Anejo 3 La programación en ProToolkit

3.1 INTRODUCCIÓN

ProToolkit es una herramienta de programación que permite expandir las posibilidades de Pro/E. Utiliza el lenguaje de programación C/C++ e incluye una extensa librería de funciones para poder acceder y manipular desde el exterior las aplicaciones de Pro/E. Nuestro interés es el de poder transformar el chasis del Anejo 2 dibujado en Pro/E en un archivo utilizable en Cestri y Calest y poder llevar a cabo el cálculo de la estructura. El resultado es una aplicación que tiene por nombre "DeProEaCestri" que lleva a cabo este objetivo satisfactoriamente. Mediante esta aplicación, tal y como se ha visto en el Anejo 2, se obtiene un archivo llamado "chasis.txt", que contiene la geometría del chasis en el formato adecuado.

Como seguramente el lector no estará familiarizado con las funciones y el modo de programar de ProToolkit, previamente se hará una explicación de lo necesario para saber lo que se está haciendo. Eso sí, se va a presuponer que el lector tiene unos conocimientos básicos de la programación en C/C++ y en el diseño con Pro/E.

ProToolkit incluye un buscador online en el que se pueden consultar tanto el manual de usuario como una base de datos con una escueta explicación de cada función de ProToolkit. Esta base de datos es muy útil porque agrupa las funciones según las aplicaciones de Pro/E a las que se refieren. También explica las dependencias entre las diferentes funciones y los archivos "header" necesarios para que puedan funcionar. El manual de usuario también incluye ejemplos que son de gran ayuda para los que se enfrentan por primera vez a la programación en ProToolkit, ya que son simples y aplicables a otros programas de cosecha propia. Queda por decir que para la programación en C/C++ se va a emplear el programa Visual C.

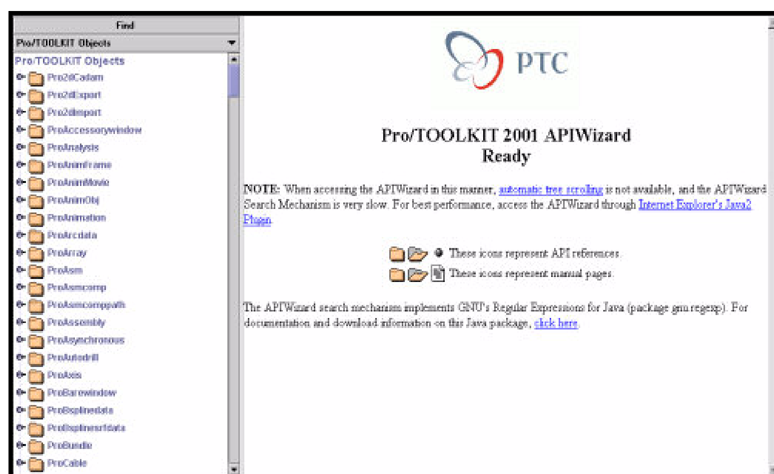

figura 133: El buscador online de ProToolkit

3.2 LA INSTALACIÓN DE PROTOOLKIT

En este apartado se va a tratar sobre cómo hay que preparar el sistema para que una aplicación de ProToolkit pueda ser compilada y linkada correctamente. Según el sistema operativo que esté instalado en el ordenador se deberá actuar de una manera u otra. A continuación se indica de modo esquemático qué es lo que hay que hacer:

3.2.1 Windows 9x

- Incluir en el autoexec.bat: *set PRO_COMM_MSG_EXE=c:\ptc\proe2001\i486_win95\obj*
- Incluir en VisualC: Settings=>Link=>Category=>Input=>Additional library path (All configurations):
  <Ruta de Pro/E>\protoolkit\i486_win95\obj
  <Ruta de Pro/E>\prodevelop\i486_win95\obj

3.2.2 Windows NT/2k

- Añadir al sistema la siguiente variable de entorno: *PRO_COMM_MSG_EXE,* de valor: *<Ruta de Pro/E>*\i486_nt\obj
- Incluir en VisualC: Settings=>Link=>Category=>Input=>Additional library path (All configurations):
  <Ruta de Pro/E>\protoolkit\i486_winNT\obj
  <Ruta de Pro/E>\prodevelop\i486_winNT\obj

3.2.3 Común para ambos tipos de sistemas operativos

- Incluir en la variable "path" de entorno del sistema, o en VisualC: Settings=>C/C++=>Category=>Preprocessor=>Additional include directories (All configurations):
  <Ruta de Pro/E>\protoolkit\includes
  <Ruta de Pro/E>\protoolkit\protk_appls\includes
  <Ruta de Pro/E>\prodevelop\includes
  <Ruta de Pro/E>\prodevelop\prodev_appls\includes
- Incluir en VisualC: Settings=>Link=>Category=>General=>Object library modules (All configurations):
  prodevelop.lib
  pt_asynchronous.lib
  protoolkit.lib
  libc.lib
  wsock32.lib
  mpr.lib

Para que no dé un warning de linkado de que la librería libc.lib da problemas con otras librerías, hay que escribir en el lugar de libc.lib lo siguiente:
/NODEFAULTLIB:libc.lib

3.3 TIPOS DE APLICACIONES PROGRAMABLES

Cuando una aplicación programada en ProToolkit entra en funcionamiento hay dos programas independientes que pasan a la acción: Pro/E y el mismo ProToolkit. Durante la ejecución habrá unas operaciones que las tendrá que realizar ProToolkit y otras Pro/E. Según cómo se haya programado la aplicación, Pro/E y ProToolkit podrán trabajar en paralelo o uno detrás de otro, de manera alternada. Estos dos modos de programar se llaman “modo asíncrono” y “modo síncrono”.

Dentro de estos dos tipos de programación también habrá otros subtipos que difieren en el tipo de comunicación entre Pro/E y ProToolkit durante la ejecución. Habrá que decidir qué modo de programación es el más conveniente para crear la aplicación “DeProEaCestri”. Todo ello se verá a continuación.

3.3.1 Modo asíncrono

Con este tipo de programación se consigue que Pro/E y ProToolkit trabajen en paralelo. Cuando se ejecuta la aplicación el primer programa que se activa puede ser cualquiera de los dos. Lo de trabajar en paralelo es su gran ventaja.

