УНИВЕРСАЛЬНЫЙ МЕХАНИЗМ 10

Программирование в среде UM

Рассматриваются особенности разработки пользователями собственного программного кода и его последующего подключения к моделям механических систем

2024

Оглавление

5. Программирование в среде UM

Универсальный механизм поддерживает два способа подключения внешнего программного кода к моделям механических систем: программирование в файле управления (п. 5.1. *"Программирование в файле управления"*, с. 5-3) и использование *внешних библиотек* (п. 5.3. *"Создание и использование внешних библиотек"*, с. 5-55).

Для реализации прикладных задач рекомендуется использовать метод программирования с использованием *внешних библиотек*, как более современный и лишенный некоторых недостатков программирования в *файле управления*.

Вместе с тем использование одного из способов не исключает использование другого, оба этих способа могут быть использованы одновременно в рамках подготовки одной модели.

5.1. Программирование в файле управления

Моделирование сложных технических систем предполагает овладение пользователем программирования в среде комплекса UM. Возможные языки программирования – *Pascal, Delphi, C* и *C++*. Поддерживаются следующие компиляторы: *Borland (Embarcadero) Delphi* и *C++ Builder, MS Visual C++*. Программирование в среде имеет целью расширение ограниченных стандартных возможностей программы при задании сил, описании внешних функций и для управления процессом моделирования с использованием сообщений программы моделирования. Пользователь имеет доступ к набору стандартных процедур комплекса. Программирование осуществляется с использованием *файла управления*, автоматически генерируемого при синтезе уравнений движения объекта и каждой из внешних подсистем, подключенных к объекту.

Пользователь несет ответственность за правильность написания процедур и функций, описанных им в файле управления. Отладку можно проводить с использованием программ-отладчиков, в том числе встроенных в вышеперечисленные программные системы (см. п. 5.1.8. *"Отладка файлов управления в среде Delphi"*, с. 5-49).

Пользователь может подключить к *файлу управления* любое число разработанных им программных модулей.

Альтернативой программированию в *файле управления* является использование внешних библиотек, которое подробно рассматривается в п. 5.3. *"Создание и использование внешних библиотек"*, с. 5-55.

5.1.1. Стандартные константы, типы и переменные

Приведем некоторые константы, типы и переменные, полезные при программировании в среде UM с указанием соответствующих модулей.

5.1.1.1. Модуль CtvSt.pas

- Константы

{Тип перемножения матрицы на столбец Ax (NORMAL), A'x(TRANSPON), штрих – знак транспонирования)}

```
NORMAL = 0;
TRANSPON = 1;
```

- Типы

```
real_        = double; {Основной тип представления числа с плавающей точкой}
coordin      = array[1..3] of real_; {Вектор координат}
trans_matr   = array[1..3,1..3] of real_;{Обычно – матрица направляющих косинусов 3x3}
```

{Типы динамических массивов}

```
VectReal     = array[1..MaxArReal] of real_;
VectReal0    = array[0..MaxArReal] of real_;
VectSing     = array[1..MaxArSing] of single;
VectInt      = array[1..MaxArInt] of integer;
VectByte     = array[1..MaxArByte] of byte;
VectChar     = array[1..MaxArByte] of char;
```

{Одномерные массивы}

```
VectSPtr     = ^VectSing;
VectRPtr     = ^VectReal;
VectIPtr     = ^VectInt;
VectBPtr     = ^VectByte;
VectCPtr     = ^VectChar;

MatrReal     = array[1..MaxArPtr] of VectRPtr;
MatrInt      = array[1..MaxArPtr] of VectIPtr;
MatrByte     = array[1..MaxArPtr] of VectBPtr;
```

{Двумерные массивы}

```
MatrRPtr     = ^MatrReal;
MatrIPtr     = ^MatrInt;
MatrBPtr     = ^MatrByte;
```

5.1.1.2. Модуль CtvDLL.pas

- Константы

```
{Ключи UM сообщений}
  OBJECTLOADED_MESSAGE  = -10;
  FIRSTINIT_MESSAGE     = 1;
  EQUATIONS_MESSAGE     = 10;
  INTEGRBEGIN_MESSAGE   = 30;
  INTEGREND_MESSAGE     = 40;
  STEPEND_MESSAGE       = 50;
  XVABEGIN_MESSAGE      = 60;
  XVAEND_MESSAGE        = 70;
  INTEGRFORM_MESSAGE    = 80;
  IDENT_MESSAGE         = 90;
  INITIALS_MESSAGE      = 100;
  PAUSE_MESSAGE         = 110;
  STEPSINGLE_MESSAGE    = 140;
  OBJECTCLOSE_MESSAGE   = 150;
  XVASTEP_MESSAGE       = 160;
  INTEGRPROCESS_MESSAGE = 180;
  FORCESCALC_MESSAGE    = 190;
  CALCMASSMATRIX_MESSAGE = 200

{Максимальное число внешних подсистем в объекте}
  NSubsMax    = 1000;

{Типы элементов}
  eltBody = 1;
  eltJoint = 2;
  eltSubsystem =3;
  eltBFrc = 4;
  eltLFrc = 5;
  eltCFrc = 6;
  eltAFrc = 7;
  eltSFrc = 8;
  eltGO = 11;
  eltIdentifier = 12;

{Типы сообщений}
  _mtConfirmation = 0;
  _mtInformation  = 1;
  _mtError        = 2;

{Типы контактных сил}
  _cftSum = 0;
  _cftNormal = 1;
  _cftFriction = 2;

{Ключ системы координат, в которой представлен вектор}
  BodyCoordinateSystem = 0;
  BaseCoordinateSystem = 1;
```

- Переменные

```
t: real_;- текущее значение времени в процессе моделирования движения объек-
та;
NSubSystems: integer;  - число внешних подсистем в объекте;
SubIndx: VectIPtr; - массив локальных индексов внешних подсистем;
UserVars: array[0..1001] of real_; {элементы массива доступны для отображения
в графическом окне}
```

- Типы процедур, используемых при программировании в среде UM

Модуль содержит описания типов процедур, доступных пользователю при программировании в среде.

5.1.2. Модуль DGetVars.pas

Модуль содержит процедуры и функции, полезные, например, при описании силового взаимодействия тел в системе с использованием файла управления.

5.1.3. Структура файла управления

Файл управления задачи имеет имя *cl[имя].pas (hpp)*, где *[имя]* – имя текущей задачи, автоматически создается блоком синтеза уравнений движения и размещается в каталоге задачи. Данный файл является основой программирования в среде UM.

