das Dateiverzeichnis, in dem `ZGUI` ablaufen soll (normalerweise `pwd`), eine Folge weiterer Dateiverzeichnisse in denen Tcl–Prozeduren oder Resource–Dateien gesucht werden und schließlich das Verzeichnis, in dem die Standard–Tcl–Prozeduren des `ZGUI` aufbeahrt werden. Die Liste aller Pfade ist unter dem globalen Namen `ZG_DIRPATH` gespeichert. Ein zusätzliche Pfad zur Ablage von Bilder kann über den optional Parameter `pictDir` definiert werden.

```
zuInitGUI [pwd] {/home/neville/zgui/test} $env(ZGUI_LIB)
```

`zuPwd` liefert den Pfad aus, unter dem `ZGUI` gestartet wurde. Die information ist auch über die globale Variable `ZG_PWDDIR` verfügbar.

`zuApplDir` liefert den Pfad aus, unter dem die Anwendung gestartet wird. Die information ist auch über die globale Variable `ZG_APPLDIR` verfügbar.

`zuLibDir` liefert den Pfad aus, in dem die tcl–Dateien gefunden werden. Die information ist auch über die globale Variable `ZG_LIBDIR` verfügbar.

`zuPictDir` liefert den Pfad aus, unter dem Bilder gesucht werden. Die information ist auch über die globale Variable `ZG_PICTDIR` verfügbar.

`zuDirSep` liefert den maschinenabhängigen Seperator für die Teile eines Dateipfadnames aus.

`zuFindFP path name access` sucht eine Datei `name` in allen Verzeichnissen, die in `path` definiert sind und überprüft, ob die Datei im Modus `access` (`r` oder `w`) zu eröffnen ist. Ergebnis ist der vollqualifizierte Dateiname. Existiert keine Datei oder kann der Zugriff nicht erfüllt werden, ist das Egebnis `0`.

```
set fullName [zuFindFP $ZG_DIRPATH lists.tcl]
```

1.2 Ressourcen

Ressourcedateien enthalten Tcl-Programmstücke, in denen Variable gesetzt werden. Sie können per Hand erstellt oder automatisch generiert werden.

`zrReadResources filePattern` liest Dateien ein, deren Namen `filePattern` genügen.

`zrWriteResources arrayList fileName mode` schreibt die Arrayvariablen aus `arrayList` in die Datei `fileName`, so daß sie mit `zrReadResources` wieder eingelesen werden können.

1.3 Ereignisse

Beim Eintritt bestimmter Ereignisse wird aller Tcl–Code, der in einer Arrayvariablen hinterlegt ist, ausgefuehrt. Die Reihenfolge ist die alphabetische der Arrayindizes. So sorgt zum folgendes Programmstück dafür, daß am Ende ein Text ausgedruckt wird:

```
set ZR_AtExit(zzz) { puts "Schüß Leute" }
```

`zeInitEvent` initalisiert das Eventsystem. Die `exit` Prozedure wird umdefiniert und führt jetzt das Ereignis `ZR_AtExit` am Ende aus.

`zeDoEvent eventName` führt alle Programmstücke, die an dem Arrayname `eventName` definiert sind, aus.

1.4 Fensterverwaltung

Für Fenster, die mit den hier beschriebenen Prozeduren geöffnet werden, wird beim Schliessen das Ereigniss `ZW_AtClose` ausgeführt. Außerdem wird automatisch die Position des Fensters in einer Resource–Datei hinterlegt und bei einer Neueröffnung mit dem selben Pfadnamen, diese Position übernommen.

`zwOpenWindow pathName displayName iconName` eröffnet ein Fenster, gegebenfalls an der abgespeicherten Position.

`zwCloseWindow pathName` schließt das Fenster. Die aktuelle Postion wird abgespeichert.

`zwICloseAllWindows` schließt alle Fenster. Diese Routine wird am Ende des Programmes automatisch aufgerufen.

1.5 Einrichten des Hauptfensters

Das Hauptfenster hat das in Bild 1 angezeigte Layout.

`zuMakeMain appTitle winTitle winLogo helpText` richtet das Hauptfenster (siehe Bild 2) ein. Der erste Parameter definiert die Überschrift, gefolgt von einem Hilfetext und einer Datei, die das Bild eines Logos enthält. Innerhalb des Fensters stehen Bereiche für die Menüleiste, Hilfetexte und Startwidgets zur Verfügung. Die Namen sind in dem globalen Feld `ZG_AREAS` hinterlegt (siehe Tabelle 1).

<table>
<tr><td>Titel</td><td rowspan="4">Logo</td></tr>
<tr><td>Menüs</td></tr>
<tr><td>Anwendungssteuerung</td></tr>
<tr><td>Hilfe Texte</td></tr>
<tr><td>Quit</td><td>Ausgewähltes Beispiel</td></tr>
</table>

Abbildung 1: Layout des Hauptfensters

```
zuMakeMain "Kaskade 3.0" "ZIB GUI" ZIBlogo-color.ppm\
  "Kaskade: A toolbox for solving partial differential equations
     with finite element methods (1,2, or 3 space dimensions)
     Authors: Rudi Beck, Bodo Erdmann, Rainer Roitzsch
     mail: erdmann@zib-berlin.de roitzsch@zib-berlin.de"
```

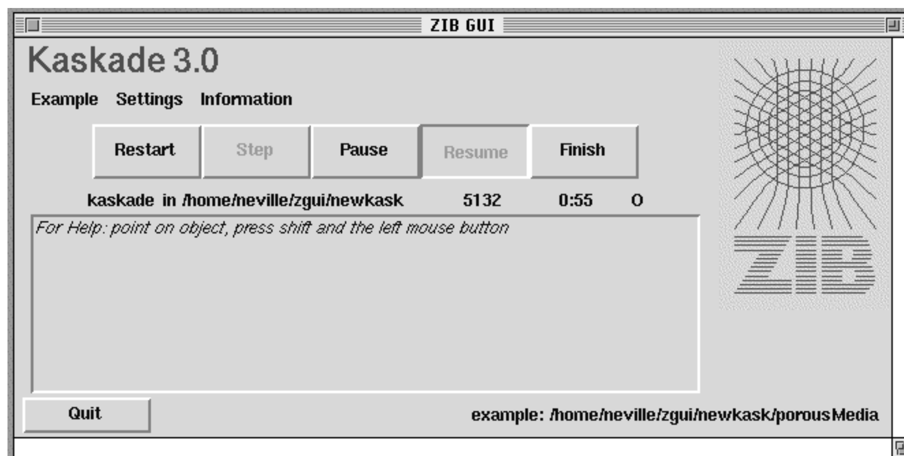

Abbildung 2: Hauptfenster mit Menüleiste und Programmkontrolle

`zuExit` beendet `ZGUI`. Es werden alle Fenster geschlossen und die letzten Positionen in der Datei `zgui.rc` im Startverzeichnis hinterlassen.

