FLUENT 5 Programmierung

Mirko Javurek

Linz, im August 1999

Institut für Mechanik
Abteilung für Strömungslehre und Wärmeübertragung

der Technisch-Naturwissenschaftlichen Fakultät
der Johannes Kepler Universität Linz

1. User-Defined Functions

Dieser Bericht stellt nur eine Ergänzung zum Kapitel 24, "User-Defined Functions" des FLUENT 5-Handbuchs (Ausgabe Juli 1998) dar. In diesem Bericht sollen jene Erfahrungen und Erkenntnisse, die im Zuge der Programmierung gemacht wurden, aber zum Teil in keiner FLUENT-Dokumentation enthalten sind, festgehalten und für andere zugänglich gemacht werden.

2. Geometrieelemente

Ein unstrukturiertes Netz besteht aus Knoten (nodes), Zellflächen (faces) und Zellvolumen (cells), die folgende Gestalt annehmen können (in Klammer die Anzahl der Knoten):

	2D	3D
cell	Dreieck (3), Viereck (4)	Tetraeder (4), Hexaeder (6 oder 8)
face	Linie (2)	Dreieck (3), Viereck (4)

Jedes Zellvolumen ist von einer entsprechenden Anzahl von Zellflächen umgeben, gemeinsame Zellflächen bei aneinandergrenzenden Zellvolumen sind nur einfach vorhanden (Bild 1).

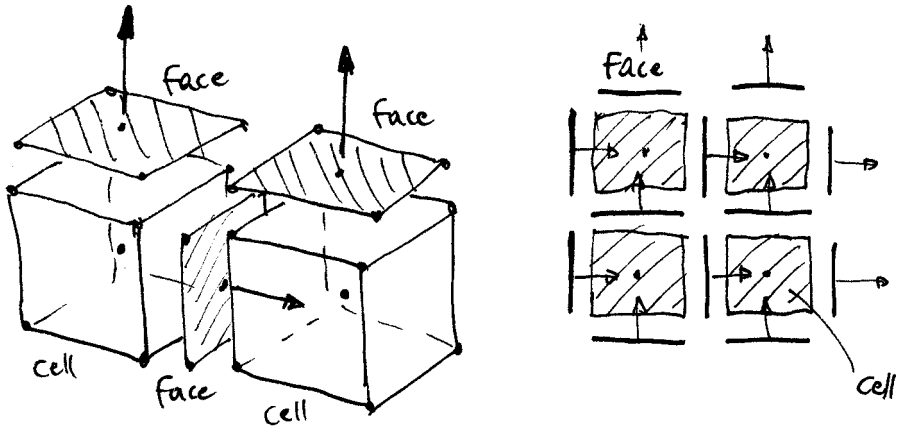

Bild 1: "cells" und "faces" in 3D und 2D

3. Gliederung

Die Zellflächen und Zellvolumen sind jeweils zu Zonen (zones bzw. threads) zusammengefaßt (s.a. Bild 2 und Bild 3):

type	**zone-name**
cells	fluid
faces	interior
	wall, pressure inlet, ...

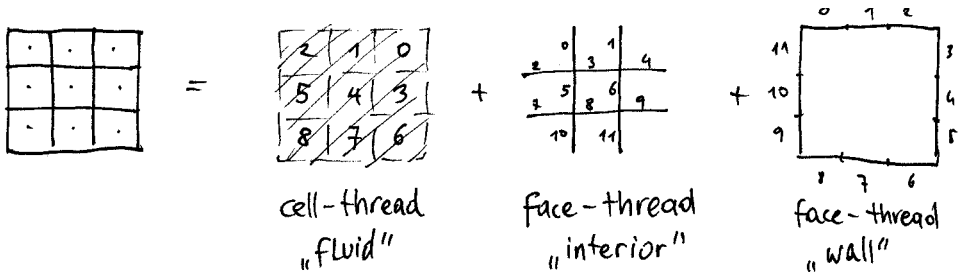

Bild 2: cell- und face-threads am Beispiel einer einfachen 2d-Geometrie

Gewöhnlich gibt es nur ein "cell-thread", nämlich die "fluid"-Zone, ein "interior face-thread", und entsprechend der Randzonen für die Randbedingungen diverse andere "face-threads". Jeder Zone ist eine "ID" zugeordet. Die Zonen einer Geometrie und ihre ID scheinen in FLUENT im "define/boundary conditions"-Requester auf.