La comunicación entre ambos programas es mediante el proceso llamado rpc (proceso de llamadas remotas). Sin entrar en detalles de qué se trata este proceso hay que decir que es un proceso más lento que el resto. Esta es su gran desventaja. Por esta razón hay que utilizar este modo de programación tan sólo cuando sea estrictamente necesario trabajar en paralelo. Este podría ser el caso de optimizaciones de mecanismos dibujados en Pro/E con programas matemáticos del estilo de Matlab. Para más detalles consultar el proyecto fin de carrera “Comunicación bidireccional entre Matlab y Pro/E. Aplicación a la optimización de un dispositivo háptico bajo criterios mecánicos”. Javier Martín Amézaga. 2002. Existen dos subtipos de programación en modo asíncrono: el “modo asíncrono simple” y el “modo asíncrono completo”.

Modo asíncrono simple

En la intercomunicación entre programas, Pro/E no puede hacer llamadas a ProToolkit. Un ejemplo en el que se pueda ver sus repercusiones sería el que no se pueden añadir botones a Pro/E que hagan funcionar la aplicación programada.

Modo asíncrono completo

En este caso existen funciones de ProToolkit como *ProEventProcess*() y *ProTermFuncSet*() con las cuales ProToolkit puede “escuchar” los mensajes desde Pro/E.

3.3.2 Modo síncrono

Este modo es el más usual debido a su simplicidad. Durante la ejecución de la aplicación Pro/E y ProToolkit trabajan de manera alternada y el primero que entra en funcionamiento es Pro/E. Otra diferencia respecto al modo asíncrono es que la función *main()* del programa no lo controla es usuario sino Pro/E. El usuario tiene la posibilidad de insertar su código dentro de las funciones *user_initialize*() y *user_terminate*(). La primera función entra en funcionamiento cuando se activa la aplicación y la segunda cuando se desactiva. Esto último puede parecer un poco absurdo, pero puede que alguien le encuentre utilidad. Lo normal es programar dentro de la función *user_initialize*(). Esta función siempre debe dar un valor de retorno aunque no sea trascendente. Podría ser algo así como: *return(0)*. La función *user_terminate*() no tiene porqué devolver nada, pero ha de ser escrita aunque sea vacía de contenido. Existen dos subtipos de programación en modo síncrono: el “modo DLL” y el “modo multiproceso”.

Modo DLL

Se podría decir que este es el modo estándar de programación de aplicaciones ProToolkit. La comunicación entre Pro/E y ProToolkit es mediante las denominadas funciones de llamada directa. Para más detalles al respecto consultar el manual.

Modo multiproceso

La comunicación entre Pro/E y ProToolkit es mediante un sistema de mensajes que simulan funciones de llamada directa pasando la información necesaria para identificar la función y los valores de sus argumentos entre los dos procesos. Su ventaja es que permite ejecutar la aplicación con un “debugger” sin tener que cargar el ejecutable de Pro/E en el “debugger”, con lo que el proceso de “debug” es más rápido.

3.4 REGISTRAR LA APLICACIÓN

Registrar una aplicación de ProToolkit quiere decir proveer información a Pro/E sobre los ficheros que componen dicha aplicación. Para ello hace falta un fichero de texto llamado “archivo de registro”. Este fichero se carga en Pro/E desde el menu superior de la pantalla. Concretamente en “Utilities”, “Auxiliary aplications”. Se abre una ventana en la que hay un botón llamado “Register”. Si lo clickamos se puede elegir el archivo de registro. Este archivo normalmente se llamará “protk.dat”. Su contenido básico debe ser el siguiente:

name NOMBREDELARCHIVO

startup dll *(si el archivo es .dll, si no nada)*

exec_file NOMBREDELARCHIVO.dll (o *exec_file NOMBREDELARCHIVO.exe*)

text_dir RUTADELARCHIVO

revision NUMERODEVERSIONDEPROE

end

Si se quiere ejecutar en modo multiproceso en lugar de *startup dll*, hay que escribir *startup spawn*. La versión de Pro/E que estamos utilizando es la 2001. Éste es el número que deberíamos escribir donde antes hemos puesto NUMERODEVERSIONDEPROE.

Si quisieramos tener la posibilidad de parar la aplicación una vez ejecutada deberíamos añadir al texto anterior lo siguiente:

allow_stop TRUE

Si quisieramos que la aplicación entrase en funcionamiento únicamente cuando nosotros se lo indicáramos explicitamente, deberíamos añadir al texto anterior lo siguiente:

delay_start TRUE

3.5 DESBLOQUEAR LA APLICACIÓN

Una vez programada y compilada la aplicación, para que la pueda usar cualquier usuario, tenga o no ProToolkit, es necesario desbloquearla. Para ello hay que ejecutar el siguiente commando:

<Pro/ENGINEER>/bin/protk_unlock RUTADELARCHIVO/NOMBREDELARCHIVO.EXTENSION

Siendo <Pro/ENGINEER> la ruta donde está instalado Pro/E.

3.6 LA APLICACIÓN "DEPROEACESTRI"

Tal y como se ha indicado en el Anejo 2, esta aplicación tiene el propósito de transformar la geometría Pro/E a geometría Cestri y Calest para poder calcularla estáticamente. El modo de programación que vamos a emplear para programar la aplicación es el modo dll. La aplicación tendrá la extensión *.exe* y tendrá por nombre "DeProEaCestri". Sabiendo todo esto ya podemos decir que el archivo de registro "protk.dat" va a tener el siguiente contenido:

name DeProeaCestri

exec_file DeProeaCestri.exe

text_dir RUTADELARCHIVO

revision 2001

delay_start TRUE

allow_stop TRUE

end

En Visual C, lo primero que hay que hacer es crear un nuevo *workspace*, de extensión *.dsw*. Al clickar sobre *New*, se abre una ventana con varias pestañas. La que está abierta es la pestaña *Projects*. En ella hay que elegir el tipo de proyecto. Nuestro caso es el *Win32 Console Application*. Se le da un nombre y se

elige la ruta donde se quiere tener dicho proyecto. Se clicka sobre *OK* y se abre otra ventana. En esta elegimos la opción por defecto *An empty project* y clickamos sobre *Finish*. No nos queda más por elegir. Llegaremos a una pantalla en la que tendremos a la izquierda una ventana con dos pestañas: *ClassView* y *FileView*. En *FileView* veremos lo siguiente:

figura 134: Las carpetas donde se inserta el código.