`zuSetExampleName name` setzt einen neuen Beispielnamen ein.

Index	Erläuterung
`MenuBar`	Menüleiste
`Run`	Startwidgets
`Help`	Hilfetexte
`ExName`	Beispielname

Tabelle 1: Bereiche für Widgets im Hauptfenster (Feld `ZG_AREAS`)

2 `ZGUI`: Menüs

Die folgenden Tcl–Prozeduren können zur Verwaltung der Menüleiste im Hauptfenster verwenden werden.

2.1 Einrichten der Menüleiste

Für die Gestaltung der Beispiel– und Optionen–Menüs (siehe Bild 3) stehen ein Reihe Tcl–Prozeduren zur Verfügung.

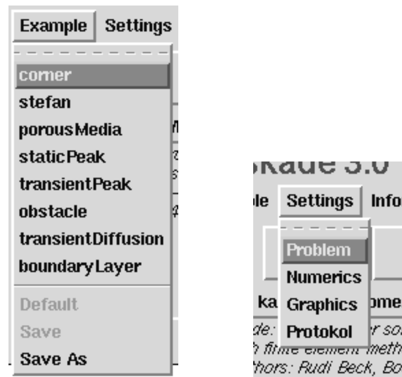

Abbildung 3: Beispiel– und Optionen–Menü

`zuAddMenuItem short name` fügt den Menüpunkt `name` ein. `short` definiert eine Kürzel für die Generierung eines Widget–Pfadnamens.

```
zuAddMenuItem ex Example
zuAddMenuItem opt Settings
```

`zuCreateFileMenu short directory extension readProc defaultExt` kreiert für alle Dateien aus dem Verzeichnis `directory`, deren Namen mit

`.extension` endet, für das Menü `short` Menüpunkte. Bei der Auswahl eines solchen Menüs wird die Benutzerroutine `readProc` mit dem entsprechenden Dateinamen ausgeführt. Die dazugehörigen Menüpunktnamen werden als Liste in `ZG_MENU_FILES($short)` gespeichert. `defaultExt` definiert die Dateinamenerweiterung, die Default–Dateien auszeichnet.

```
zuCreateFileMenu ex $ZG_APPLDIR ex zuReadExample "-default"
```

`zuSetFileMenu short mode` setzt den Status (`normel` bzw. `disabled` für alle Menüpunkte, die in durch `zuCreateFileMenu` entstanden sind.

`zuAddSave short saveproc saveasproc defaultproc` fügt die Menüpunkte `Save`, `Save As` und `Default` zum Menü `short` an.

```
zuAddSave ex zuSaveExample zuSaveAsExample zuDefaultExample
```

`zuAddQuit short quitProc` fügt ein `Quit` Menüpunkt zu.

`zuAddOptMenu short name userproc` fügt einen Menüpunkt `name` an `short` an. Bei der Auswahl des Menüpunktes wird die Benutzeprozedur `userproc` aufgerufen. Die Hilfsmittel zum Aufbau der Fenster zur Verwaltung von Optionen (Parameter) sind im Abschnitt 4 beschrieben.

```
zuAddOptMenu opt Problem zuUpdProblem
```

2.2 Hilfsroutine für das Example-Menü

Die folgenden Routinen können zur Abarbeitung der Beispiel–Menüpunkte verwendet werden.

`zuReadExample short name` liest die Parameterdatei `name`.

`zuSaveExample` sichert die lokale Parameterliste in die aktulle Beispieldatei.

`zuSaveAsExample` fragt den Benutzer nach einen Dateinamen und sichert die lokale Parameterliste in diese Datei.

`zuDefaultExample` kopiert die Voreinstellungsdatei auf die entsprechende Beispieldatei.

Außerdem werden alle Programmstücke, die in der Arrayvariablen `ZG_NewExample` hinterlegt sind, bei der Auswahl eines neuen Beispiels ausgeführt.

3 `ZGUI`: Textfenster

Text– und Optionsfenster haben ein ähnliches Layout (siehe Bild 4). Die verschieden Textfenster differieren in der Kontrolleiste und der Möglichkeit, den Text selbst zu ändern.

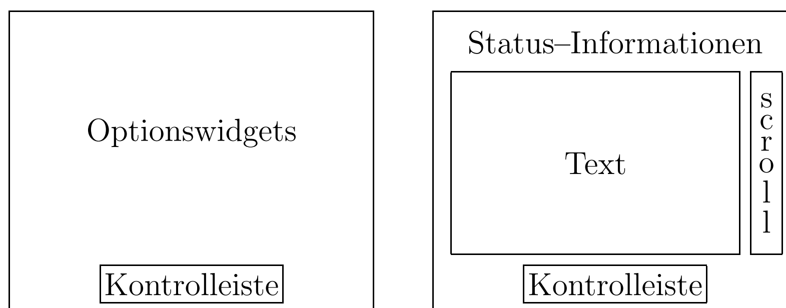

Abbildung 4: Options– und Textfenster

`zwEdit windowPath displayName iconName fileName` öffnet ein Textfenster, liest die Datei `fileName` in den Textbereich ein und definiert die Knöpfe `Dismiss`, `SaveAs` und `Save`.

`zwView windowPath displayName iconName fileName` öffnet ein Textfenster, liest die Datei `fileName` in den Textbereich ein, sperrt den Textbereich für Änderungen und definiert die Knöpfe `Dismiss` und `SaveAs`.

`zwArray windowPath displayName iconName arrName arrDisplayName` öffnet ein Textfenster, schreibt das Feld `arrName` in den Textbereich und definiert die Knöpfe `Dismiss`, `SaveAs`, `Default` und `SaveArray`. `Default` erstellt den ursprünglichen Inhalt des Textbereiches, `SaveArray` definert das Feld auf Grund des Textbereiches neu.

Diese beiden Routinen verwenden folgende Prozeduren:

`zwOpenTextWindow pathName displayName iconName writable changedProc` eröffnet ein Textfenster. `changedProc` wird aufgerufen, falls der Fensterinhalt geändert wurde.

`zwCloseTextWindow pathName` schließt ein Textfenster.

`zwHead windowPath changed tefileNamext readonly` ändert die Einträge in der Statuszeile. Bei einem leeren String bleibt der alte Text erhalten.

`zwControl windowPath args` fügt zu dem `Dismiss`-Knopf weitere Knöpfe hinzu. Zulässige Werte sind `Default`, `Clear`, `SaveAs` und `Save`.

`zwAddText windowPath text` fügt `text` ans Ende des Texbereiches ein.

`zwAddLine windowPath line` fügt `line` und ein Zeilenwechsel ans Ende des Texbereiches ein.

`zwClear windowPath` löscht den Textbereich.

`zwAddFile windowPath fileName` fügt den Inhalt der Datei `fileName` ans Ende des Texbereiches ein.