Innerhalb dieser Zonen sind die cells und faces in einer *unbestimmten Reihenfolge* durchnummeriert (sozusagen "aufgefädelt", deshalb vermutlich auch die Bezeichnung "thread" = Faden); sie werden über einen einzelnen Index angesprochen (im Gegensatz dazu sind im *strukturierten* Gitter die Zellen einer zwei- oder dreidimensionalen Matrix entsprechend angeordnet, und werden über zwei bzw. drei Indizes angesprochen). Die Nachbarzellen einer Zelle (bzw. die Nachbar-faces eines faces) können also nicht einfach über die Nachbarindizes angesprochen werden!

Die Zonen wiederum bilden zusammen eine Domäne (domain). Eine Geometrie besteht gewöhnlich nur aus einer Domäne.

4. Datenstrukturen

Die Daten der Zellvolumen und -flächen (z.B. Koordinaten, Druck, Geschwindigkeitskomponeneten...) werden in der jeweiligen thread-structure in einem array abgespeichert. Um den Wert einer bestimmten Zellfäche (bzw. Zellvolumens) anzusprechen, ist also die Angabe des threads, in dem sich die Zellfäche (bzw. das Zellvolumen) befindet, und die Nummer (index) der Zellfäche (bzw. des Zellvolumens) anzugeben. Folgende Datentypen (Programmiersprache C) treten dabei auf:

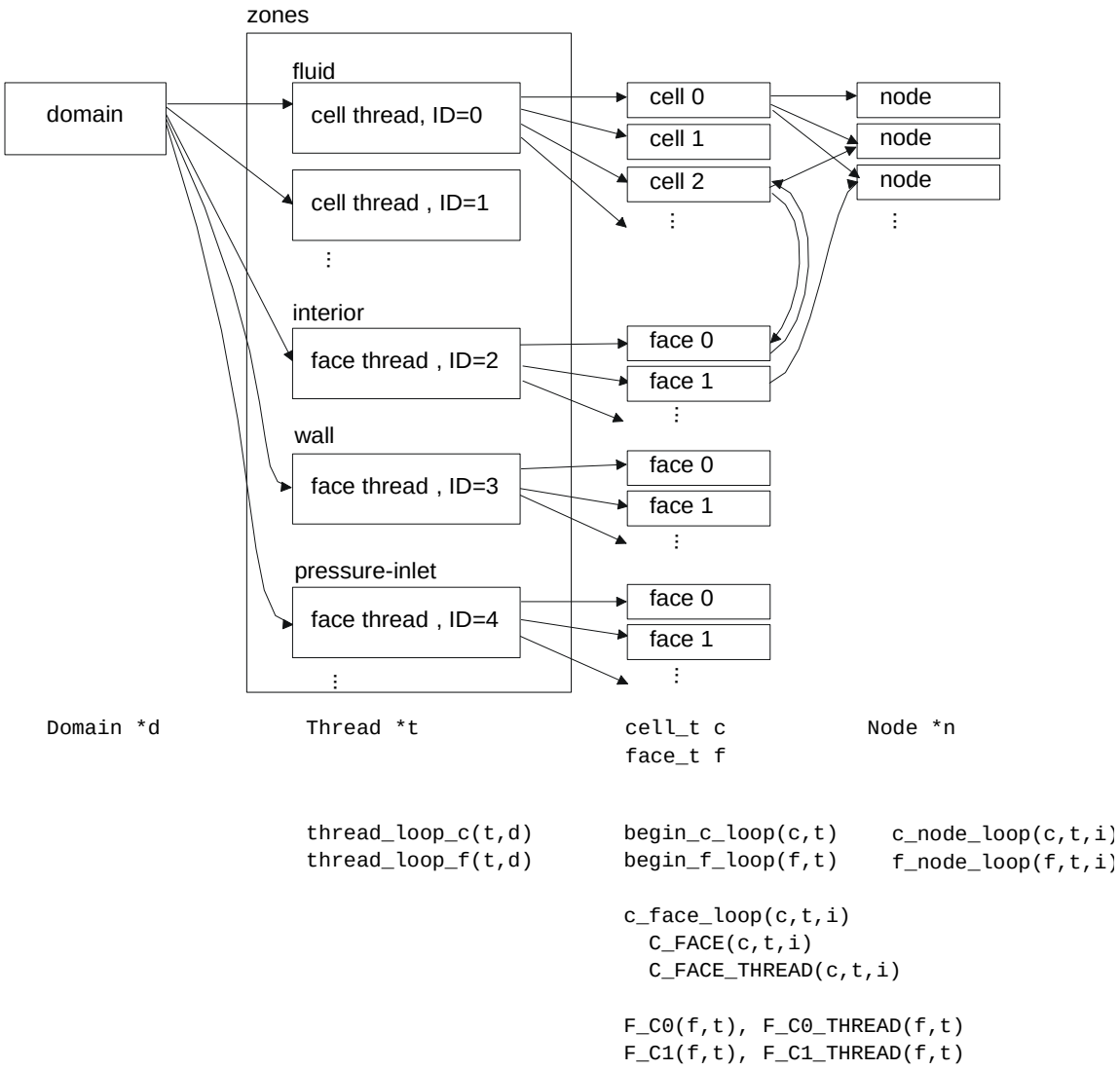

Bild 3: Geometriestruktur in FLUENT 5

`Domain *d` (pointer to) domain-structure
`Thread *t` (pointer to) thread-structure
`cell_t c` cell index
`face_t f` face index
`Node *n` (pointer to) node-structure

Im folgenden sind FLUENT-Makros angeführt, mit denen auf die gespeicherten Werte zugegriffen werden kann. Die meisten dieser Makros sprechen direkt die entsprechende Variable an, sodaß ein Zugriff nicht nur lesend, sondern auch schreibend erfolgen kann; ein paar wenige Makros sind allerdings Funktionen ohne Rückgabewert, die nur als Befehl aufgerufen werden können (mit * gekennzeichnet).

4.1. Thread-Variablen

Threads können durch ihre ID und ihren Typ identifiziert werden:

`int THREAD_ID(t)` thread-ID
`int THREAD_TYPE(t)` thread-type

Für den Thread-Typ (fluid, interior, wall, ...) sind in "threads.h" entsprechende Konstanten definiert.

Es folgen die Variablen für cells, faces und nodes:

4.2. Geometrievariablen

	cell	face	node
Koordinatenvektor `x`	`C_CENTROID(x,c,t)`*	`F_CENTROID(x,f,t)`*	`x=NODE_COORD(n)`
Volumen	`C_VOLUME(c,t)`	-	-
Flächenvektor `a`	-	`F_AREA(a,f,t)`*	-

Die Richtung der Flächenvektoren ist bei interior-faces unbestimmt, bei außenliegenden faces nach außen zeigend (?).

Wenn das Gitter defomiert werden soll, so müssen die Knotenkoordinaten verändert werden. Anschließend muß die Funktion `Update_Volume(d)` aufgerufen werden; gleichzeitig werden auch die cell- und face-centroid-Koordinaten, die Volumina und die Flächenvektoren aktualisiert.

4.3. Feldvariablen

Sämtliche Feldvariablen sind auf jeden Fall in den cells definiert, aber nur teilweise an den faces und gar nicht an den nodes:

	cell	face	node
Dichte	`C_R(c,t)`	`F_R(f,t)`	-
Geschwindigkeiten	`C_U(c,t), C_V(c,t), C_W(c,t)`	`F_U(f,t), ...`	-
...	...	...	-

Eine vollständigere Auflistung ist im Fluent-Handbuch bzw. eine zwar ganz vollständige, aber kommentarlose im FLUENT-include-file "mem.h" zu finden.

Gesondert hervorzuheben sind noch die Gradienten: für jede Feldvariable (nicht nur für die Geschwindigkeiten) gibt es einen Gradientenvektor (zusätzliche Endung `_G`, also z.B. `C_P_G(c,t)[0]`, `C_P_G(c,t)[1]`, ... für den Druckgradienten). Damit auf diese Varaiblen zugegriffen werden kann, muß allerdings im Textinterface unter "solve/set/expert" "keep temporary solver memory from being freed?" auf "yes" gesetzt werden.