Стандартный вид файла управления для некоторого объекта с именем *pend* (при отсутствии внешних функций в описанном объекте) следующий:

```
unit Clpend;
interface
uses  CtvSt, CtvDll;

procedure UserCalc( _x, _v, _a : VectRPtr; _isubs, _UMMessage : integer; var
WhatDo : integer ); cdecl; export;
procedure TimeFuncCalc( _t : real_; _x, _v : VectRPtr; _isubs : integer );
procedure  UserConCalc(  _x,  _v  :  VectRPtr;  _Jacobi  :  MatrRPtr;  _Error  :
Vec3RPtr; _isubs, _ic : integer; _predict : boolean; _nright : integer );
cdecl; export;
procedure  EstimationFuncCalc(  _optMode  :  integer;  _ECCount  :  integer;
_ECVectOptPtr : TVectOptPtr; var _GoOn : boolean; var _Estimation : real_;
_Msg : pchar ); cdecl; export;

implementation

uses  DGetVars, pendC, _Tpend;

{ Функция "TimeFuncCalc" используется исключительно для расчета функций вре-
мени.
Не используйте функции типа "GetPoint" внутри этой процедуры.
Недопустим расчет сил в этой процедуре. }
procedure TimeFuncCalc( _t : real_; _x, _v : VectRPtr; _isubs : integer );

var   _ : _pendVarPtr;
begin
  _ := _PzAll[SubIndx[_isubs]];
end;

procedure ForceFuncCalc( _t : real_; _x, _v : VectRPtr; _isubs : integer );
var   _ : _pendVarPtr;

begin
  _ := _PzAll[SubIndx[_isubs]];
end;

procedure  UserConCalc(  _x,  _v  :  VectRPtr;  _Jacobi  :  MatrRPtr;  _Error  :
Vec3RPtr; _isubs, _ic : integer; _predict : boolean; _nright : integer );
var   _ : _pendVarPtr;
begin
  _ := _PzAll[SubIndx[_isubs]];
end;

procedure  EstimationFuncCalc(  _optMode  :  integer;  _ECCount  :  integer;
_ECVectOptPtr : TVectOptPtr; var _GoOn : boolean; var _Estimation : real_;
_Msg : pchar );
begin
```

```
  _Estimation := 0;
  case _optMode of
    optTest : begin
      _GoOn := false;
    end;
    optPreEstimation : begin
    end;
    optEstimation : begin
    end;
  end;
end;

procedure UserCalc( _x, _v, _a : VectRPtr; _isubs, _UMMessage : integer; var
WhatDo : integer );
var   Key : integer;
begin
  Key := WhatDo;
  WhatDo := NOTHING;
  case _UMMessage of
    FORCESCALC_MESSAGE : begin
      try
        ForceFuncCalc( t, _x, _v, _isubs );
      except
        WhatDo := -1;
      end;
    end;
  end;
end;

procedure ControlPanelMessage( _x, _v, _a : VectRPtr; _isubs, _index : inte-
ger; _Value : double );
var
  _ : _brickVarPtr;
begin
  _ := _PzAll[SubIndx[_isubs]];
end;

end.
```

В интерфейсной части модуля описаны следующие процедуры:

- TimeFuncCalc
  Вычисление функций времени для нестационарных связей (шарнирные координаты, являющиеся функциями времени, заданные внешними функциями).
- ForceFuncCalc
  Вычисление сил общего типа, а также дополнительных сил, не описанных в программе ввода (см. также 5.1.6.6).
- UserCalc
  Процедура обработки сообщений программы.
- UserConCalc
  Дополнительные уравнения связей, задаваемые пользователем (не поддерживается для линеаризованных уравнений).
- EstimationFuncCalc
  Расчет целевой функции для процесса оптимизации объекта.
- ControlPanelMessage
  Дополнительная обработка сообщений, приходящих с панели управления.

Первые две процедуры вызываются из модуля *Al[имя].pas*. Поскольку структура этого файла позволяет понять последовательность расчета элементов уравнений движения, рассмотрим этот файл для задачи *pend*.

```
unit Alpend;
interface
uses   CtvSt, CtvDll;

procedure AllCalc( _x, _v : VectRPtr; var _isubs : integer; _kinemat : boole-
an; _alpha, _alpha2 : real_ ); cdecl; export;

implementation
uses  Clpend, _Tpend, pendC, pendE, DGetVars;

procedure AllCalc( _x, _v : VectRPtr; var _isubs : integer; _kinemat : boole-
an; _alpha, _alpha2 : real_ );
var   _ : _pendVarPtr;
begin
  _ := _PzAll[SubIndx[_isubs]];
  _GAlpha := _alpha;
  _GAlpha2 := _alpha2;

//Первое обращение
  if _kinemat then begin
_._ap := CommonData.APredVector;
    try
      TimeFuncCalc( t, _x, _v, _isubs );
    except
      _isubs := -2;
      exit;
    end;
//Расчет тригонометрических функций
    _._s1 := sin( _x[1] );
    _._c1 := cos( _x[1] );
    DoElement( _x, _v, 1, _isubs );
  end;

//Второе обращение
  if CommonData.ForcesCalculation then begin
    _._ap := CommonData.APredVector;
    DoElement( _x, _v, 2, _isubs );
  end;

//Третье обращение
  if CommonData.MassMatrixCalculation then begin
    _a := CommonData.MMatrixPtr;
    DoElement( _x, _v, 3, _isubs );
  end;
end;

end.
```

Процедура *AllCalc* организует расчет элементов уравнений движения. На каждой итерации шага интегрирования, на котором происходит расчет элементов уравнений движения, программа обращается к данной процедуре три раза для определения текущих значений:

- соотношений кинематики;
- обобщенных сил инерции и активных сил;
- матрицы масс.

Расчет указанных типов переменных производится в процедуре *DoElement*, поэтому вызов данной процедуры выполняется трижды.

Вызов процедуры *TimeFuncCalc* поставлен перед расчетом кинематики, поэтому в данной процедуре можно определить величины, которые определяют кинематику объекта (например, функции времени, которые используются для расчета значений координат – функций времени). По этой же причине в процедуре *TimeFuncCalc* невозможен расчет сил и вызов процедур, связанный с определением текущего кинематического состояния объекта.

Не забывайте создавать копии файла управления, если вы программируете в среде UM!

5.1.4. Полные имена элементов

Полное имя элемента содержит его собственное имя, а также имена всех подсистем (внешних и включенных) на пути от подсистемы, включающей элемент, до объекта по дереву системы.

Для того чтобы получить список полных имен всех элементов объекта выполните следующие действия:

- запустите программу ввода данных (**UM Input**);
- откройте объект;
- создайте файл элементов, используя пункт меню **Инструменты | Файл элементов**.

Файл с именем *n[ИмяОбъекта].txt* будет автоматически открыт во встроенном текстовом реакторе и записан в каталог объекта.