4 ZGUI: Optionen

4.1 Die drei Ebenen

Es gibt drei Ebenen, in denen Parameter gesetzt sein können. Die erste Ebene, die Voreinstellungsliste, enthält einen möglichst vollständigen Satz von Parametern mit den Voreinstellungen. Diese Parameterliste wird beim Start eingelesen (etwa aus der Datei `ss6.cmd` und ist über die globale Variable `PARAM_pl` verfügbar. Der Benutzer kann über ein Menü ein Beispiel auswählen, damit wird eine zweite Parameterlist (aus `name.ex` eingelesen. Sie ist über die tcl–Variable `PARAM_local` verfügbar.

Das `Settings`–Menü (siehe Bild 3) dient der Eröffnung von Fenstern, die einen Ausschnitt der Parameterliste sichtbar machen. Den Eingabefeldern, die dort sichtbar sind, werden tcl–Variable zugeordnet. Das ist die dritte Ebene.

Für die Programmierung mit Parameterlisten stehen die folgenden, einfachen Tcl–Prozeduren bereit.

`zuGetVal pList parName` holt einen Wert eines Parameters aus der Parameterliste.Im Fehlerfall liefert die Prozedur den leeren String zurück.

```
zuGetVal PARAM_local spaceDim
```

`zuGetNewestVal varName` ermittelt den Wert einer Variablen aus der lokalen Parameterliste, beziehungsweise falls er dort nicht definiert ist, aus der Voreinstellungsliste. Im Fehlerfall liefert die Prozedur den leeren String zurück.

```
zuGetNewestVal spaceDim
```

`zuSetNewestArray parName varName` besetzt das Feld `varName` mit den Werten des Parameterfeldes `parName` aus der lokalen Parameterliste beziehungsweise der Voreinstellungsliste.

4.2 Optionsfenster

Ein Optionsfenster ist eine Schnittstelle zum Ändern einer Teilmenge des vollen Parametersatzes. Es wird über einen `Settings`–Menüpunkt geöffnet. Es enthält außer der Titelzeile mindestens eine Reihe von Knöpfen zum Verwalten der drei Parameterebenen (siehe Bild 4):

Apply die geänderten Parameter werden in die lokale Parameterliste übernommen;

Reset die Änderungen werden auf den Ausgangszustand zurückgesetzt;

Default es werden die Werte der Voreinstellungsliste gezeigt;

Cancel das Fenster wird geschlossen, eventuell angezeigte Änderungen werden ignoriert und

Apply&Close die geänderten Parameter werden in die lokale Parameterliste übernommen und das Fenster wird geschlossen.

Zur Implementierung benötigt man eine Zuordnungsliste von Parameternamen und Tcl–Variablennamen, die die dritte Ebene definieren. Um die Konsistenz (und den Programmierkomfort) zu sichern, stehen Tcl–Prozeduren zur Verwaltung der Informationen zur Verfügung. Der Benutzer hat "lediglich" das Layout der Fenster und die Zuordnung der Namen, Texte, und Widgetpfade festzulegen.

Mit den folgenden Prozeduren kann ein Optionsfenster und die oben beschriebene Knopfleiste erstellt werden.

`zuOptWindOpen name displayName menuName` eröffnet ein (toplevel) Fenster unter dem Pfadname `name`. Der Fenstertitel wird durch den 2. Parameter definert, der Name des Icons durch den Namen des entsprechenden Menü–Punktes. Die Position des Fensters ergibt sich aus der zuletzt verwendeten. Mit dem öffnen des Fensters wird der entsprechende `Settings`–Menü–Punkt entschärft.

```
zuOptWindOpen .problem "Problem definition" "Problem"
```

`zuApplyBar path onClose` definiert die Knopfleiste relative zum Widgetpfad `path`. Die Benutzerprozedur `onClose` wird beim Schliessen des Fensters aufgerufen.

```
zuApplyBar .problem.apply ProblemClose
```

Damit liegt der Rahmen für eine Routine, die über einen `Settings`–Menüpunkt aufgerufen wird, fest:

```
proc KASK_UpdProblem {} {
  zuOptWindOpen .problem "Problem definition" "Problem"
  frame .problem.apply
  pack .problem.apply
  zuApplyBar .problem.apply ProblemClose
}
```

4.3 Hilfsprozeduren zum Aufbau von Widgets

Die folgenden tcl–Prozeduren verwalten für die häufig auftretetenden Fällen tk–Unterwidgets und die zugehörigen Parameter– und Variablenlisten. Alle `path`–Parameter sind Widget–relative angaben, etwa `.problem.type`

`zuUpdNumber path parName displayName varName widt default helpText`

generiert ein Unterwidget relativ zu `path`. `parName` gibt den Namen des Parameters in den Paramerlisten der ersten und zweiten Ebene an, `displayName` die Beschriftung des Eingabefelds in dem Optionsfenster und `varName` den tcl–Namen an. Zusätzlich kann für die Programmierung von Konsistenzüberprüfungen eine Bedingung für die Zulässigkeit des Wertes der Variable angegeben werden.

```
zuUpdNumber .graph.lev.lev plotLevels "Anzahl" \
            solLevels 10 10 "Number of plot levels"
```

Das Bild 5 zeigt das optische Ergebnis.

Abbildung 5: Zahleneingabe

`zuUpdAlternatives path nlist parName displayName varName direction default helpT`

generiert ein Unterwidget zu Eingabe von Alternativen. Das sind Parameter, für die ein Satz Schlüsselworte erlaubt sind. Die Darstellung wird über einen vertikal angeordneten Satz “Radioknöpfe” gemacht, siehe Bild 6. Die Liste der Schlüsselworte `nlist` wird durch eine Folge von Trippel, die sich aus einem Kürzel, dem Parameterwert und einem Beschriftungsnamen zusammensetzen, gebildet. Das Kürzel dient zur Benennung des Pfades und muß mit einem Kleinbuchstaben beginnen.

```
set problemList { \
  {heat StaticHeatConduction StaticHeatConduction "Wärmeleitung"} \
  {trans TransientHeatConduction TransientHeatConduction \
   "Zeitabhängige Wärmeleitung"} \
  {por PorousMedia PorousMedia "Poröses Medium"} \
  {obst Obstacle Obstacle "Hindernisproblem"} \
  {stefan Stefan Stefan "Stefan-Problem""} \
}
zuUpdAlternatives .problem.mbar.prob $problemList problem \
                  Type ProbVar "Problemauswahl"
```