Die willkürliche Anordnung der Zellen in unstrukturierten Netzen macht es unmöglich, den Wert einer Feldvariablen $\varphi(\vec{x})$ an einem beliebigen Punkt mit dem Ortsvektor $\vec{x}$ direkt zu ermitteln; dazu muß z.B. erst unter allen(!) Zellen diejenige gesucht werden, die den Punkt $\vec{x}$ enthält, bzw. dem Punkt $\vec{x}$ am nächsten ist. In einem strukturierten Netz kann dieses Problem oft durch geeignete Ausrichtung des Gitters umgangen werden.

4.4. Querverweise

Erst durch die folgenden Querverweise (von denen kein einziger im FLUENT-Handbuch angeführt ist!) wird die Implementierung von etwas komplexeren Erweiterungen möglich:

	cell	face
i-ter Knoten	`n=C_NODE(c,t,i)`	`n=F_NODE(c,t,i)`
angrenzende cells	-	`c=F_C0(f,t), F_C1(f,t)`
angrenzende cell-threads	-	`t=F_C0_THREAD(f,t), F_C1_THREAD(f,t)`
i-tes face	`f=C_FACE(c,t,i)`	-
i-tes face-thread	`t=C_FACE_THREAD(c,t,i)`	-

Der Zugriff auf die Knoten ist nur über die cells bzw. faces möglich. Es gibt offenbar keine nach außen hin "sichtbare" Querverweise auf die Knoten einer domain bzw. zwischen den Knoten selbst. Wenn cells und faces einen gemeinsamen Knoten haben, so verweisen sie auch auf den selben Knoten (`C_NODE(c,t,i)==F_NODE(f,t,j)`).

Der Zugriff von Zellen auf ihre Nachbarzellen ist nur indirekt über die faces möglich; es wurde aber eine Routine implementiert, die einen direkten Zugriff ermöglicht (siehe unten).

Vorsicht ist bei Gitterverfeinerung mit "hanging nodes" geboten: hier gibt es an den Grenzen des Verfeinerungsbereiches keine Verweise von den groben Zellen auf die feineren Zellflächen!

Für Verweise auf die threads einer domain siehe Kapitel "Schleifen".

5. Schleifen

	cell	face
threads in domain	`thread_loop_c(t,d)`	`thread_loop_f(t,d)`
cells/faces in threads	`begin_c_loop(c,t)`	`begin_f_loop(f,t)`
nodes of cell/face	`c_node_loop(c,t,i)`	`f_node_loop(f,t,i)`
faces of cell	`c_face_loop(c,t,i)`	-
cells of face	-	$\nexists$

6. Schnittstellen

Folgende Funktionen können in einer user-defined-function implementiert und anschließend in FLUENT 5 aktiviert werden. Während die Init-, Adjust- und Profil-Funktionen nur einmal pro Iteration aufgerufen werden, werden die meisten anderen Funktionen für jede cell bw. für jedes face extra aufgerufen.

6.1. Init- und Adjust-Funktionen

werden unter "define/user-defined/function hooks" gesetzt. Die Init-Funktion wird von FLUENT beim Initialisieren aufgerufen, die Adjust-Funktion am Anfang jeder Interation.

```
DEFINE_INIT(name, domain) void name(Domain *domain)
DEFINE_ADJUST(name, domain) void name(Domain *domain)
```

6.2. Profilfunktionen für Randbedingungen

werden unter "define/boundary conditions" gesetzt.

```
DEFINE_PROFILE(name, t, i) void name(Thread *t, int i)
```

Bei "solve/iterate" kann eingestellt werden, wie oft die Profilfunktion aufgerufen und damit die Randwerte aktualisiert werden sollen. Für genaure Informationen siehe Handbuch.

6.3. Materialkonstanten

Unter "define/materials" kann für die Diffusionskonstante die Funktion

```
DEFINE_DIFFUSIVITY(name, c, t, i) real name(cell_t c, Thread *t, int i)
```

mit `return` $D(\vec{x})$ und für Dichte, Viskosität, thermische Leitfähigkeit und Wärmekapazität die Funktion

```
DEFINE_PROPERTY(name, c, t) real name(cell_t c, Thread *t)
```

angegeben werden.