Пример файла элементов

```
Объект : vehicle
********************Список тел**********************
  Кузов
  Тележка1.Frame
  Тележка1.КМБлок1.Мотор
  Тележка1.КМБлок1.Ротор
  Тележка1.КМБлок1.Венец
……..
********************Список шарниров*********************
  jКузов
  Тележка1.jFrame
  Тележка1.КМБлок1.jМотор
  Тележка1.КМБлок1.jРотор
  Тележка1.КМБлок1.jВенец
……..
********************Список идентификаторов**********************
  v0
  xsh
  mc
  icx
  icy
  Тележка1.ixframe
  Тележка1.iyframe
  Тележка1.dz2
  Тележка1.fwnl2
  Тележка1.КМБлок1.v0
  Тележка1.КМБлок1.yspring
  Тележка1.КМБлок1.c1x
  Тележка1.КМБлок1.c1z
  Тележка1.КМБлок1.c1y
  Тележка1.КМБлок1.fst1
  Тележка1.КМБлок1.xmotor
  Тележка1.КМБлок1.rrotor
```

5.1.5. Индексация элементов

При описании объекта пользователь использует имена для идентификации элементов: тел, шарниров, силовых элементов и так далее, однако внутренняя идентификация элементов основана на их индексации. С помощью индексов при программировании в среде UM пользователь имеет доступ к значениям переменных, характеризующих элемент (например, к скорости какой-либо точки тела).

Индексация элементов начинается с 1.

В UM используется двойная индексация каждого элемента (за исключением внешних подсистем):

1. индекс внешней подсистемы isubs (1..NSubSystems), которой принадлежит элемент; при отсутствии внешних подсистем этот индекс всегда равен 1.
2. индекс элемента в пределах соответствующей внешней подсистемы.

Индексация предусмотрена для следующих типов элементов:

- Подсистемы;
- Графические объекты;
- Координаты;
- Тела;
- Шарниры;
- Биполярные силы;
- Общие силы;
- Линейные силы;
- Контактные силы;
- Специальные силы;
- Идентификаторы.

Существует два способа, позволяющие определить индексы элемента.

3. В программе ввода данных (**UM Input**) в списке элементов модели выбрать пункт Индексы, информация отобразится в инспекторе данных.

Рассмотрим пример, представленный на рис. 5.1. Объект содержит две внешних подсистемы. Элементы каждого типа представлены в виде двухуровневого дерева: первый уровень – тип элемента, во втором уровне – непосредственно элементы с указанием значений индексов (в пределах каждого типа). Если элемент принадлежит подсистеме, то его имя начинается с имени подсистемы и точки. Например, графический объект с именем Ротор, принадлежащий подсистеме Электромотор, имеет имя «Электромотор.Ротор» и индекс 4.

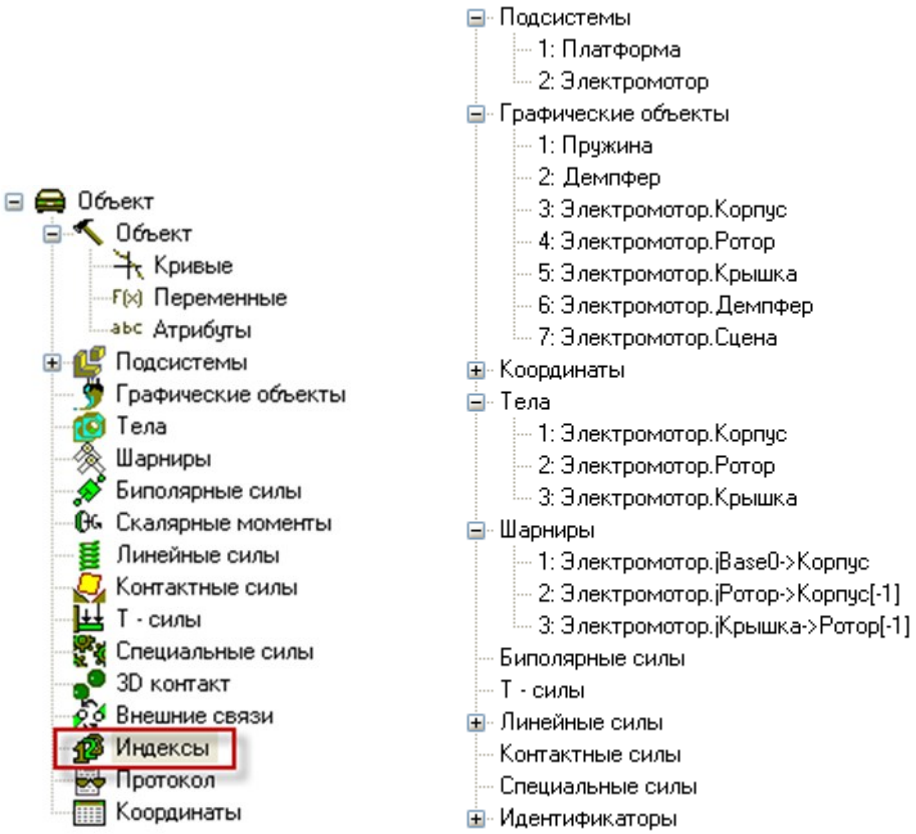


Рис. 5.1. Индексация элементов

Данный способ определения индексов имеет существенный недостаток: индексы могут меняться при модификации объекта: добавления и удаления элементов и подсистем. Потому рекомендуется использовать второй способ.

4. Программный способ определения индексов элементов по их полным именам.

Для определения индексов элементов используется функция

function **GetElementIndexByName**(elType : integer; const Name : string;
var index, isubs : integer): integer; cdecl;[1]

Входные параметры:

elType – тип элемента из списка

Тип	Комментарий
EltBody	Тело
EltJoint	Шарнир
EltSubsystem	Подсистема
EltBFrc	Биполярная сила
EltLFrc	Линейный силовой элемент
EltCFrc	Контактный силовой элемент
EltAFrc	Сила общего типа
EltSFrc	Специальная сила

[1] При использовании C для программирования в среде декларации типов смотрите в файле CtvDll.hpp

EltGO	Графический объект
EltIdentifier	Идентификатор

Name – полное имя элемента (см. п. 5.1.4. *"Полные имена элементов"*, с. 5-10)

Возвращает:

index : индекс элемента;

isubs : индекс внешней подсистемы;

значение функции: 0 – успешное исполнение, -1 – ошибка в имени элемента или типе.

Указания

Для определения индексов рекомендуется использовать процедуру *UserCalc* файла управления и сообщение **FIRSTINIT_MESSAGE**, которое выполняется однократно непосредственно после загрузки объекта в модуль моделирования.

Для запоминания индексов обязательно используйте глобальные или статические переменные.