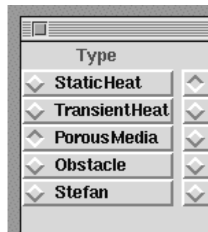

Abbildung 6: Alternativen

`zuUpdFeatures path nlist displayName direction default helpText`
generiert ein Unterwidget zur Auswahl einer Reihe von Leistungen. Die Darstellung erfolget über einen vertikal angeordenten Satz von Knöpfen, siehe Bild 7. Die Liste `nlist` der dazugehörigen logischen Parameter besteht aus einer Folge von Quadrupel, die sich aus einem Kürzel, einem Parameternamen, einem Beschriftungsnamen und einem tcl–Variablennamen zusammen setzt, gebildet. Das Kürzel dient zur Benennung des Pfades und muß mit einem Kleinbuchstaben beginnen.

```
set graphList { \
   {sol plotSolution Loesung solGraphVar "Lösung zeichnen"} \
  {time plotTimeStep ZeitSchritt timeGraphVar \
   "Nur nach jedem Zeitschritt zeichnen"} \
  {keep plotKeep "Neues Fenster" keepGraphVar \
    "Für jede Zeichnung ein neues Fenster öffnen"} \
}
zuUpdFeatures .graph.opt.todo $graphList Leistungen "Graphikoptionen"
```

`zuUpdFileMenu path pat parName displayName varName ok helpText` generiert ein Subwidget mit einem Knopf zu Eröffnen eines Dateisuchdialogs und zeigt den aktuell gewählten Dateinamen an. `pat` beschreibt

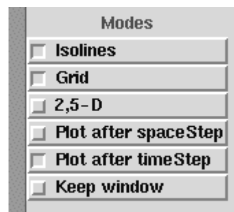

Abbildung 7: Auswahl

die Zu Auswahl anstehenden Dateien (etwa `.geo`. Falls eine neue Datei gewählt wird, wird die Prozedure `ok` mit dem vollqualifizierten Dateinamen aufgerufen.

```
zuUpdFileName .problem.file *.geo file "Geometry" fName \
              CheckDimension "Geometriebeschreibungsdatei auswählen"
```

Abbildung 8: Dateiauswahl

`zuUpdArray path arrayName displayName varName helpText` eröffnet ein neues Fenster, in dem das Feld `arrayName` geändert werden kann. Das Feld wird in `varName` abgespeichert.

```
zuUpdArray .problem.startVector startVals "Initial values" startVals
```

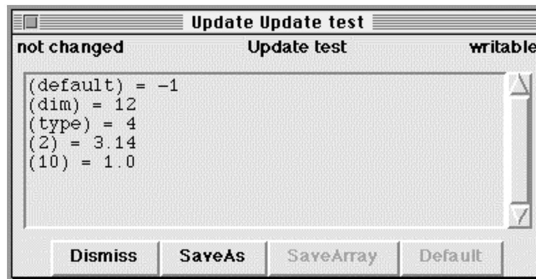

Abbildung 9: Array–Edit Fenster

5 ZGUI: Applikation starten

5.1 tcl/tk–Schnittstelle

Folgende tcl/tk–Funktionen stehen zur Verfügung:

`zgAppStart` startet eine Applikation. Die Standardausgabe der Applikation wird automatisch kontrolliert, das heißt nach dem Empfang einer Zeile beziehungsweise bei einem Prompt können Benutzerprozeduren aufgerufen werden. Die Applikation kann über den Wert einer Variablen angesprochen werden.

```
zgAppStart ad /home/neville/roitzsch/flatten/ss6 cmd=gui.cmd
```

`zgSetProcPrompt` definiert die Tcl–Prozedur, die evaluiert wird, wenn die Applikation `ad` den Prompt (2. Parameter) schreibt.

```
zgSetProcPrompt ad " <CR>" OnAppPrompt
```

Eine solche Prozedur ist etwa

```
proc OnAppPrompt {prompt} {
  global oneStepMode ad
  if {$oneStepMode} { zgAppSendIn $ad "\n" }
```

`zgSetProcLine` definiert den Namen der Prozedur, die nach dem Empfang einer Ausgabezeile der Applikation `ad` aufgerufen wird. Der erste Parameter beim Aufruf ist diese Zeile.

```
zgSetProcLine ad ShowAppOutput
```

`zgSetProcFin` definiert den Namen der Prozedur, die bei Beenden der Applikation `ad` aufgerufen wird. Man beachte, daß die Variable `ad` nach Ausführung der Prozedur umdefiniert wird.

```
zgSetProcLine ad ResetInterface
```

`zgAppSendIn` schreibt eine Eingabe für die Applikation `ad`.

```
zgAppSendIn ad "c\n"
```

`zgAppInform` liefert eine Liste (Tabelle 2) der Statusinformation über die Applikation `ad`.

```
lindex [zgAppInform ad] 2
```

Index	Erläuterung
0	Prozess ID (PID)
1	Rechenzeit (TIME)
2	Speicherbedarf (SZ)
3	Status (STAT)

Tabelle 2: Statusinformation

`zgAppKill` bricht die Applikation `ad` sofort ab.

```
zgAppKill ad
```

`zgInfProcs` liefert die Liste der zu `ad` angemeldeten Prozeduren ab.

```
zgInfProcs ad
{procPrompt " <CR>" OnAppPrompt} {procLine OnLine} .....
```

`zgStopReceive` unterbricht den Empfang der Ausgabe des Anwendungsprogramms.

```
zgStopReceive ad
```

`zgResumeReceive` setzt den Empfang der Ausgabe des Anwendungsprogramms fort.

```
zgResumeReceive ad
```

5.2 Ein Widget zur Kontrolle des Laufes eines Anwendungsprogramms

Für den besonders einfachen Fall, daß ein Anwendungsprogramm lediglich an einigen Haltepunkten einen Prompt in der Standardausgabe absetzt und dann in Abhängigkeit von Standardantworten entweder einen weiteren Schritt rechnet oder den Programmablauf beendet, gibt es ein vorgefertigtes Widget, mit dem man den Programmlauf steuern kann. Zusätzlich kann ein Startprompt und ein Endeprompt behandelt werden. Die Standardausgabe des Anwendungsprogrammes wird in eine Log–Fenster geleitet.

`zuRunWidget runArea runPars` baut innerhalb des Frames `runArea` eine Reihe von Knöpfen zum Starten und Ausführen eines Schrittes, zur fortlaufenden Ausführung, zum Anhalten und definierten Beenden eines Programmes. Zusätzlich werden in einer Statuszeile einige wichtige Daten (Rechenzeit, Speicherbedarf, usw.) des laufenden Programmes angezeigt, siehe Bild 10. Für die entsprechenden Teile werden Hilfe–Texte definiert.