6.4. Quellterme

Quellterme können für (fast) alle von FLUENT gelösten Gleichung (Massenerhaltung, Impulserhaltungen (2 bzw. 3 Gleichungen), Energie, Turbulenz (k und ϵ), Species, UDS, ...) bei "define/boundary conditions/fluid" aktiviert werden:

```
DEFINE_SOURCE(name, c, t, dS, j)
  real name(cell_t c, Thread *t, real dS[], int j)
```

Obwohl der Quellterm q für jede Zelle (c,t) anzugeben ist, muß der volumsbezogene Quellterm mit `return` $q\,(\vec{x})$ angegeben werden (= Quellterm in differentieller Formulierung der Feldgleichungen). Falls der Quellterm q von der Variable φ, nach der die jeweilige Gleichung gelöst wird, abhängig ist (z.B. $q\,(u_x)$ in der x-Komponente der Impulserhaltungsgleichung), so muß die Ableitung nach dieser Variablen auch gesetzt werden:

$$\texttt{dS[j]} = \frac{\partial q}{\partial \varphi}$$

6.5. Euler-Lagrange-Modell

Für das Euler-Lagrange-Modell (DPM) gibt es eine Reihe von Funktionen, die Erweiterungen ermöglichen (Befehl `#include"dpm.h"` nötig):

- Teilchenkraft:

```
DEFINE_DPM_BODY_FORCE(name, p, i) real name(Tracked_Particle *p, int i)
```

- Widerstandsgesetz (`return` $C_\mathrm{D} \cdot \mathrm{Re}$!):

```
DEFINE_DPM_DRAG(name, Re, p) real name(real Re, Tracked_Particle *p)
```

- Analog zu den UDS gibt es DPM-scalars: Das sind Skalare, die entlang der Partikelbahnen definiert sind, und bei der Darstellung der Partikelbahnen zur Einfärbung verwendet werden können. Der Zugriff auf das j-te Skalar in der Funktion erfolgt mit `P_USER_REAL(p,j)`:

```
DEFINE_DPM_SCALAR_UPDATE(name, c, t, initialize, p)
  void name(cell_t c, Thread *t, int initialize, Tracked_Particle *p)
```

Für den Zugriff auf die Partikelvariablen (z.B. `P_DIAM(p)` für den Partikeldurchmesser) siehe "dpm.h".

—

Es gibt noch mehr Arten von Funktionen (siehe "udf.h"), deren Funktion bzw. Verwendung mangels Dokumentation allerdings unklar ist, weshalb sie hier nicht angeführt wurden.

Es gibt in FLUENT 5 keine Möglichkeit, Variablen einer UDF vom Programm FLUENT aus zu setzen (USRVAR in FLUENT 4). Um Parameter (z.B. Zahlenkonstanten) einer UDF zu ändern, müssen diese im UDF-Quelltext geändert und die UDF-Bibliothek neu geladen werden.

6.6. Textinterface

Für Ausgaben ins Textinterface kann der Befehl `CX_Message("...format...", ...)` verwendet werden; die Argumente sind gleich wie beim `printf(...)`-Befehl.

7. User-Defined Scalars

Benutzerdefinierte Skalargleichung in ϕ:

$$I + \frac{\partial}{\partial x_i}\left(F_i\phi - \Gamma\frac{\partial\phi}{\partial x_i}\right) = q$$

Folgende UDFs ermöglichen die Implementierung der UDS-Gleichung (Nachdem mehrere UDS definiert werden können, steht j für die j-te Skalargleichung):