Las carpetas que vamos a llenar con archivos son "Source Files" y "Header Files". En "Source Files" irá los archivos que componen el código del programa; el principal y las funciones programadas por el usuario. En "Header Files" irá un archivo de extensión *.h* necesario porque contiene las definiciones de las funciones definidas por el usuario. La ventana de la figura 2 quedará al final tal y como se muestra a continuación:

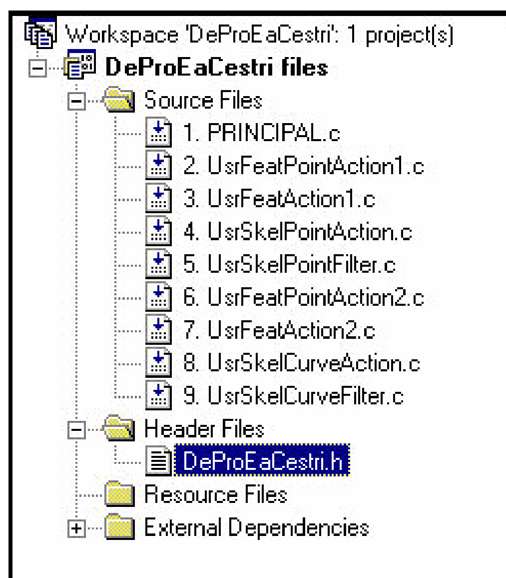

figura 135: El estado final de las carpetas.

La carpeta "External Dependencies" las gestiona el ordenador tras compilar la aplicación, por lo que no hay que preocuparse de ella. Hay que decir que la compilación, para que la aplicación pueda darse por definitiva, debe hacerse en modo "release" y no en modo "debug". El código de todos estos archivos se muestra a continuación:

1. PRINCIPAL.c

```c
/*=======================================================================================*\
ESTA APLICACIÓN ES PARTE DEL SIGUIENTE PROYECTO FIN DE CARRERA DE LA ESCUELA DE INGENIEROS
INDUSTRIALES DE SAN SEBASTIÁN, TECNUN:
"DISEÑO, CONSTRUCCIÓN Y CÁLCULO ESTÁTICO DE CHASIS TUBULARES PARA COCHES DE COMPETICIÓN"
AUTOR: LUI
FECHA: 5/3/2002. Martes.
\*=======================================================================================*/

/*=======================================================================================*\
ARCHIVOS INCLUDE DE PROTOOLKIT
\*=======================================================================================*/
#include <ProToolkit.h>      //Necesario para todo programa de ProToolkit. Contiene la definición los wide string y
                             //los siguientes archivos header: stdio.h, string.h, stddef.h y stdlib.h.
#include <malloc.h>          //Define calloc()
#include <ProSkeleton.h>     //Define ProAsmSkeletonGet()
#include <ProUdf.h>          //Define ProSolidGroupVisit()
#include <ProUtil.h>         //Define ProStringToWstring()
#include <ProSolid.h>        //Define ProSolidInit()
#include <ProArray.h>        //Define ProArrayAlloc() y ProArraySizeGet()
#include <ProPoint.h>        //Define ProPointIdGet() y ProPointCoordGet()
#include "DeProEaCestri.h"   //Define UsrSkelPointAction(), UsrSkelPointFilter(), UsrSkelCurveAction() y
                             //UsrSkelCurveFilter()
/*=======================================================================================*\
LA APLICACIÓN SÍNCRONA
\*=======================================================================================*/
int user_initialize()
{
	FILE *fp;
	char nombre_assembly[30];
	ProName wnombre;
	ProType tipo = PRO_ASSEMBLY;
	ProError error;
	ProSolid assembly;
	ProMdl skeleton;
	ProPoint *puntos;
	int *padres_rectas1;
	int **padres_rectas2;
	int **padres_rectas3;
	ProVector *coord_puntos;
	int n_puntos, n_rectas, p, q, *id_puntos, diametro, espesor, E, G;
	double A, Ix, Iy, Iz;
	double Pi = 3.141592654;
```

```
/*-----------------------------------------------------------------------------------------------------------------------------------------------*\
 Se recurre al nombre del assembly desde un archivo llamado "assembly.txt".
\*-----------------------------------------------------------------------------------------------------------------------------------------------*/
        fp = fopen("assembly.txt", "r");
        fscanf(fp, "%s", nombre_assembly);
        fclose(fp);
        printf("NOMBRE DEL ASSEMBLY=%s\n\n", nombre_assembly);
/*-----------------------------------------------------------------------------------------------------------------------------------------------*\
 Se inicializa el assembly como variable a partir del nombre obtenido en el archivo y del assembly que está
ejecutado en Pro/E cuyo nombre es el mismo que el que está escrito en "assembly.txt".
\*-----------------------------------------------------------------------------------------------------------------------------------------------*/
        ProStringToWstring(wnombre, nombre_assembly);
        error = ProSolidInit(wnombre, tipo, &assembly);
        if (error !=  PRO_TK_NO_ERROR)
        {
                printf("INICIALIZACION DEL ASSEMBLY INCORRECTA!\n\n");
                return(0);
        }
/*-----------------------------------------------------------------------------------------------------------------------------------------------*\
Se obtiene el esqueleto del assembly y de éste se recogen los puntos y las curvas que forman el chasis.
\*-----------------------------------------------------------------------------------------------------------------------------------------------*/
        ProArrayAlloc(0, sizeof(ProPoint), 1, (ProArray*)&puntos);
        ProArrayAlloc(0, sizeof(int), 1, (ProArray*)&padres_rectas1);
        ProAsmSkeletonGet(assembly, &skeleton);

        ProSolidGroupVisit(skeleton, UsrSkelPointAction, UsrSkelPointFilter, &puntos);
        PoSolidGroupVisit(skeleton, UsrSkelCurveAction, UsrSkelCurveFilter, &padres_rectas1);

        ProArraySizeGet(puntos, &n_puntos);
        ProArraySizeGet(padres_rectas1, &n_rectas);

        n_rectas = n_rectas/2;
/*-----------------------------------------------------------------------------------------------------------------------------------------------*\
Se guardan las ID de los padres de las rectas en el array "padres_rectas2".
\*-----------------------------------------------------------------------------------------------------------------------------------------------*/
        padres_rectas2 = (int **)calloc(n_rectas, sizeof(int*));

        for (p=0; p<n_rectas; p=p++)
        {
                padres_rectas2[p] = (int *)calloc(2, sizeof(int));
                padres_rectas2[p][0] = padres_rectas1[2*p];
                padres_rectas2[p][1] = padres_rectas1[2*p+1];
        }
```

```
    printf("NUMERO DE PUNTOS=%d\n\n", n_puntos);
    printf("NUMERO DE RECTAS=%d\n\n", n_rectas);
```

/*--*\

Se guardan las ID y las coordenadas de los puntos en los array "id" y "coordenadas".