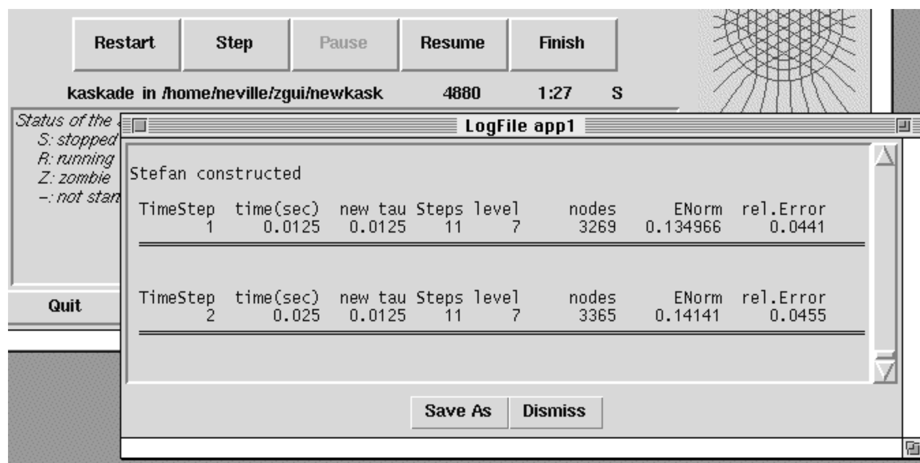

Abbildung 10: Steuerung des Anwendungsprogrammablaufes

Die wichtigsten Steuerinformationen werden über den `runPars` Parameter übergeben. Die für dieses Feld vorgesehenen Parameter sind in Tabelle 3 zusammengestellt.

`zuCloseRunWidget runArea` löscht den Frame `runArea`. Die zugehörigen internen Daten werden gelöscht.

Indexname	Erläuterung
Application	zu startendes Programm
RunDirectory	Directory in dem das Programm gestartet wird
StartPrompt	Prompt, mit dem das Programm beim Programmstart anhält (optionaler Wert)
StandardPrompt	Prompt, nach jedem Programmschritt
EndPrompt	Prompt, mit dem das Programm den letzen Halt signalisiert
StepPromptReply	Standardantwort, die die Ausführung eines Programmschrittes veranlasst
FinishPromptReply	Antwort, die den Abbruch des Programmes veranlasst
StartProc	Benutzerprozedur, die benötigte Parameterdateien erstellt und als Wert Parameter für den Start des Programmes abliefert
FinishProc	Benutzerprozedur, die nach Beendigung der Programmes aufgerufen wird

Tabelle 3: Indizes für das `runPars` Feld

6 `ZGUI`: Datenvisualiserung

Werden von dem Anwendungsprogramm Daten formatiert übermittelt, kann sich der Benutzer die definierte Datenhierarchie (siehe Abschnitt 8.2) ansehen und die graphische Darstellung, etwa als XY–Plot anfordern.

7 Applikation: Parameter–Verwaltung

In diesem Abschnitt soll die Verwaltung von Parameterlisten beschrieben werden. Parameterlisten bestehen aus Paaren von Schlüsselworten und dazugehörigen Werten. Parameterlisten können

- eingerichtet und gelöscht,
- erweitert, abgefragt, eingelesen, gedruckt und geschrieben

werden. Die entsprechenden Basisfunktionen werden für die gängigen Programmiersprachen (C, Fortran, C++, tcl) zur Verfügung gestellt. Über das externe Dateiformat erfolgt der Austauch zwischen Programmen (zum Beispiel einem Anwendungsprogram und einem graphischen Benutzerinterface).

Die Zugriffsfunktionen auf Parameterlisten werden angeboten, um das externe Dateiformat festzuschreiben.

7.1 Interne Datenstrukturen

Die Werte werden einfach als Strings abgespeichert, so daß der Benutzer bei der Abfrage über die Interpretation entscheidet. Als Schlüsselworte sind (intern) beliebige Strings zugelassen. Für das externe Dateiformat sind jedoch nur Namen im üblichen Sinn zulässig.

Die internen Datenstrukturen sind hier nicht interessant. Rudi Beck verwendet Stacks mit Stringvergleich, im tcl–Rahmen bieten sich etwa Hash–Listen an.

7.2 Dateiformat

In externen Parameterbeschreibungen sind Kommentare zulässig sein. Sie beginnen mit einem '#'.

Zur Trennung von Parametername und –wert dient das Gleichheitzeichen '='. Das Ende eines Parameterwertes wird durch das Newline–Zeichen oder einem Semikolon signalisiert. Blanks am Anfang und Ende werden entfernt. Das kann umgangen werden, indem den Parameterwert in Stringquotes einschließt. In diesem Fall kann ein String auch über mehrere Zeilen gehen.

Für Felder und Matrizen wird eine spezielle Syntax für die Interpretation von Parameterwerten verwendet. Dabei können Dimension und Voreinstellungswerte über die speziellen Indexnamen `dim` und `default` definiert werden.

7.3 C Schnittstelle

Die C-Schnittstelle enthält Funktionen zum Einrichten, Informieren, Ergänzen, Ändern und Löschen von Parameterlisten so wie die Definition einer Datenstruktur (`ZP_parList`), die für den Benutzer lediglich zur Identifikation der Liste dient. Alle Funktionen und Variablen sind durch ein vorangestelltes `ZP_` gekennzeichnet.

`ZP_Create` definiert eine neue Parameterliste.

```
extern ZP_parList *ZP_Create(char *name);
....
    ZP_parList *pl = ZP_Create("My parameter list");
    if (pl==ZP_noList) abort();
```

`ZP_Find` sucht eine Parameterliste über einen Namen.

```
# -------------- class Problem --------------

Material = DefaultMaterial
DirichletBCs = ConstDirichletBCs

# --------------   graphics   ---------------

graphics = on              # graphics on
plotSize = 0.5

# -----------  Felder, Matrizen    ----------

steps(dim) = 10
steps(default) = -1.0
steps(type) = 3
steps(1) = 0.0 1.0 -1.0
steps(6) = 3.14

diffmat(dim) = 2,3
diffmat(default) = 0.0
diffmat(type) = 3
diffmat(1,1) = 1.0, 0.0
diffmat(2,1) = 0.0, 1.0

# ------------   lange Strings   ------------

diffpart = "
c
c  root finding sine
c-------------------
c
      dval(1,1)=1.d0+0.75d0*sin(t)
      dval(2,2)=1.d0
      dval(2,1)=0.d0
      dval(1,2)=0.d0
"
```

Abbildung 11: Beispiel einer Parameterdatei

```
extern ZP_parList ZP_Find(char *name);
....
    pl = ZP_Find("defaults");
```

`ZP_Delete` löscht eine existierende Parameterliste.

```
extern void ZP_Delete(ZP_parList *pl);
....
```

```
    ZP_Delete(pl);
```

`ZP_Read` liest eine Parameterdatei in eine Parameterliste ein.

```
extern ZP_parList ZP_Read(ZP_parList *pl,char *fileName);
....
    ZP_parList pl = ZP_Read(0,"/home/xyz/i/.defaultset");
    if (pl==0) abort();
```

`ZP_Write` schreibt eine Parameterliste in eine Parameterdatei.

```
extern int ZP_Write(ZP_parList *pl,char *fileName,
                        char *ioMode);
....
    if (!ZP_Write(pl, "localset", "a");
```

`ZP_SetVal` definiert einen neuen Parameter in einer Parameterliste. Name und Wert sind C–Strings. Falls der Name bereits definiert war, wird ein Pointer auf den alten Wert zurückgemeldet.

```
extern char *ZP_SetVal(ZP_parList *pl,char *parName,
                       char *parValue);
...
    char *oldValue = ZP_SetVal(pl, "plotSolution", "1");
    if (oldValue!=0) { printf("plotSolution reset\n"); }
```