Пример

В примере определяются индексы двух линейных силовых элементов и запоминаются в массивах, описанных как глобальные. При некорректном выполнении процедур будет подано соответствующее сообщение.

```
procedure UserCalc;
var Key : integer;
begin
  Key :=WhatDo;
  WhatDo:=NOTHING;
  case UMMessage of
    FORCESCALC_MESSAGE : begin
      try
        ForceFuncCalc( t, _x, _v, _isubs );
      except
        WhatDo := -1;
      end;
    end;
  FIRSTINIT_MESSAGE : begin
      If GetElementIndexByName(eltLFrc,'Тележка1.КМБлок1.ПружинаЛ',
            LFrc1Indices[1,1], LFrc1Subs[1,1])=-1 then
        UMMessage('Ошибка при определении индекса элемента '+
                  'Тележка1.КМБлок1.ПружинаЛ ', _mtError);
      If GetElementIndexByName(eltLFrc,'Тележка1.КМБлок1.ПружинаП',
            LFrc1Indices[2,1], LFrc1Subs[2,1])=-1 then
        UMMessage('Ошибка при определении индекса элемента '+
                   'Тележка1.КМБлок1.ПружинаП ', _mtError);
    end;
  end;
end;
```

5.1.6. Функции и процедуры

Доступ к процедурам и функциям осуществляется после подключения модуля Dget-Vars. В приведенных процедурах *isubs* – номер внешней подсистемы (единица – для объекта, не содержащего внешние подсистемы).

5.1.6.1. Число элементов

Для определения числа элементов в объекте или внешней подсистеме используется функция

- function GetNElements(elType,isubs : integer) : integer;

Входные параметры:

elType – тип элемента из списка

Тип	Комментарий
EltBody	Тело
EltJoint	Шарнир
EltSubsystem	Подсистема
EltBFrc	Биполярная сила
EltLFrc	Линейный силовой элемент
EltCFrc	Контактный силовой элемент
EltAFrc	Сила общего типа
EltSFrc	Специальная сила
EltGO	Графический объект
EltIdentifier	Идентификатор

isubs : индекс внешней подсистемы.

Функция возвращает число элементов указанного типа при успешном завершении, в противном случае возвращает –1.

5.1.6.2. Значения координат объекта

Для определения текущих значений координат и их производных по времени используется функция

function **GetCoord** (nr, isubs, deriv: integer) : real_;

Возвращает текущее значение n-ой производной от координаты с индексом *nr*.

Входные параметры:

nr – индекс координаты,

isubs – индекс подсистемы,

deriv – степень производной.

Степень производной задается параметром *deriv*. Вызов функции с аргументом *deriv=0* вернет непосредственно значение координаты, с аргументом *deriv=1* – ее первой производной и т.д.

5.1.6.3. Кинематические характеристики тел

- procedure **GetPoint** (body, isubs : integer; ro : coordin; var r : coordin);
  Определение координат в базовой системе координат (СК0) точки тела.
  Входные параметры: *body* – номер тела, *ro* – координаты точки тела в связанной СК.
  Выходные параметры: *r* – координаты точки тела в СК0.
- procedure **GetVel** (body, isubs : integer; ro : coordin; var v : coordin);
  Определение скорости точки тела относительно СК0.
  Входные параметры: *body* – номер тела, *ro* – координаты точки тела в связанной СК.
  Выходные параметры: *v* – вектор скорости точки тела в СК0.
- procedure **GetAi0**(body, isubs : integer; var ai0 : Trans_Matr);
  Возвращает матрицу направляющих косинусов ai0 тела body относительно СК0.

Замечание 1. Матрица Ai0 переводит вектор из СК0 в СК, связанную с телом, имеющим номер i=body, то есть r(i)= Ai0 r(0). Обратный переход осуществляется транспонированной матрицей.

Замечание 2. Если уравнения движения объекта линеаризованы и *при нулевых значениях координат оси, связанные с телом, параллельны осям СК0*, то элементы матрицы определяют значения углов поворота тела относительно СК0.
Пусть α, β, γ – углы поворота тела относительно осей *x, y, z* соответственно.
Тогда $\underline{A}_{i0} = \begin{pmatrix} 1 & \gamma & -\beta \\ -\gamma & 1 & \alpha \\ \beta & -\alpha & 1 \end{pmatrix}$,
то есть по элементам матрицы можно получить значения углов поворотов.

- procedure **GetVelAng** (body, isubs : integer; var v : coordin);
  Определение вектора *v* угловой скорости тела *body* относительно СК0. Вектор представлен в СК0.

Группа процедур определяет относительное положение и движение пары тел. Следующие входные параметры процедур являются общими:

bd1, bd2 – индексы тел, определяется движение тела bd2 относительно тела bd1;

isubs1, isubs2 – индексы подсистем каждого тела;

bdref, isubsref – индексы, задающие тело, в проекциях на оси СК которого задаются векторы.

- procedure **GetRelCoord** (bd1, isubs1, bd2, isubs2, bdref, isubsref : integer; const ro1, ro2 : coordin; var r12 : coordin);

Процедура возвращает вектор r12, соединяющий две точки тел; координаты этих точек задаются в СК каждого тела векторами ro1, ro2.

- procedure **GetRelVeloc** (bd1, isubs1, bd2, isubs2, bdref, isubsref : integer; const ro2 : coordin; var v12 : coordin);

Процедура возвращает вектор скорости v12 точки ro2 второго тела относительно первого тела. Вектор ro2 задается в СК второго тела.

- procedure **GetRelAcc** (bd1, isubs1, bd2, isubs2, bdref, isubsref : integer; const ro2 : coordin; var a12 : coordin);

Процедура возвращает вектор ускорения a12 точки ro2 второго тела относительно первого тела. Вектор ro2 задается в СК второго тела. *Процедуру следует использовать только при обработке сообщений* **STEPSINGLE_MESSAGE** и **STEPEND_MESSAGE** *в процедуре UserCals*.

- procedure **GetRelAng** (bd1, isubs1, bd2, isubs2, bdref, isubsref : integer; var ang12 : coordin; var angle : real_);

Процедура возвращает вектор поворота ang12 и величину угла поворота второго тела относительно первого тела.

- procedure **GetRelVelAng** (bd1, isubs1, bd2, isubs2, bdref, isubsref : integer; var om12 : coordin);

Процедура возвращает вектор угловой скорости om12 второго тела относительно первого тела.

- procedure **GetRelAccAng** (bd1, isubs1, bd2, isubs2, bdref, isubsref : integer; var e12 : coordin);

Процедура возвращает вектор углового ускорения e12 второго тела относительно первого тела. *Процедуру следует использовать только при обработке сообщений* **STEPSINGLE_MESSAGE** и **STEPEND_MESSAGE** в *процедуре UserCals*.