--/

```
    coord_puntos = (double (*)[3])calloc(n_puntos, sizeof(ProVector));
    id_puntos = (int *)calloc(n_puntos, sizeof(int));

    for(p=0; p<n_puntos; p++)
    {
        ProPointIdGet(puntos[p], &id_puntos[p]);
        ProPointCoordGet(puntos[p], coord_puntos[p]);
    }
```

/*--*\

Se renombran las ID de los puntos con números "normalizados".

--/

```
    padres_rectas3 = (int **)calloc(n_rectas, sizeof(int*));

    for (p=0; p<n_rectas; p++)
    {
        padres_rectas3[p] = (int *)calloc(2, sizeof(int));

        for (q=0; q<n_puntos; q++)
        {
            if (id_puntos[q] == padres_rectas2[p][0])
                padres_rectas3[p][0] = q+1;

            if (id_puntos[q] == padres_rectas2[p][1])
                padres_rectas3[p][1] = q+1;
        }
    }
```

/*--*\

Se recogen las carecterísticas del perfil más común del archivo "perfil.txt". Las unidades están en mm, pero las características de área y momentos de inercia deberán estar en cm con su correspondiente potencia.

--/

```
    fp = fopen("perfil.txt", "r");
    fscanf(fp, "%d %d", &diametro, &espesor);
    fclose(fp);

    A = Pi/4*(((double)diametro/10.)*((double)diametro/10.)-(((double)diametro-2*(double)espesor)/10.)
    *(((double)diametro-2*(double)espesor)/10.));

    Iy = Pi/64*(((double)diametro/10.)*((double)diametro/10.)*((double)diametro/10.)*((double)diametro/10.)-
    (((double)diametro-2*(double)espesor)/10.)*(((double)diametro-2*(double)espesor)/10.)*(((double)diametro-
    2*(double)espesor)/10.)*(((double)diametro-2*(double)espesor)/10.));

    Iz = Iy;
```

```
        Ix = 2*Iy;
        printf("DIAMETRO=%d mm, ESPESOR=%d mm, A=%.2f cm2, Ix=%.2f cm4, Iy=Iz=%.2f cm4\n\n", diametro, espesor, A, Ix, Iy);
/*----------------------------------------------------------------------------------------------------------------------------*\
Se recogen las carecterísticas del material del archivo "material.txt". Las unidades están en kg/cm2.
\*----------------------------------------------------------------------------------------------------------------------------*/
        fp = fopen("material.txt", "r");
        fscanf(fp, "%d %d", &E, &G);
        fclose(fp);
        printf("MODULO DE ELASTICIDAD=%d kg/cm2, MODULO DE CORTADURA=%d kg/cm2\n\n", E, G);
/*----------------------------------------------------------------------------------------------------------------------------*\
Se escriben los datos en un fichero de texto para poder exportarlo a CESTRI.
\*----------------------------------------------------------------------------------------------------------------------------*/
        fp = fopen("chasis.txt", "w");

        fprintf(fp, "ELAS %d\n", E);
        fprintf(fp, "GCOR %d\n\n", G);

        for(p=0; p<n_puntos; p++)
        {
                fprintf(fp, "NUDO %d   %.2f %.2f %.2f\n", p+1, (coord_puntos[p][2]/10), (coord_puntos[p][0]/10), (coord_puntos[p][1]/10));
                //Las coordenadas se cambian de posición para una mejor visualización en Cestri: x->y, y->z, z->x.
                //Además se pasan de mm a cm.
        }
        fprintf(fp, "\n");

        for(p=0; p<n_rectas; p++)
        {
                fprintf(fp, "VI3EE %d  %d %d  %.2f %.2f %.2f %.2f  0  0  0 0 0\n", (p+1), padres_rectas3[p][0], padres_rectas3[p][1], A, Ix, Iy, Iz);
        }

        fclose(fp);
/*----------------------------------------------------------------------------------------------------------------------------*\
Mensajes finales.
\*----------------------------------------------------------------------------------------------------------------------------*/
        printf("CHASIS EXPORTADO A FORMATO CESTRI CORRECTAMENTE.\n\n");
        printf("EL ARCHIVO SE LLAMA 'CHASIS.TXT'. PUEDE FINALIZAR EL PROGRAMA.\n");

        return(0);
}
void user_terminate()
{
}
```

2. UsrFeatPointAction1.c

```
#include <ProGeomitem.h>    //Define ProGeomitemToPoint()
#include <ProArray.h>       //Define ProArrayObjectAdd()
/*==========================================================================================*\
FUNCIÓN: UsrFeatPointAction1()
PROPÓSITO: Añade un punto de un grupo al final de un array. (puntos)
\*==========================================================================================*/
ProError UsrFeatPointAction1(
        ProGeomitem *p_handle,
        ProError status,
        ProAppData data)
{
        ProPoint punto;

        ProGeomitemToPoint(p_handle, &punto);
        ProArrayObjectAdd(data, -1, 1, &punto);

        return(PRO_TK_NO_ERROR);
}
```