`ZP_IsSet` überprüft, ob ein Parametername in einer Parameterliste belegt ist.

```
extern int ZP_IsSet(ZP_parList *pl,char *parName);
....
    if (ZP_IsSet(pl, "plotSolution"))
```

`ZP_GetVal` liefert den Stringwert zu einem Parameter aus. Im Fehlerfall meldet die Routine `false` zurück.

```
extern int ZP_GetVal(ZP_parList *pl,char *parName,
                         char **sVal);
....
    char *val;
    if (!ZP_GetVal(pl, "plotSolution", &val)) abort();
```

`ZP_GetBool` interpretiert einen Parameterwert als logische Größe. Dabei werden `"0"`, `"false"` und `"off"` als `false` ausgewertet, alle anderen Werte als `true`. Existiert der Parameter nicht, wird `false` als Funktionswert zurückgeliefert.

```
extern int ZP_GetBool(ZP_parList *pl,char *parName,
                      int *bVal);
....
    int val;
    if (ZP_GetBool(pl, "graphics", &val)) abort();
```

`ZP_GetChar` gibt das erste Zeichen des Parameterwertes aus. Falls die Variable nicht definiert ist, wird `cVal` auf `'\0'` gesetzt.

```
extern int ZP_GetChar(ZP_parList *pl,char *parName,
                          char *cVal);
....
    char val;
    if (ZP_GetChar(pl, "graphics", &val)) abort();
```

`ZP_GetShort` wie `ZP_GetLong`

`ZP_GetLong` wandelt den zu einem Parameter gehörenden String in eine Integerzahl. Im Fehlerfall meldet die Routine `false` zurück. Falls die Variable nicht definiert ist, wird `iVal` auf `-1` gesetzt.

```
extern int ZP_GetLong(ZP_parList *pl,char *parName,
                         long *iVal);
....
    long val;
    if (!ZP_GetLong(pl, "plotSolution", &val)) abort();
```

`ZP_GetDouble` wandelt den zu einem Parameter gehörenden String in eine doppeltgenaue reele Zahl. Im Fehlerfall meldet die Routine `false` zurück. Falls die Variable nicht definiert ist, wird `dVal` auf `-1.0` gesetzt.

```
extern int ZP_GetDouble(ZP_parList *pl,char *parName,
                           double *dVal);
....
    double val;
    if (!ZP_GetDouble(pl, "plotSolution", &val)) abort();
```

`ZP_CheckVal` vergleicht den Wert eines Parameters mit einem String. `true` oder `false` wird über die Variable `bVal` zurückgeliefert.

```
extern int ZP_CheckVal(ZP_parList *pl,char *parName,
                           char *keyWord,int *bVal);
....
    int bVal;
    if (!ZP_CheckVal(pl,"DirichletBCs",
                     "ConstDirichletBCs",&bVal)) abort();
    if (bVal) dirichlet = constDirichlet;
```

`ZP_GetArrayDescr` holt den Typ, die Dimension und den Voreinstellungswert eines Feldes. Der Type wird durch Integerkonstanten beschrieben, die in Tabelle 4 beschreiben sind.

Index	Name	Erläuterung
-1	`TYPEUNKNOWN`	unbekannt
1	`TYPESHORT`	short
2	`TYPELONG`	long
3	`TYPEFLOAT`	float
4	`TYPEDOUBLE`	double

Tabelle 4: Arraytypen

```
extern int ZP_GetArrayDescr(ZP_parList pl,char *parName,
                            int *type, char **default, int *low,
                            int *up);
....
   int type, low, up;
   char *defValString;

   ZP_GetArrayDescr(pl,"steps",&type,&defValString",&low,&up);
```

`ZP_GetShortArray` wie `ZP_GetDoubleArray`

`ZP_GetLongArray` wie `ZP_GetDoubleArray`

`ZP_GetFloatArray` wie `ZP_GetDoubleArray`

`ZP_GetDoubleArray` extrahiert ein Feld von doppelt langen Gleitpunktzahlen aus einer Parameterliste. Nicht besetzte Werte erhalten die Voreinstellung. Das Feld `dVals` zeigt auf das nullte Element. Wird kein Hinweis auf den Parameternamen gefunden, liefert die Routine `false` zurück.

```
extern int ZP_GetDoubleArray(ZP_parList pl,char *parName,
                             double *dVals, int low, int up);
....
   double dVals[13];
   if (!ZP_GetDoubleArray(pl,"steps",dVals,0,12)) abort();
```

Das Feld `dVals` hat dann die Besetzung `-1.0, 0.1, 0.2, 0.4, 0.8, 1.6, -1.0, -1.0, -1.0, 0.0, -1.0, -1.0` (vergleiche Abbildung 11).

`ZP_EraseArray` löscht alle Einträge der Parameterliste, die mit dem Arraynamen verbunden sind.

```
extern int ZP_EraseArray(ZP_parList pl,char *parName);
....
   if (!ZP_EraseArray(pl,"steps")) abort();
```

`ZP_GetMatrixDescr` holt den Typ, die Dimension und den Voreinstellungswert einer Matrix. Der Type wird durch Integerkonstanten beschrieben, die in Tabelle 4 beschreiben sind.

```
extern int ZP_GetMatrixDescr(ZP_parList pl,char *parName,
                            int *type, char *default,
                            int *lowCol, int *upCol,
                            int *lowRow, int *upRow);
....
   int type, lowCol, upCol, lowRow, upRow;
   char defValString[100];

   ZP_GetMatrixDescr(pl,"steps",&type,defValString",
                     &lowCol,&upCol, &lowRow,&upRow);
```

`ZP_GetShortMatrix` wie `ZP_GetDoubleMatrix`

`ZP_GetLongMatrix` wie `ZP_GetDoubleMatrix`

`ZP_GetFloatMatrix` wie `ZP_GetDoubleMatrix`

`ZP_GetDoubleMatrix` extrahiert ein Matrix von doppelt langen Gleitpunktzahlen aus einer Parameterliste. Dabei wird von einem spaltenweisen Layout, wie in Fortran, `layout=1` oder einem zeilenweisen, wie in C, `layout=0` ausgegangen. Die Dimension der Spalten bzw. Zeilen wird mit dem `firstDim`–Parameter übergeben. Die Feldadresse `dVals` zeigt auf das Element `0,0`. Wird kein Hinweis auf den Parameternamen gefunden, liefert die Routine `false` zurück.

```
extern int ZP_GetDoubleMatrix(ZP_parList pl,char *parName,
                              double *dVals,
                              int lowCol, int upCol,
                              int lowRow, upRow,
                              int layout, int firstDim);
....
   double dMat[12][5];
   if (!ZP_GetDoubleMatrix(pl,"steps",dMat,0,11,0,4,0,4)) abort();
```

Des weiteren bieten wir Routinen an, mit denen man sich die definierten Parameternamen verschaffen kann.