- procedure **GetRelVelPoints** (bd1, isubs1, bd2, isubs2, bdref, isubsref : integer; const ro1,ro2 : coordin; var v12 : coordin; var v : real_);

Процедура возвращает величину v скорости изменения расстояния между парой точек тел, заданных векторами ro1, ro2 в СК соответствующих тел, а также вектор скорости v12, направленной по вектору, соединяющему точки, причем $|v_{12}| = |v|$. Таким образом, v равно производной по времени от длины вектора r12.

- procedure **GetRelAccPoints** (bd1, isubs1, bd2, isubs2, bdref, isubsref : integer; const ro1,ro2 : coordin; var a12 : coordin; var a : real_);

Процедура возвращает величину a ускорения изменения расстояния между парой точек тел, заданных векторами ro1, ro2 в СК соответствующих тел, а также вектор ускорения a12, направленной по вектору, соединяющему точки, причем $|a_{12}| = |a|$. Таким образом, *a* равно второй производной по времени от длины вектора r12. *Процедуру следует использовать только при обработке сообщений* **STEPSINGLE_MESSAGE** и **STEPEND_MESSAGE** *в процедуре UserCals*.

5.1.6.4. Операции с векторами и 3x3-матрицами

Для выполнения матричных операций с векторами (тип *coordin*) и 3x3-матрицами (тип trans_*matr*) можно использовать набор внутренних процедур и функций UM.

- procedure **Mult_vec_val**(var a : coordin; r : real_; var b : coordin);
  Произведение вектора на скаляр, *b=ca*

- function **ScalarMult**(var a, b : coordin) : real_;
  Функция возвращает скалярное произведение векторов.

- function **Vect_Len**(var a : coordin) : real_;
  Функция возвращает евклидову норму вектора.

- function **Vect_Mult**(a, b : Coordin; i : integer) : real_;
  Функция возвращает i-ю компоненту векторного произведения **a×b**

- procedure **Mult_Vect**(a,b : coordin; var c : coordin);
  Процедура возвращает векторное произведение **c=a×b**

- procedure **Turning**(angle : real_; e : coordin; var a10 : trans_matr);
  Процедура реализует формулу конечного поворота. Пусть СК1 является результатом поворота СК0 на угол *angle* вокруг оси, задаваемой вектором e. Процедура возвращает матрицу направляющих косинусов A_{10}. Если передается нулевой вектор e, то возвращается единичная матрица.

- procedure **MkVectAng**(var angle : real_; var rvect : coordin; var a10 : trans_matr);
  Процедура выполняет действие, обратное процедуре **Turning**. Пусть ориентация СК1 есть результат поворота СК0 на угол *angle* вокруг вектора **e**. Процедура возвращает значение угла и вектора по заданной матрице направляющих косинусов *a10*.

- procedure **GetRotAngles**(const A: trans_matr; i, j, k: integer; var ai, aj, ak: real_);
  Процедура возвращает три угла ориентации ai, aj, ak по заданной матрице направляющих косинусов A и последовательности поворотов i,j,k.

- procedure **Mult_m_v_3x1**(type_ :byte; var a: trans_matr; b: coordin; var c: coordin);
  Процедура вычисляет произведение 3x3-матрицы на столбец, *c=ab*, если *type_=NORMAL* и *c=a'b*, если *type_=TRANSPON* (штрих – знак транспонирования). Используется главным образом для перевода вектора из одной системы координат в другую.
- procedure **Mult_m_m**(a,b : trans_matr; var c : trans_matr);
  Вычисляет произведение 3x3- матриц, *c=ab*.

- procedure **Mult_mT_m**(a,b : trans_matr; var c : trans_matr);

Вычисляет произведение 3x3- матриц, *c=a'b*, первая матрица в произведении – транспонированная.

- procedure **Mult_m_mT** (a,b : trans_matr; var c : trans_matr);

Вычисляет произведение 3x3- матриц, *c=ab'*, вторая матрица в произведении – транспонированная.

- procedure **Mult_mT_mT** (a,b : trans_matr; var c : trans_matr);

Вычисляет произведение 3x3- матриц, *c=a'b'*, первая и вторая матрицы в произведении – транспонированные.

5.1.6.5. Решение систем алгебраических уравнений

- procedure **GaussCalc**(a,b : MatrRPtr; na,nb,code : integer; eps : real_);

Реализация метода исключения Гаусса решения системы линейных матричных алгебраических уравнений $AX=B$ с невырожденной матрицей A с поиском ведущего элемента по столбцу. Список параметров:

a – входной параметр, матрица A; матрица не сохраняется;

b – вход – матрица правых частей B, выход – матрица решения X.

na – размер матрицы A na× na;

nb – размер матрицы B na× nb.

Тип *MatrRPtr* соответствует двухиндексному динамическому массиву;

code – признак вырожденности матрицы системы, code=1 – вырождение не обнаружено, code=0 – обнаружено вырождение, решение не найдено;

eps – положительное число, определяющее вырождение. Матрица считается вырожденной, если ведущий элемент в столбце в процессе метода исключения оказался по абсолютной величине меньше eps. Рекомендуемое значение eps=1.0e-12.

- procedure **GaussSingleCalc**(a,b: pointer; na : integer; var code : integer; eps : real_);

Реализация метода исключения Гаусса решения системы линейных матричных алгебраических уравнений $Ax=b$ с невырожденной матрицей A с поиском ведущего элемента по столбцу. Список параметров:

a – входной параметр, матрица A; матрица *не сохраняется*;

b – вход – столбец правых частей b, выход – столбец решений x.

na – размер матрицы A na× na;

code – признак вырожденности матрицы системы, code=1 – вырождение не обнаружено, code=0 – обнаружено вырождение, решение не найдено;

eps – положительное число, определяющее вырождение. Матрица считается вырожденной, если ведущий элемент в столбце в процессе метода исключения оказался по абсолютной величине меньше eps. Рекомендуемое значение eps=1.0e-12.

5.1.6.6. Добавление сил и моментов

Пользователь может добавить дополнительные активные силы, действующие на тела объекта, *не описанные в модуле ввода*. Значения сил и моментов должны быть вычислены в процедуре *ForceFuncCalc* или при обработке сообщения **FORCESCALC_MESSAGE** в процедуре *UserCalc* и добавлены к силам, действующим на объект.

В процедурах, описанных ниже, параметр CoordSystem определяет систему координат, в которой задан вектор силы и/или момента и принимает одно из двух значений:

BaseCoordinateSystem – базовая система координат;

BodyCoordinateSystem – система координат тела (или *второго* тела в случае процедуры **AddForceToBodyAtPoint**).