3. UsrFeatAction1.c

```
#include <ProFeature.h>         //Define ProFeatureTypeGet() y ProFeatureGeomitemVisit()
#include "DeProEaCestri.h"      //Define UsrFeatPointAction1()
/*==========================================================================================*\
FUNCIÓN: UsrFeatAction1()
PROPÓSITO: Visita las geomitems de las features. (puntos)
\*==========================================================================================*/
ProError UsrFeatAction1(
        ProFeature* feature,
        ProError status,
        ProAppData data)
{
        ProFeattype p_type;

        ProFeatureTypeGet(feature, &p_type);

        if((int)p_type == 931)
        {
                ProFeatureGeomitemVisit(feature, p_type, UsrFeatPointAction1, NULL, data);
        }

        return(PRO_TK_NO_ERROR);
}
```

4. UsrSkelPointAction.c

```
#include <ProUdf.h>            //Define ProGroupFeatureVisit()
#include "DeProEaCestri.h"     //Define UsrFeatAction1()
/*=====================================================================================*\
FUNCIÓN: UsrSkelPointAction()
PROPÓSITO: Visita las features contenidas en el grupo del esqueleto del assembly y los recopila en un array. (puntos)
\*=====================================================================================*/
ProError UsrSkelPointAction(
        ProGroup *group,
        ProError status,
        ProAppData data)
{
        ProGroupFeatureVisit(group, UsrFeatAction1, NULL, data);

        return(PRO_TK_NO_ERROR);
}
```

5. UsrSkelPointFilter.c

```
#include <ProUdf.h>            //Define ProUdfNameGet()
#include <ProUtil.h>           //Define ProWstringToString()
#include <string.h>            //Define strcmp()
/*=====================================================================================*\
FUNCIÓN: UsrSkelPointFilter()
PROPÓSITO: Sólo deja que se visite el grupo "PUNTOS_CHASIS". (puntos)
\*=====================================================================================*/
ProError UsrSkelPointFilter(
        ProGroup *group,
        ProAppData data)
{
        FILE *fp;
        ProName wnombre;
        ProName name;
        char nombre1[30];
        char nombre2[30];

        fp = fopen("puntos.txt", "r");        //Las mayúsculas y las minúsculas las distingue.
        fscanf(fp, "%s", nombre1);
        fclose(fp);

        ProUdfNameGet(group, wnombre, name);
        ProWstringToString(nombre2, wnombre);
```

```
        if (strcmp(nombre1, nombre2) != 0)
                return(PRO_TK_CONTINUE);

        else
                return(PRO_TK_NO_ERROR);
}
```

6. UsrFeatPointAction2.c

```
#include <ProGeomitem.h>    //Define ProGeomitemToPoint()
#include <ProPoint.h>       //Define ProPointIdGet()
#include <ProArray.h>       //Define ProArrayObjectAdd()
/*=========================================================================================*\
FUNCIÓN: UsrFeatPointAction2()
PROPÓSITO: Recoge los ids de los padres de las rectas. (rectas)
\*=========================================================================================*/
ProError UsrFeatPointAction2(
        ProGeomitem *p_handle,
        ProError status,
        ProAppData data)
{
        ProPoint punto;
        int id;

        ProGeomitemToPoint(p_handle, &punto);
        ProPointIdGet(punto, &id);
        ProArrayObjectAdd(data, -1, 1, &id);

        return(PRO_TK_NO_ERROR);
}
```

7. UsrFeatAction2.c

```
#include <ProFeature.h>       //Define ProFeatureTypeGet(), ProFeatureParentsGet(), ProFeatureInit(),
                              //ProFeatureSolidGet() y ProFeatureGeomitemVisit()
#include "DeProEaCestri.h"    //Define UsrFeatPointAction2()
/*=========================================================================================*\
FUNCIÓN: UsrFeatAction2()
PROPÓSITO: Visita los padres de las rectas. (rectas)
\*=========================================================================================*/
ProError UsrFeatAction2(
        ProFeature* feature,
        ProError status,
        ProAppData data)
```

```
{
	ProFeattype p_type1;
	ProFeattype p_type2 = 931;
	int *p_parents;
	int p_count;
	ProSolid p_mdl_handle;
	ProFeature p_feat_handle[2];

	ProFeatureTypeGet(feature, &p_type1);

	if((int)p_type1 == 949)
	{
		ProFeatureParentsGet(feature, &p_parents, &p_count);
		ProFeatureSolidGet(feature, &p_mdl_handle);
		ProFeatureInit(p_mdl_handle, p_parents[0], &p_feat_handle[0]);
		ProFeatureInit(p_mdl_handle, p_parents[1], &p_feat_handle[1]);
		ProFeatureGeomitemVisit(&p_feat_handle[0], p_type2, UsrFeatPointAction2, NULL, data);
		ProFeatureGeomitemVisit(&p_feat_handle[1], p_type2, UsrFeatPointAction2, NULL, data);
	}

	return(PRO_TK_NO_ERROR);
}
```

8. UsrSkelCurveAction.c

```
#include <ProUdf.h>          //Define ProGroupFeatureVisit()
#include "DeProEaCestri.h"   //Define UsrFeatAction2()
/*==========================================================================================*\
FUNCIÓN: UsrSkelCurveAction()
PROPÓSITO: Visita las features contenidas en el grupo del esqueleto del assembly y los recopila en un array.
(rectas)
\*==========================================================================================*/
ProError UsrSkelCurveAction(
	ProGroup *group,
	ProError status,
	ProAppData data)
{
	ProGroupFeatureVisit(group, UsrFeatAction2, NULL, data);

	return(PRO_TK_NO_ERROR);
}
```

9. UsrSkelCurveFilter.c

```
#include <string.h>            //Define strcmp()
#include <ProUdf.h>            //Define ProUdfNameGet()
#include <ProUtil.h>           //Define ProWstringToString()
/*==========================================================================================*\
FUNCIÓN: UsrSkelCurveFilter()
PROPÓSITO: Sólo deja que se visite el grupo "RECTAS_CHASIS". (rectas)
\*==========================================================================================*/
ProError UsrSkelCurveFilter(
        ProGroup *group,
        ProAppData data)
{
        FILE *fp;
        ProName wnombre;
        ProName name;
        char nombre1[30];
        char nombre2[30];

        fp = fopen("rectas.txt", "r");      //Las mayúsculas y las minúsculas las distingue.
        fscanf(fp, "%s", nombre1);
        fclose(fp);

        ProUdfNameGet(group, wnombre, name);
        ProWstringToString(nombre2, wnombre);

        if (strcmp(nombre1, nombre2) != 0)
                return(PRO_TK_CONTINUE);

        else
                return(PRO_TK_NO_ERROR);
}
```