`ZP_OpenScan` definiert einen Laufindex durch eine ParameterListe.

`ZP_Next` liefert das nächste Element einer Parameterliste bezüglich einem Laufindex aus.

`ZP_CloseScan` gibt den Laufindex wieder frei..

```
extern ZP_parIndex *ZP_OpenScan(ZP_parList *pl);
extern int ZP_Next(ZP_parIndex *pi,char **parName,
                        char **parValue);
extern void ZP_CloseScan(ZP_parIndex **pi);
....
    ZP_parIndex *pi = ZP_OpenScan(pl);
    char *parName, *parValue;
    while (1)
    {
      if (!ZP_Next(pi,&parName,&parValue)) break;
    }
    ZP_CloseScan(&pi);
```

7.4 C++ Schnittstelle

7.5 Fortran–Schnittstelle

Fortran–Name	Parameter	C Name
ZPCR	PLNAME,IPL,IRTN	ZP_Create
ZPFD	PLNAME,IPL,IRTN	ZP_Find
ZPDL	IPL,IRTN	ZP_Delete
ZPRD	IPL,FILNAM,IRTN	ZP_Read
ZPWR	IPL,FILNAM,MODE,IRTN	ZP_Write
ZPSV	IPL,PAR,VAL,IRTN	ZP_SetVal
ZPGV	IPL,PAR,VAL,IRTN	ZP_GetVal
ZPDF	IPL,PAR,IRTN	ZP_IsSet
ZPCHK	IPL,PAR,VAL,BVAL,IRTN	ZP_CheckVal
ZPGLV	IPL,PAR,BVAL,IRTN	ZP_GetBool
ZPGCV	IPL,PAR,CVAL,IRTN	ZP_GetChar
ZPGIV	IPL,PAR,IVAL,IRTN	ZP_GetInt
ZPGRV	IPL,PAR,RVAL,IRTN	ZP_GetFloat
ZPGDV	IPL,PAR,DVAL,IRTN	ZP_GetDouble
ZPGIA	IPL,PAR,IARR,LNG,IRTN	ZP_GetLongArray
ZPGFA	IPL,PAR,FARR,LNG,IRTN	ZP_GetFloatArray
ZPGDA	IPL,PAR,DARR,LNG,IRTN	ZP_GetDoubleArray
ZPGIM	IPL,PAR,IMAT,IROW,ICOL,IDIM,IRTN	ZP_GetLongMatrix
ZPGFM	IPL,PAR,FMAT,IROW,ICOL,IDIM,IRTN	ZP_GetFloatMatrix
ZPGDM	IPL,PAR,DMAT,IROW,ICOL,IDIM,IRTN	ZP_GetDoubleMatrix
ZPOSC	IPL,IND,IRTN	ZP_OpenScan
ZPNXT	IND,PAR,VAL,IRTN	ZP_Next
ZPCSC	IND,IRTN	ZP_CloseScan

PLNAME, FILNAM, MODE, PAR und VAL sind Zeichenfelder,
IPL, IND, IRTN Integervariable,
LNG Länge des Ausgabefeldes,
IROW, ICOL Dimension der Ausgabematrix, IDIM, Zeilenlänge der Ausgabematrix,
BVAL ist Ausgabeparameter vom Type CHARACTER
IVAL, IARR, IMAT sind Ausgabeparameter vom Type INTEGER
FVAL, FARR, FMAT sind Ausgabeparameter vom Type REAL
DVAL, DARR, DMAT sind Ausgabeparameter vom Type DOUBLE
Rückmeldungen der Routine erfolgen über den IRTN–Parameter, die häufigsten Fehlercodes sind in Tabelle 6 zusammengestellt.

Tabelle 5: Fortran–Schnittstelle

Die Fortran–Schnittstelle (Tabelle 5) beschränkt sich auf die Nutzung des Fortran 77 Sprachumfangs. Die Namensgebung ist entsprechend kryptisch (6 Zeichen je Name, nur Großbuchstaben).

Bei der Auswertung von Zeichenfelder werden abschließende Blanks entfernt, bei der Rückmeldung in Zeichenfelder werden diese mit Blanks gefüllt.

Fehlercode	Erläuterung
0	alles in Ordnung
1	`ZP_Next` hat das Listenende erreicht
-1	Fehler in `ZP_`–Routine
-2	Variable nicht definiert
-3	Variable hat kein Wert
-4	Variablewert syntaktisch falsch
-20	IPL keine Parameterliste
-21	IND keine Laufvariable
-50	internal buffer too short (`CopyTrim`)
-51	internal buffer too short (`CopyFill`)

Tabelle 6: Fortran–Fehlercodes

Reicht der Platz nicht aus, erfolgt eine Fehlermeldung.

Zur Demonstration ein einfaches Beispiel: Routinen.

```
      CHARACTER*24 VAL
      DOUBLE PRECISION X, STEPS(12)
      LOGICAL L1, L2

      IPL = 0
      CALL ZPRD(IPL,'ss6.cmd',IRTN)
      WRITE(6,100) IPL, IRTN
      IF (IRTN.NE.0) STOP

      CALL ZPGV(IPL,'problem',VAL,IRTN)
      WRITE(6,101) IRTN, VAL
      CALL ZPGRV(IPL,'bloedsinn',Y,IRTN)
      WRITE(6,102) IRTN, Y
      CALL ZPGDV(IPL,'plotSize',X,IRTN)
      WRITE(6,102) IRTN, X
      CALL ZPGDA(IPL,'steps',STEPS,12,IRTN)
      WRITE(6,103) IRTN, STEPS
      CALL ZPGLV(IPL,'plotTimeStep',L1,IRTN)
      WRITE(6,104) IRTN, L1
      CALL ZPGLV(IPL,'writeTimeStep',L2,IRTN)
      WRITE(6,104) IRTN, L2

      CALL ZPWR(IPL,'ftntest.cmd','w',IRTN)
      CALL ZPDL(IPL,IRTN)
 100  FORMAT('IPL=', I8, ', IRTN=', I2)
 101  FORMAT('ZPGV : IRTN=', I2, ', VAL="', A24, '"')
 102  FORMAT('ZPGDV: IRTN=', I2, ', X=', F5.2)
 103  FORMAT('ZPGDA: IRTN=', I2, ', X=', 6F5.1/ 18X ,6F5.1)
 104  FORMAT('ZPGLV: IRTN=', I2, ', L=', L1)
      STOP
      END
```

Das Programm erzeugt die folgende Ausgabe:

```
(35)%ftnpartest
Read 107 lines, 71 parameters from ss6.cmd
IPL=  365080, IRTN= 0
ZPGV : IRTN= 0, VAL="staticHeatConduction  "
ZPGDV: IRTN=-4, X= 0.00
ZPGDV: IRTN= 0, X= 0.40
ZPGDA: IRTN= 0, X= -1.0  0.1  0.2  0.4  0.8  1.6
                   -1.0 -1.0 -1.0  0.0 -1.0 -1.0
ZPGLV: IRTN= 0, L=T
ZPGLV: IRTN= 0, L=F
Write 71 lines written to 'ftntest.cmd'
(36)
```