Силы должны быть приведены к началу отсчета СК, связанной с телом (кроме процедуры **AddForceToBodyAtPoint**).

При успешном выполнении функции возвращают 0, в противном случае –1.

- function **AddForceToBody**(ibody,isubs : integer; Force : coordin;
  CoordSystem : integer) : integer;

Функция добавляет вектор силы *Force* к телу *ibody* подсистемы *isubs* в начало отсчета СК, связанной с телом.

- function **AddMomentToBody**(ibody,isubs : integer; Moment : coordin;

CoordSystem : integer) : integer;

Функция добавляет вектор момента *Moment* к телу *ibody* подсистемы *isubs*.

- function **AddForceToBodyAtPoint**(ibody1,isubs1, ibody2, isubs2: integer;

Point, Force, Moment : coordin; CoordSystem : integer) : integer;

Функция добавляет силы взаимодействия к паре тел. Функции передаются сила и момент, действующие на *второе тело* (ibody2, isubs2), приведенные к точке Point (координаты точки задаются в СК второго тела). Проекции силы и момента задаются либо в СК0 (CoordSystem принимает значение BaseCoordinateSystem), либо в СК второго тела (CoordSystem принимает значение BodyCoordinateSystem).

Пример

К телу 1 добавляется внешняя активная сила (0, 10000, 0), имеющая постоянное направление в СК0 и приложенная к точке с координатами (0, 0, 0.5) в СК тела 1. В данном случае первое тело имеет индекс 0 (базовое тело), а процедура в файле управления выглядит следующим образом:

```
procedure ForceFuncCalc( _t : real_; _x, _v : VectRPtr; _isubs : integer );

var Point, Force, Moment : coordin;
begin
  Point[1]:=0;   Point[2]:=0; Point[3]:=0.5;
  Force[1]:=0;   Force[2]:=10000;   Force[3]:=0;
  Moment[1]:=0;   Moment[2]:=0;   Moment[3]:=0;
```

```
  AddForceToBodyAtPoint(0,1,  1,  1,Point,  Force,  Moment,  BaseCoordinateSys-
tem);
end;
```

Пример использования процедуры *AddForceToBody* рассмотрен в п. 5.1.7.2. *"Программирование шарнирных и биполярных сил"*, с. 5-39.

5.1.6.7. Изменение значений идентификаторов

UM использует параметризацию значительной части данных, описывающих объект, и сил, действующих на него. Поэтому изменение программным путем значений идентификаторов является мощным инструментом при моделировании сложных объектов.

Чаще всего пользователь изменяет значения идентификаторов перед началом процесса интегрирования с использованием соответствующего диалогового окна, содержащего список идентификаторов. Однако возможно изменение идентификаторов, определяющих параметры многих типов элементов путем программирования в среде, в том числе и в процессе моделирования.

С другой стороны, изменение некоторых параметров в процессе исследования динамики объекта может привести к ошибочным результатам, поскольку входит в противоречия с законами механики. К таким параметрам относятся:

- инерционные параметры: массы, моменты инерции тел, присоединенные массы;
- координаты шарнирных точек;
- параметры графических образов (ГО), при условии, что инерционные параметры тел вычисляются автоматически по этим ГО.

При изменении идентификаторов, параметризующих указанные величины, следует начинать заново процесс интегрирования.

5.1.6.7.1. Структуры идентификаторов

Идентификаторы, введенные пользователем при параметризации элементов объекта, в рамках уравнений движения объекта образуют структуры, причем различные для разных внешних подсистем. Описание типа структуры идентификаторов размещается при синтезе уравнений в файле _T[ИмяОбъекта].

Структура строится в строгом соответствии с деревом включенных подсистем, причем в качестве заголовка подструктуры, содержащей идентификаторы включенной подсистемы, используется *идентифицирующее имя подсистемы*.

Приведем пример. Объект Vehicle содержит две включенные подсистемы с идентифицирующими именами WMBlock1 и WMBlock2. В свою очередь, каждая из этих подсистем содержит по включенной подсистеме с идентифицирующим именем Wset. В результате идентификаторы объекта будут представлены структурой, описание типа которой _vehicleVars представлено ниже:

```
_vehicleVars = record
   v0 : real_;
   xsh : real_;
   WMBlock1 : record
     xmotor : real_;
     rrotor : real_;
     ……
     wset : record
       mw : real_;
       iwx : real_;
       ……
     end;
   end;
   WMBlock2 : record
     v0 : real_;
     yspring : real_;
     wset : record
       mw : real_;
       iwx : real_;
       ……
     end;
     ……
   end;
   ……
end;
```

Пользователь имеет доступ к идентификаторам объекта через переменные

_PzAll, _PzAll[ИмяОбъекта], определенные в модуле [ИмяОбъекта]C.pas и являющиеся указателями на один и тот же массив структур идентификаторов. Длина массива структур идентификаторов объекта всегда равна 1, в то время как для внешних подсистем эта длина равна числу внешних подсистем, порожденных одним объектом, и присоединенных к моделируемому объекту.

Например, объект Train содержит 11 внешних подсистем, причем одна порождена объектом Locomotive, а десять остальных – объектом Car. В этом случае длина массива идентификаторов для подсистемы, порожденной Locomotive, будет 1, тип структуры будет содержаться в файле …\Locomotive_TLocomotive.pas, переменные _PzAll, _PzAllLocomotive – в файле …\Locomotive\LocomotiveC.pas. Идентификаторы всех подсистем, порожденных

Car, будут доступны в массивах _PzAll и _PzAllCar с размером 10, определенных в файле …\Car\CarC.pas, а соответствующий тип структуры – в файле …\Car_TCar.pas.

Для доступа к идентификаторам используется следующий синтаксис:

_PzAll[ИмяОбъекта].[Элемент структуры идентификаторов]

или

_PzAll[SubIndx[isubs]].[Элемент структуры идентификаторов]

Примеры:

_PzAllVehicle[1].xsh

_PzAll[1].WMBlock1.xmotor

_PzAll[1].WMBlock2.Wset.mw

Если объект не содержит ни включенных, ни внешних подсистем, доступ к идентификаторам проще, поскольку, во-первых, массив идентификаторов всегда содержит лишь один элемент, а во-вторых, структура идентификаторов имеет только один уровень. Примеры:

_PzAll[1].mass

_PzAll[1].ix

_PzAll[1].something

_PzAllMyObject[1].some_identifier

5.1.6.7.2. Процедуры изменения идентификаторов

Значения идентификаторов могут изменяться двумя способами в зависимости от типа элемента, параметризация которого выполнена с использованием идентификаторов.