DeProEaCestri.h

```
#include <ProUdf.h>        //Define UsrSkelPointAction(), UsrSkelPointFilter(), UsrSkelCurveAction() y
                           //UsrSkelCurveFilter()
#include <ProSolid.h>      //Define UsrFeatAction1() y UsrFeatAction2()
#include <ProObjects.h>    //Define UsrFeatPointAction1() y UsrFeatPointAction2()

ProError UsrFeatPointAction1(ProGeomitem *, ProError, ProAppData);
ProError UsrFeatAction1(ProFeature *, ProError, ProAppData);
ProError UsrSkelPointAction(ProGroup *, ProError, ProAppData);
ProError UsrSkelPointFilter(ProGroup *, ProAppData);
ProError UsrFeatPointAction2(ProGeomitem *, ProError, ProAppData);
ProError UsrFeatAction2(ProFeature *, ProError, ProAppData);
ProError UsrSkelCurveAction(ProGroup *, ProError, ProAppData);
ProError UsrSkelCurveFilter(ProGroup *, ProAppData);
```

Antes de comenzar con la explicación del programa se va a indicar cómo debe ser un archivo utilizable en Cestri y Calest. Este archivo debe ser de texto con extensión *.txt*. La forma general que tiene es la siguiente:

ELAS *VALORDELMODULOELASTICO*
GCOR *VALORDELMODULODECORTADURA*

NUDO *id x y z*
…

VI3EE *id idn1 idn2 area ix iy iz modoLoc angFi Xaux Yaux Zaux*
…

Las dos primeras líneas son las características mecánicas del material, en cuanto a los módulos elástico y de cortadura. La siguiente sentencia es la manera en la que se definen los nudos. En ese caso *id* es el número de identificación del nudo y *x y z* son sus coordenadas. Los puntos suspensivos después de esta línea son una manera de representar que hay más de un nudo. No hay que escribirlos en el archivo. La siguiente sentencia es la forma en la que se definen vigas biempotradas espaciales. La explicación de sus argumentos es la siguiente:

id: Identificador de la barra. Puede ser cualquier valor numérico. No es necesario que la numeración de las barras sea correlativa.

idn1: Identificador del nudo inicial de la barra. Debe corresponder a un nudo ya definido.

idn2: Identificador del nudo final de la barra. Debe corresponder a un nudo ya definido.

area: Área de la sección transversal de la barra.

ix: Momento de inercia de la sección transversal de la barra respecto al eje X local. La rigidez a torsión de la barra se evalúa como el producto de ix por el módulo de elasticidad en cortadura, dividido por la longitud.

iy: Momento de inercia de la sección transversal de la barra respecto al eje Y local.

iz: Momento de inercia de la sección transversal de la barra respecto al eje Z local.

modoLoc: Índice que indica el modo de definición de sistema local de la barra. Vale 0 cuando el sistema local se define por medio del ángulo Y, y vale 1 cuando se usa el método del punto auxiliar (apartado 7.8.2 del libro Curso de Análisis Estructural. EUNSA. Juan Tomás Celigüeta. 1998).

angFi: Valor del ángulo Y que define la posición del sistema local.

Xaux, Yaux, Zaux: Coordenadas del punto auxiliar usado para definir el sistema local. Corresponden a un punto P cualquiera situado en el plano X, Y local, fuera del eje X local, y están medidas en el sistema de ejes general de la estructura.

Ésta es la forma del archivo objetivo. Recordemos que el dibujo del chasis lo hemos hecho con puntos y rectas. Estos puntos Pro/E se convertirán en nudos Cestri y las rectas Pro/E en vigas biempotradas espaciales Cestri. A continuación se explican los pasos que se dan para conseguir nuestro propósito. Se advierte que la explicación es muy pero que muy densa, por lo que requerirá más de una y dos lecturas y siempre con un ojo puesto en el código antes expuesto.

1. El programa busca en el mismo directorio en el que está, el archivo “assembly.txt”. Lo abre y recoge su contenido en una variable tipo *char*. Lo que se recoge es el nombre del *assembly* del que queremos obtener la geometría del chasis. En nuestro ejemplo es *carcross.asm.*

2. Mediante la función *ProStringToWstring*() se transforma esta variable *char* al formato *wide string*. Esto es necesario para después poder manipular el *assembly* con otras funciones de ProToolkit.

3. Se inicializa el *assembly* con la función *ProSolidInit*(). Si no lo hace correctamente, la función devuelve un valor de retorno que lo utilizamos para finalizar la aplicación. Aparecerá en la ventana de MS-DOS el siguiente mensaje: “INICIALIZACION DEL ASSEMBLY INCORRECTA!”.

4. Se reserva espacio para lo que van a ser los vectores *puntos* y *padres_rectas1*. En el primero de ellos se van a insertar los puntos en formato Pro/E. Este tipo de variables se llaman *ProPoint*. El segundo se usará para guardar ordenadamente los números de identificación *id* de los puntos de los que dependan las rectas. Ambos se van a rellenar de forma dinámica, por eso se hace mediante la función *ProArrayAlloc*(). Lo de reservar memoria dinámicamente quiere decir que dependiendo del número de puntos que tenga el *assembly*, variará el tamaño de estos vectores.

5. Mediante la función *ProAsmSkeletonGet*() se extrae el esqueleto del *assembly* y se guarda en la variable *skeleton*, que es de tipo *ProMdl*. En nuestro ejemplo nos referimos a *carcross_skel.prt*.