I	`DEFINE_UDS_UNSTEADY(name,c,t,j,a,b)`	Instationärer Term
F'	`DEFINE_UDS_FLUX(name,f,t,j)`	Fluß durch das face (f,t)
Γ	`DEFINE_DIFFUSIVITY(name,c,t,j)`	Diffusionsterm in der cell (c,t)
q	`DEFINE_SOURCE(name,c,t,dq,n)`	Quellterm und
$\frac{\partial q}{\partial \phi}$	`dq[n]`$=\frac{\partial q}{\partial \phi}$	dessen Ableitung in der cell (c,t)

mit dem Fluß F' durch das Face mit dem Flächenvektor ΔA_i:

$$F' = F_i\,\Delta A_i$$

Als Fluß kann statt einer UDF auch "none"

$$I - \frac{\partial}{\partial x_i}\left(\Gamma\frac{\partial\phi}{\partial x_i}\right) = q$$

oder "mass-flow-rate"

$$I + \frac{\partial}{\partial x_i}\left(\rho u_i\phi - \Gamma\frac{\partial\phi}{\partial x_i}\right) = q$$

gewählt werden.

Der instationäre Term wird in entweder mit "none"

$$\frac{\partial}{\partial x_i}\left(F_i\phi - \Gamma\frac{\partial\phi}{\partial x_i}\right) = q,$$

"default"

$$\frac{\partial}{\partial t}\left(\rho\phi\right) + \frac{\partial}{\partial x_i}\left(F_i\phi - \Gamma\frac{\partial\phi}{\partial x_i}\right) = q$$

oder über eine UDF benutzerdefinert in zeit- und ortsdiskreter (d.h. für jede Zelle) Form über die Terme a und b angegeben:

$$-\frac{a\phi^{(n)} + b}{V_{\text{Zelle}}} + \frac{\partial}{\partial x_i}\left(F_i\phi^{(n)} - \Gamma\frac{\partial\phi^{(n)}}{\partial x_i}\right) = q$$

Der "default"-Term

$$\frac{\partial}{\partial t}\left(\rho\phi\right)$$

in diskreter Formulierung

$$\frac{\rho^{(n)}\phi^{(n)} - \rho^{(n-1)}\phi^{(n-1)}}{\Delta t}$$

entspricht also z.B. den Termen

$$a = -\frac{\rho^{(n)}}{\Delta t} V_{\text{Zelle}}$$
$$b = \frac{\rho^{(n-1)}\phi^{(n-1)}}{\Delta t} V_{\text{Zelle}}$$

Auf die Feldvariablen aus dem vorhergehenden Zeitschritt kann über ein angehängtes `_M1` an das entsprechende Macro zugegriffen werden, also z.B. `C_R_M1(c,t)` für $\rho^{(n-1)}$; für die UDSs muß allerdings erst ein Macro definiert werden:

```
#define C_UDSI_M1(c,t,i)C_STORAGE_R(c,t,SV_UDS_I(i)+SV_UDS_0_M1-SV_UDS_0)
```

Den Zeitschritt erhält man über den Funktionsaufruf
$\Delta t =$`RP_Get_Real("physical-time-step");`

Als Randbedingung kann entweder ein Wert für das Skalar oder ein Fluß Φ vorgegeben werden:

$$\Phi = \int\limits_{A_{\text{Rand}}} \left(F_i\phi - \Gamma\frac{\partial\phi}{\partial x_i}\right) \mathrm{d}A_i$$

für eine Zelle angeschrieben:

$$\left(F_i\phi - \Gamma\frac{\partial\phi}{\partial x_i}\right) \Delta A_i = \Phi\frac{\Delta A}{A}$$

bzw. mit dem Einheitsnormalenvektor $\vec{n}$ ($\Delta\vec{A} = \Delta A \cdot \vec{n}$):

$$\left(F_i\phi - \Gamma\frac{\partial\phi}{\partial x_i}\right) n_i = \frac{\Phi}{A}$$

Andere Randbedingungen können zum Teil realisiert werden, indem die Randbedingung "Fluß gleich Null" gewählt wird, und in den Randzellen entsprechende Quellterme gesetzt werden.

UDSs können aber auch passiv benutzt werden, indem die UDS-Gleichung beim Solver deaktiviert wird; dann kann das UDS in Kombination mittels einer Adjust-Funktion als leistungsfähigere Alternative zu den "custom field functions" verwendet werden, z.B. zur Visualisierung von Skalarfeldern $\varphi\left(\vec{x}, ?\right)$ mit komplexer funktionaler Abhängigkeit oder aber auch zur Markierung für Verfeinerungsbereiche des Gitters.