Первый способ состоит в изменении значений идентификаторов, входящих в структуру идентификаторов объекта (п. 5.1.6.7.1. *"Структуры идентификаторов"*, с. 5-24) *без изменения их значений в ядре программы моделирования*. Такой способ используется в случае, если расчет элемента полностью сосредоточен в DLL объекта или внешней подсистемы. К таким элементам относятся

- геометрические и кинематические характеристики за некоторым исключением;

В этом случае для изменения значения идентификатора достаточно присвоить ему новое значение в соответствующей процедуре файла управления, например,

_PzAllMyObject[1].some_identifier:=0.1

Второй способ изменяет значение идентификатора не только в пределах DLL, но и передает его новое значение в ядро программы моделирования и используется в случаях, когда расчет элемента или его воздействия на объект производится непосредственно программой моделирования, или программе необходимо “знать” новое значение идентификатора. К элементам такого типа относятся:

- инерционные параметры;
- идентификаторы, параметризующие графические объекты;
- идентификаторы, параметризующие биполярные силовые элементы, контактные силы, зубчатые передачи.
- силы тяжести (точнее, направление силы тяжести);
- обобщенные линейные силовые элементы;
- дополнительные T-силы, рассчитываемые пользователем (п. 5.1.6.6. *"Добавление сил и моментов"*, с. 5-21).

В данном случае, кроме изменения значения идентификатора в структуре, следует переслать его новое значение в программу моделирования с использованием процедуры

procedure **SetIdentifierValue** (index, isubs : integer; value : real_)

где index, isubs – индекс идентификатора и соответствующей подсистемы (п. 5.1.5. *"Индексация элементов"*, с. 5-11), real_ – новое значение идентификатора.

После это при необходимости следует вызвать процедуру обновления элемента (кроме случаев T-сил и направления силы тяжести)

function **RefreshElement**(elType, index, isubs : integer) : integer

где elType – тип элемента, index, isubs – его индекс и индекс соответствующей подсистемы. Вызов последней процедуры необходим для обновления графического образа и биполярного силового элемента.

Замечание. Напомним, что речь в данном разделе идет об изменении идентификаторов в процессе численного моделирования движения и относится к элементам программирования в среде. Для изменения идентификаторов перед началом

моделирования достаточно внести их новые значения на соответствующей закладке, содержащей значения идентификаторов.

5.1.6.7.3. Изменение идентификаторов, параметризующих Т-силы

Т-силы типа предоставляют пользователю одну из возможностей расчета активных сил с использованием программирования в среде UM (см. пп. 2.4.8, 3.3.10.4). В этом случае как координаты точки приложения сил, так и проекции сил и моментов задаются с использованием идентификаторов. Соответствующие идентификаторы могут быть изменены в процессе интегрирования. Для этого следует использовать процедуру *ForceFuncCalc*, вызов которой происходит на сообщение FORCESCALC_MESSAGE в процедуре UserCalc файла управления. Изменение значений идентификаторов производится с помощью процедуры **SetIdentifierValue**. Индекс идентификатора доложен быть предварительно определен и сохранен в глобальной переменной.

Пример:

```
var index_frc1x  : integer;

procedure ForceFuncCalc( _t: real_; _x, _v: VectRPtr; _isubs: integer);
var _ : _balanzVarPtr;
begin
  _ := _PzAll[SubIndx[_isubs]];
  SetIdentifierValue(index_frc1x, 1, sin(2*_t));
  //Гармоническое возмущение вдоль оси X
end;

procedure UserCalc;
var i : integer;
begin
  WhatDo:=NOTHING;
  case CurrEvent of
    FIRSTINIT_MESSAGE : begin
        GetElementIndexByName(eltIdentifier, 'fc1x', index_frc1x, i);
   //Запоминаем значение индекса идентификатора в глобальной переменной
   end;
    FORCESCALC_MESSAGE : begin
      try
        ForceFuncCalc( t, x, v, isubs );
      except
        WhatDo := -1;
      end;
    end;
  end;
end;
```

Замечание. Использование сил общего типа является устаревшим способом программирования сил в UM. В современной версии часто удобнее пользоваться прямым добавлением рассчитанных сил (см. п. 5.1.6.6. *"Добавление сил и моментов"*, с. 5-21).

5.1.6.7.4. Изменение идентификаторов, параметризующих граф. объекты

В процессе моделирования движения объекта иногда является необходимым изменять графический объект, чтобы имитировать его изменение с течением времени. Как правило, это касается ГО, являющегося образом сцены. Весьма универсальным средством создания таких ГО является использование графических элементов типа *Z-поверхность*, пример применения которых будет рассмотрен позже. Другая возможность связана с параметризацией ГЭ и с изменением численных значений параметров путем обработки сообщений **STEPEND_MESSAGE** в процессе интегрирования уравнений движения и **XVASTEP_MESSAGE** в процессе XVA-анализа.

Для изменения значений параметров, параметризующих ГО используется процедура SetIdentifierValue, затем следует вызвать функцию RefreshElement для обновления графического образа.

SetIdentifierValue(indexIdent, isubsGO, value);

RefreshElement(eltGO, indexGO, isubsGO);

Пример. Следует изменить положение ГО, назначенного в качестве образа сцены, относительно базы в зависимости от времени. Имя ГО – “Сцена”. Положение (например, координата X одного из ГЭ относительно базы) задано идентификатором xscene. Объект не содержит ни внешних, ни включенных подсистем.

```
Var IndexGO: integer;
    IndexIdent : integer;
    OldIdentValue : real_;

procedure UserCalc;
var Key : integer;
    i : integer;
begin
  Key :=WhatDo;
  WhatDo:=NOTHING;
  case UMMessage of
    FORCESCALC_MESSAGE : begin
      try
        ForceFuncCalc( t, _x, _v, _isubs );
      except
        WhatDo := -1;
      end;
    end;
  FIRSTINIT_MESSAGE : begin
      OldIdentValue:=PzAll[1].xscene; //Сохраняем старое значение идентифика-
тора
      If GetElementIndexByName(eltIdentifier,'xscene',IndexIdent, i])=-1 then
         UMMessage('Ошибка при определении индекса элемента xscene ',
_mtError);
      If GetElementIndexByName(eltGO,'Сцена',IndexGO, i])=-1 then
        UMMessage('Ошибка при определении индекса элемента Сцена', _mtError);
    end;
  STEPEND_MESSAGE : begin
      PzAll[1].xscene:=0.2*sin(2*t);
      SetIdentifierValue(IndexIdent, 1, PzAll[1].xscene);
      RefreshElement(eltGO, IndexGO, 1)
    end;
  INTEGREND_MESSAGE : begin //Возвращаем прежнее значение
      PzAll[1].xscene:= OldIdentValue;
      SetIdentifierValue(IndexIdent, 1, PzAll[1].xscene);
```

```
      RefreshElement(eltGO, IndexGO, 1)
    end;
  end;
```

В результате элемент ГО, положение которого задается идентификатором, будет совершать гармонические колебания вдоль оси X.