6. Con la función *ProSolidGroupVisit*() entramos en la parte más densa del programa. Esta función se utiliza dos veces en el programa principal. La primera para obtener información de los puntos y la segunda lo mismo para las rectas. Los argumentos de esta función requieren una explicación. Estos argumentos son: 1. El sólido del cual se quieren visitar sus grupos, 2. Una función llamada "función de acción", 3. Una función llamada "función de filtro" y 4. Una variable externa que interese al usuario para ser manipulada dentro de la función. La primera vez que se utiliza esta función es para visitar los grupos del esqueleto del *assembly*. Se le pasa como variable externa el vector *puntos*. Dentro de esta primera función ocurre lo que se relata a continuación:

6.1 La "función de filtro" hace lo siguiente: Busca en el mismo directorio en el que está la aplicación, un archivo llamado "puntos.txt". Lo abre y recoge de su interior el nombre del grupo en el que, valga la redundancia, se han agrupado los puntos en el dibujo de Pro/E. En nuestro caso es "PUNTOS_CHASIS". Este nombre lo guarda en formato *char*. Por otro lado, con la función *ProUdfNameGet*() se obtiene el nombre del grupo que se está visitando en el esqueleto y se guarda en una variable de formato *wide string*. Con la función *ProWstringToString*() se transforma esta última variable a formato *char*. Con la función *strcmp*() se comparan ambas variables en formato *char*. Si coinciden, la "función filtro" deja que entre en funcionamiento la "función de acción".

6.2 La "función de acción" hace lo siguiente: La función *ProGroupFeatureVisit*() visita las *features* que componen el grupo. En nuestro caso cada *feature* es un punto. Esta función no tiene "función de filtro" y como argumento externo se le pasa el vector *puntos*. Dentro de esta función ocurre lo siguiente:

6.2.1 La "función de acción" de esta "función de visita" hace lo siguiente: La función *ProFeatureTypeGet*() obtiene el tipo de *feature* que se está visitando. Si el tipo de *feature* es 931, es decir, si es un punto, se ejecuta la función *ProFeatureGeomitemVisit*(). Este último paso requiere una explicación adicional: un elemento punto es una variable *ProPoint*. El *ProPoint* es lo que en lenguaje Pro/E se conoce como *Geomitem*. Una *feature* puede contener *geomitems*. Por lo tanto lo que se está haciendo es primero visitar la *feature* y obtener los *geomitems* que contiene. Tal y como hemos dibujado, cada *feature* sólo contiene una *geomitem*, que es un punto. Un grupo como es el "PUNTOS_CHASIS" tiene como primera *feature* siempre el propio grupo en sí. ¿Qué quiere decir esto? Que la primera *feature* tipo punto no es la primera del grupo sino la segunda. Por esta razón se hace la sentencia *if* para visitar únicamente las *features* tipo punto del grupo. Dentro de la función *ProFeatureGeomitemVisit*() ocurre lo siguiente:

6.2.1.1 Esta función no tiene filtro. Dentro de su "función de acción" lo que ocurre es lo que se relata a continuación: Con la función *ProGeomitemToPoint*() se transforma la *geomitem* de la *feature* en una variable tipo *ProPoint*. Esta variable se añade al final del vector externo *puntos*. De esta manera estamos llenando de

los puntos que nos interesan, los del grupo “PUNTOS_CHASIS”, en el vector *puntos*.

7. Dentro de la segunda función *ProSolidGroupVisit*() ocurre lo siguiente:

7.1 La “función de filtro” actúa de manera análoga a la de la anterior función *ProSolidGroupVisit*(). La diferencia está en que el archivo que busca es el llamado “rectas.txt”. El nombre del grupo en este caso es “RECTAS_CHASIS”.

7.2 La “función de acción” es igual a la antes vista con la única diferencia de los argumentos de la función *ProGroupFeatureVisit*() que contiene.

7.2.1 La “función de acción” de esta “función de visita” hace lo siguiente: Al igual que su equivalente vista anteriormente, primero filtra aquellas *features* que sean de tipo recta con una sentencia *if*. Después, con la función *ProFeatureParentsGet*() obtiene las *id* los padres de las rectas. Tal y como hemos dibujado las rectas, los únicos padres que tiene son los dos puntos que unen cada una de ellas. Con la función *ProFeatureSolidGet*() se obtiene el sólido que contiene la *feature* que estamos visitando. La función *ProFeatureInit*() necesita como argumentos, este sólido y la *id* del punto que se quiere inicializar. Con esta función se activa la *feature* que contiene el punto padre de la recta que estamos visitando. Por supuesto que se deben activar ambos padres de cada recta. Ya tenemos los *features* activados. Sólo nos queda extraer los *geomitem* tipo punto de estas *features*. Esto se hace con la función *ProFeatureGeomitemVisit*(). Lo que ocurre en esta función se explica a continuación:

7.2.1.1 Esta función no tiene filtro. La “función de acción” hace lo siguiente: La función *ProGeomitemToPoint*() transforma el *geomitem* en una variable tipo *ProPoint*. La función *ProPointIdGet*() obtiene el *id* del punto y este *id* se almacena en el vector externo padres_rectas1. Este rocambolesco modo de obtener los *id*, se hace de esta manera para que se tengan los padres de cada recta ordenados en un vector, primero los dos padres de una recta, después los dos padres de otra recta y así sucesivamente.

8. Llegados a este punto se tienen dos vectores llenos de información útil: *puntos* y *padres_rectas1*. Con la función *ProArraySizeGet*() se obtiene el tamaño de estos dos vectores. El número de puntos del dibujo es el tamaño del vector *puntos*, mientras el número de rectas es el tamaño del vector *padres_rectas1* dividido entre 2.

9. Con la función *calloc*() se reserva memoria estáticamente para las filas de lo que va a ser la matriz *padres_rectas2*. Esta matriz va a tener en cada fila las *id* de los puntos padre de cada recta. Lo que se está haciendo con el *for*, es reordenar la información de *padres_rectas1* en la matriz *padres_rectas2*.

10. Una vez más con la función *calloc*() se reserva memoria estáticamente, esta vez para los arrays *coord_puntos* e *id_puntos*.

11. Con la función *ProPointIdGet*() se obtienen las *id* de los componentes del vector *puntos* y se guardan en formato *int* en *id_puntos*.