8. Vektorarithmetik

Folgende Makros (in "flow.h" definiert) führen Vektoroperationen entsprechend der Dimensionen (2D oder 3D) aus:

```
/* NV_S (a, =, 2) */
#define NV_S(a,EQ,s)
#define ND_S(a0,a1,a2,EQ,s)
/* NV_V (a, =, x) */
#define NV_V(a,EQ,x)
#define ND_D(a0,a1,a2,EQ,x0,x1,x2)
#define NV_D(a,EQ,x0,x1,x2)
#define ND_V(a0,a1,a2,EQ,x)
/* NV_VS (a, =, x,/,2) */
#define NV_VS(a,EQ,x,S,s)
#define NV_DS(a,EQ,x0,x1,x2,S,s)
#define ND_VS(a0,a1,a2,EQ,x,S,s)
#define ND_DS(a0,a1,a2,EQ,x0,x1,x2,S,s)
/* NV_VV (a, =, x,+,y) */
#define NV_VV(a,EQ,x,V,y)
#define NV_DD(a,EQ,x0,x1,x2,V,y0,y1,y2)
#define ND_DD(a0,a1,a2,EQ,x0,x1,x2,V,y0,y1,y2)
/* NV_VV_S (a, =, x,+,y, /, 2) */
#define NV_VV_S(a,EQ,x,V,y,S,s)
#define NV_DV_S(a,EQ,x0,x1,x2,V,y,S,s)
#define NV_DD_S(a,EQ,x0,x1,x2,V,y0,y1,y2,S,s)
#define ND_VD_S(a0,a1,a2,EQ,x,V,y0,y1,y2,S,s)
#define ND_DV_S(a0,a1,a2,EQ,x0,x1,x2,V,y,S,s)
#define ND_DD_S(a0,a1,a2,EQ,x0,x1,x2,V,y0,y1,y2,S,s)
/* NV_V_VS (a, =, x, +, (y, *, 0.5)) */
#define NV_V_VS(a,EQ,x,V,y,S,s)

#define ND_SET(x0,x1,x2,y0,y1,y2)ND_D(x0,x1,x2,=,y0,y1,y2)
#define ND_SUM(x0,x1,x2)
#define NV_SUM(x)

#define ND_DOT(x0,x1,x2,y0,y1,y2)
#define NV_DOT(x,y)
#define NVD_DOT(x,y0,y1,y2)

#define ND_MAG2(x0,x1,x2)
#define NV_MAG2(x)
#define ND_MAG(x0,x1,x2) sqrt(ND_MAG2(x0,x1,x2))
#define NV_MAG(x) sqrt(NV_MAG2(x))
```

```
#define NV_DST2(x1,x2)
#define NV_DST(x1,x2) sqrt(NV_DST2(x1,x2))

#if RP_3D
# define ND_CROSS_X(x0,x1,x2,y0,y1,y2)(((x1)*(y2))-((y1)*(x2)))
# define ND_CROSS_Y(x0,x1,x2,y0,y1,y2)(((x2)*(y0))-((y2)*(x0)))
#else
/* (x0,x1,0) X (y0,y1,0) */
# define ND_CROSS_X(x0,x1,x2,y0,y1,y2)(0.0)
# define ND_CROSS_Y(x0,x1,x2,y0,y1,y2)(0.0)
#endif
#define ND_CROSS_Z(x0,x1,x2,y0,y1,y2)(((x0)*(y1))-((y0)*(x1)))
#define NV_CROSS_X(x,y)ND_CROSS_X(x[0],x[1],x[2],y[0],y[1],y[2])
#define NV_CROSS_Y(x,y)ND_CROSS_Y(x[0],x[1],x[2],y[0],y[1],y[2])
#define NV_CROSS_Z(x,y)ND_CROSS_Z(x[0],x[1],x[2],y[0],y[1],y[2])
#define NV_CROSS(a,x,y)NV_D(a,=,NV_CROSS_X(x,y),NV_CROSS_Y(x,y),NV_CROSS_Z(x,y))

#if RP_3D
# define ND_ND 3
# define ND_VEC(x,y,z)x,y,z
#else
# define ND_ND 2
# define ND_VEC(x,y,z)x,y
#endif
```