7.6 Tcl Schnittstelle

`zpCreate` definiert eine neue Parameterliste. Der Wert der Funktion ist "identifizierender" String. Es wird überprüft, ob das erste Argument eine Variable und diese noch nicht andersweitig definiert ist. Bei erfolgreichem Ablauf ist `pl` ein Variable.

```
zpCreate pl
```

`zpDelete` löscht eine Parameterliste.

```
zpDelete pl
```

`zpRead` lesen einer Parameterdatei und eintragen in eine existierende beziehungsweise neu kreierten Parameterliste.

```
zpRead pl ss6.cmd
```

`zpWrite` schreibt eine Parameterliste in eine Parameterdatei.

```
zpWrite pl user.cmd
```

`zpSetVal` setzt/ändert einen Parameter. Falls ein alter Parameterwert existierte, wird dieser zurückgemeldet.

```
set oldVal [zpSetVal pl solution 1]
```

`zpGetVal` liefert einen Parameterwert aus.

```
zpGetVal pl solution
```

`zpGetArray` liefert ein Feld `x` als tcl–array aus. Die Dimension, der Vorbesetzungswert und der Typ werden unter den Indices `dim`, `default` und `type` (siehe Tabelle 4) abgelegt.

```
zpGetArray pl x
foreach elem [array names x] {
  puts [format "%s(%s) = %s" x $i x($i)]
}
```

`zpSetArray` setzt ein Feld `x`. Es werden nur vom Vorbesetzungswert verschiedene Werte abgelegt und obsolete Einträge in der Parameterliste vorher gelöscht.

```
zpSetArray pl x
```

`zpIsSet` überprüft, ob ein Parameter definiert ist. Ergebnis ist `0` oder `1`.

```
zpIsSet pl solution
```

`zpForeach` bildet in Analogie zu `foreach` eine Schleife über alle Elemente einer Parameterliste.

```
zpForeach name val pl {
puts [format "%s %s" $name $val]
}
```

8 Applikation: Ausgabeformatierung

8.1 Haltepunkte

Zum richtigen Setzen der Prompts steht für Fortran und C (C++) jeweils ein Unterprogramm zur Verfügung, mit dem ein Prompt gesetzt wird. Dahinter verbirgt sich eine Verpackung des Prompt–Textes, um zufällige Übereinstimmungen mit der normalen Ausgabe zu vermeiden, und das automatische Ausstülpen der Standardausgabe (flush). Gegebenenfalls wird auf eine Antwort gewartet.

C/C++ Das Unterprogramm `ZP_Prompt` schreibt einen Prompt. Die (einzeilige) Antwort wird in `buffer` hinterlegt, falls `bufferLng>0` ist. Die Länge der Antwort wird zurückgemeldet.

```
extern int ZP_Prompt(char *prompt, char *promptReply);
....
    lng = ZP_Prompt(" <CR>", buffer, bufferLng);
```

Fortran Das Unterprogramm ZGPMPT(PROMPT,LINE,BUFLNG) schreibt einen Prompt PROMPT. Die (einzeilige) Antwort wird im LINE hinterlegt, falls des Länge BUFLNG>0 ist. Die Länge der Antwort wird in LNG zurückgemeldet.

```
CHARACTER*80 LINE

BUFLNG = 80
CALL ZGPMPT(' <CR>',LINE,BUFLNG,LNG)
```

8.2 Ausgabe spezieller Daten

In diesem Abschnitt werden Routinen beschrieben, mit denen ein Anwendungsprogramm Daten über die Standardausgabe so schreibt, das möglichst automatisch eine entsprechende Verarbeitung durch andere Programme, insbesondere natürlich ZGUI erfolgen kann. Da in unserem Kontext adaptive Verfahren im Zentrum stehen, wird zunächst ein einfaches hierarchisches Datenformat vorgesehen. Dazu kann eine Datennamen einer bestimmten Hierarchiestufe zugeordnet werden. Die Hierarchiestufen werden wieder durch Datennamen definiert.

Zum Beispiel sei folgende Datennamen–Hierarchie zu definieren.

```
Depth
    IterationStep
    IterationError
#Iterations
GlobalError
#nodes
```

Im Anwendungsprogramm kann diese Hierarchie etwa durch folgende Unterprogrammaufrufe definiert werden:

```
depthID = ZD_DataDef("Depth",0);
pdeIterID = ZD_DataDef("#Iterations",0);
linIterID = ZD_DataDef("IterationStep",depthID);
linIteErrorID = ZD_DataDef("IterationError",depthID);
globErrorID = ZD_DataDef("GlobalError",0);
nodesID = ZD_DataDef("#nodes",0);
```

Die Daten können dann mit entsprechenden Unterprogrammaufrufen geschrieben werden:

```
ZD_IntWrite(nodesID,actTriang->noOfPoints);
ZD_RealWrite(globErrorID,globError);
```

Diese Routine schreiben die entsprechende Information nach `stdout` oder in eine Datei.

```
$ZDDefine Depth ""
$ZDDefine #Iterations ""
$ZDDefine IterationStep #Depth
$ZDDefine IterationError #Depth
$ZDDefine GlobalError ""
$ZDDefine #nodes ""
  ....
$ZDData #nodes 1234
$ZDData GlobalError 0.00987
```

Diese Daten können dann von `ZGUI` extrahiert und verarbeitet werden.

8.2.1 C Schnittstelle

Die im folgendem verwendete `ZD_DataDesc`–Datenstruktur bleibt dem Benutzer verborgen.

`ZD_DataDef` definiert einen neuen Datennamen `name`, der von `dep` abhängt.

```
extern ZD_DataDesc *ZD_DataDef(char *name,ZD_DataDesc *dep);
....
    ZD_DataDesc *pdeIterID = ZD_DataDef("Depth",0);
    ZD_DataDesc *linIterID = ZD_DataDef("IterationStep",pdeIterID);
```

`ZD_IntWrite`, `ZD_RealWrite` schreibt einen Integer– bzw. Real–Datum.

```
extern void ZD_IntWrite(ZD_DataDesc *d,int data);
extern void ZD_RealWrite(ZD_DataDesc *d,double data);
....
   ZD_IntWrite(nodesID,actTriang->noOfPoints);
   ZD_RealWrite(globErrorID,globError);
```

`ZD_State` setzt den iternen Status dieser Routinen. Mögliche Werte sind `ZD_SILENT` oder `ZD_ACTIVE`. Der alte Status wird zurückgemeldet. Voreinstellung ist `ZD_SILENT`.

```
extern int ZD_State(int state);
....
   ZD_State(ZD_SILENT);
```