12. Con la función *ProPointCoordGet*() se obtienen las coordenadas de los componentes del vector puntos y se guardan en formato *int* en *coord_puntos*.
13. Las *id* recogidas en *padres_rectas2* tienen unas cifras del orden de millares y por supuesto que no son consecutivos. Resumiendo, son números "feos", no presentables. Nuestra intención es que la primera *id* sea 1 y que las demás sean 2, 3, 4 y así sucesivamente. Estos números hay que "normalizarlos". La matriz *padres_rectas2* con el siguiente *for*, pasará su información "normalizada" a *padres_rectas3*.
14. Se recogen las medidas del perfil común a todas las barras desde el archivo "perfil.txt". En este caso es 40 2, es decir, diámetro exterior de 40 mm y espesor de 2 mm. A partir de estas medidas se calculan las características de rigidez.
15. Se recogen las características mecánicas del material desde el archivo "material.txt". En este caso son 2100000 800000, es decir, E = 2100000 kg/cm^2 y G = 800000 kg/cm^2.
16. Se abre un archivo llamado "chasis.txt". En él se escribe en formato Cestri/Calest toda la información obtenida hasta ahora. Éste es el momento en el que estamos capacitados para hacerlo. En el código se puede ver con detalle este punto.
17. Si todo ha ido bien, la función *user_initialize*(), donde hemos escrito todo el código podrá dar el valor de retorno *return*(0) con el que finaliza su cometido. La función *user_terminate*() no tiene código.

En este ejemplo el archivo de salida "chasis.txt" queda como se muestra a continuación:

```
ELAS 2100000
GCOR 800000

NUDO 1  156.10 30.00 59.73
NUDO 2  156.10 -30.00 59.73
NUDO 3  150.00 25.00 -10.00
NUDO 4  160.06 25.00 105.00
NUDO 5  160.06 -25.00 105.00
NUDO 6  150.00 -25.00 -10.00
NUDO 7  90.00 25.00 98.87
NUDO 8  90.00 -25.00 98.87
NUDO 9  60.10 30.00 47.09
NUDO 10  60.10 -30.00 47.09
NUDO 11  44.05 25.00 19.28
NUDO 12  44.05 -25.00 19.28
NUDO 13  44.05 25.00 -10.00
NUDO 14  44.05 -25.00 -10.00
NUDO 15  -5.95 25.00 -10.00
NUDO 16  -5.95 -25.00 -10.00
NUDO 17  -5.95 25.00 19.28
NUDO 18  -5.95 -25.00 19.28
NUDO 19  19.05 25.00 34.28
```

NUDO 20 19.05 -25.00 34.28
NUDO 21 -17.50 -25.00 2.50
NUDO 22 -17.50 25.00 2.50
NUDO 23 257.50 -25.00 -10.00
NUDO 24 257.50 25.00 -10.00
NUDO 25 217.50 -25.00 -10.00
NUDO 26 217.50 25.00 -10.00
NUDO 27 257.50 25.00 15.00
NUDO 28 217.50 25.00 15.00
NUDO 29 257.50 -25.00 15.00
NUDO 30 217.50 -25.00 15.00
NUDO 31 237.50 25.00 55.00
NUDO 32 237.50 -25.00 55.00
NUDO 33 217.50 -7.50 -10.00
NUDO 34 217.50 7.50 -10.00
NUDO 35 208.50 7.50 1.00
NUDO 36 208.50 -7.50 1.00
NUDO 37 217.50 -7.50 15.00
NUDO 38 217.50 7.50 15.00
NUDO 39 211.50 -7.50 22.50
NUDO 40 211.50 7.50 22.50
NUDO 41 177.50 -7.50 -10.00
NUDO 42 177.50 -7.50 11.00
NUDO 43 177.50 7.50 -10.00
NUDO 44 177.50 7.50 11.00
NUDO 45 150.00 -7.50 -10.00
NUDO 46 150.00 7.50 -10.00
NUDO 47 156.10 10.00 59.73
NUDO 48 156.10 -10.00 59.73
NUDO 49 151.55 10.00 65.05
NUDO 50 151.55 -10.00 65.05
NUDO 51 134.00 25.00 -10.00
NUDO 52 109.00 25.00 -10.00
NUDO 53 134.00 25.00 -3.00
NUDO 54 109.00 25.00 -3.00
NUDO 55 134.00 -25.00 -10.00
NUDO 56 109.00 -25.00 -10.00
NUDO 57 134.00 -25.00 -3.00
NUDO 58 109.00 -25.00 -3.00
NUDO 59 145.45 25.00 -4.68
NUDO 60 145.45 -25.00 -4.68

VI3EE 1 43 44 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 2 41 43 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 3 1 3 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 4 1 4 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 5 4 5 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 6 2 5 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 7 2 6 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 8 7 8 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 9 4 7 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 10 5 8 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 11 7 9 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 12 8 10 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 13 9 10 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 14 1 9 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 15 2 10 2.39 8.64 4.32 4.32 0 0 0 0 0

VI3EE 16 9 11 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 17 10 12 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 18 11 13 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 19 12 14 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 20 1 13 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 21 2 14 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 22 13 14 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 23 13 15 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 24 14 16 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 25 15 16 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 26 17 18 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 27 9 19 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 28 10 20 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 29 19 20 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 30 11 19 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 31 12 20 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 32 17 19 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 33 18 20 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 34 15 22 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 35 17 22 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 36 16 21 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 37 18 21 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 38 23 24 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 39 24 26 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 40 3 26 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 41 23 25 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 42 6 25 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 43 26 28 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 44 25 30 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 45 29 30 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 46 27 28 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 47 24 27 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 48 23 29 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 49 27 29 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 50 6 30 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 51 2 30 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 52 3 28 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 53 1 28 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 54 29 32 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 55 30 32 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 56 28 31 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 57 27 31 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 58 1 31 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 59 2 32 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 60 27 32 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 61 31 32 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 62 5 32 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 63 4 31 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 64 26 34 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 65 33 34 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 66 25 33 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 67 34 35 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 68 33 36 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 69 30 37 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 70 37 38 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 71 28 38 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 72 37 39 2.39 8.64 4.32 4.32 0 0 0 0 0

VI3EE 73 38 40 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 74 33 41 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 75 41 42 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 76 3 46 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 77 45 46 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 78 34 43 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 79 6 45 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 80 41 45 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 81 43 46 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 82 1 47 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 83 47 48 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 84 2 48 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 85 2 4 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 86 47 49 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 87 48 50 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 88 3 51 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 89 51 52 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 90 13 52 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 91 51 53 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 92 52 54 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 93 3 59 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 94 14 56 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 95 56 58 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 96 55 56 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 97 55 57 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 98 6 55 2.39 8.64 4.32 4.32 0 0 0 0 0
VI3EE 99 6 60 2.39 8.64 4.32 4.32 0 0 0 0 0