5.1.6.8. Анимация векторов пользователя

Пользователь имеет возможность динамически сформировать список векторов, величины которых рассчитываются в файле управления. Основная цель данного списка – предоставить возможность отображения в анимационном окне нестандартных векторов, которые не могут быть получены автоматически с помощью мастера переменных.

Для добавления вектора в список используется функция

function **AddUserVector**(var Name : string; vtype : integer) : integer;

Функция создает переменную типа вектор, добавляет к списку и возвращает индекс вектора, по которому можно назначить вектору численное значение в процессе моделирования. Входными параметрами являются *Name* – имя вектора и *vtype* – тип вектора. Тип вектора формируется из подмножества множества типов, описанного в модуле *CtvDll*:

(vtNoType, vtVelocity,vtAcceleration,vtRotation,vtAngularVelocity,
vtAngularAcceleration,vtForce,vtMoment).

При вызове функции следует преобразовать соответствующий элемент множества к типу, совместимому с типом integer, например, *ord(vtVelocity)*, это будет означать, что вектор имеет размерность скорости.

Вызовы данной функции следует размещать в процедуре *UserCalc* при обработке сообщения FIRSTINIT_MESSAGE.

Численное значение вектора пересылается с помощью функции

function **SetVectorValue**(index : integer; Value, Point : coordin) : integer;

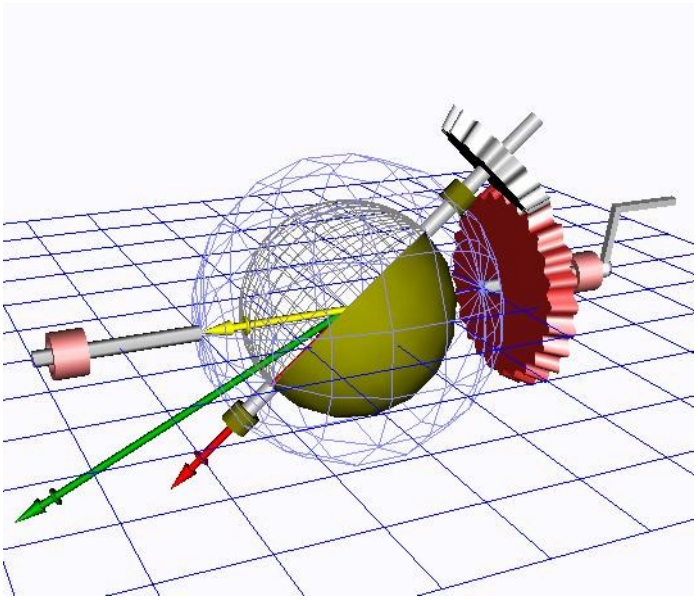
Здесь *index* – индекс вектора, *Value, Point* – его значение и точка приложения. Два последних вектора должны рассчитываться в СК0. При успешном выполнении функция возвращает нулевое значение.

Вызовы данной функции следует размещать в процедуре *UserCalc* при обработке сообщений STEPEND_MESSAGE и XVASTEP_MESSAGE.

При моделировании движения объекта доступ к списку векторов пользователя осуществляется с помощью мастера переменных (закладка **User | Векторы**).

Пример.

Рассмотрим модель шаровой дробилки (объект *Crusher* в каталоге *Tutorial*). Данная модель иллюстрирует сложение вращений тела, а программирование векторов используется для анимации векторов абсолютной, относительной и переносной угловых скоростей центрального шара дробилки (заметим, что в данном примере все перечисленные векторы угловых скоростей могли бы быть сформированы с помощью мастера переменных, то есть пример носит иллюстративный характер)



Рассмотрим соответствующий файл управления.

Вводятся следующие глобальные переменные:

var avIndex : integer; //индекс вектора абсолютной угловой скорости
rvIndex : integer; //индекс вектора относительной угловой скорости
evIndex : integer; //индекс вектора переносной угловой скорости
BodyIndex : integer; //индекс тела – корпуса дробилки
CrusherIndex : integer; //индекс тела – центрального шара дробилки

Процедура UserCalc имеет следующий вид:

```
procedure UserCalc( _x, _v, _a : VectRPtr; _isubs, _UMMessage : integer; var
WhatDo : integer );

const ro0 : coordin = (0,0,0);
var
  Key : integer;
  s : string;
  oma : coordin; // Абсолютная угловая скорость
  ome : coordin; // Переносная угловая скорость
  omr : coordin; // Относительная угловая скорость
  i : integer;

begin
  Key := WhatDo;
  WhatDo := NOTHING;
  case _UMMessage of
       FORCESCALC_MESSAGE : begin
         try
           ForceFuncCalc( t, _x, _v, _isubs );
         except
           WhatDo := -1;
         end;
       end;
    FIRSTINIT_MESSAGE : begin
//Создаются векторы, запоминаются индексы векторов и тел
        s:='Absolute ang. veloc.';
        avIndex:=AddUserVector(s,Ord(vtAngularVelocity));
        s:='Realtive ang. veloc.';
        rvIndex:=AddUserVector(s,Ord(vtAngularVelocity));
```

```
        s:='Transient ang. veloc.';
        evIndex:=AddUserVector(s,Ord(vtAngularVelocity));
        GetElementIndexByName(eltBody,'Body',BodyIndex,key);
        GetElementIndexByName(eltBody,'Crusher',CrusherIndex,key);
      end;
    XVASTEP_MESSAGE,
    STEPEND_MESSAGE : begin
//Рассчитываются векторы
        GetVelAng(CrusherIndex,1,oma);
        GetVelAng(BodyIndex,1,ome);
        for i:=1 to 3 do omr[i]:=oma[i]-ome[i];
//Пересылаются значения векторов и нулевые координаты точек приложения
        SetVectorValue(avIndex,oma,ro0);
        SetVectorValue(rvIndex,omr,ro0);
        SetVectorValue(evIndex,ome,ro0);
      end;
  end;
end;
```

5.1.7. Программирование внешних функций

Внешние функции используются при описании:

- шарнирных координат, являющихся функциями времени (вращательный, поступательный шарнир и шарнир обобщенного тапа, п. 3.3.9);
- шарнирных сил и моментов в случае шарнира обобщенного типа (п. 3.3.9.4);
- биполярных силовых элементов (п. 3.3.10.1);
- контактной поверхности для контактных силовых элементов типа Z-сфера, Z-окружность;
- графического элемента типа Z-поверхность.

Во всех перечисленных случаях пользователь задает только идентификатор функции, а ее расчет должен запрограммировать в файле управления.

При синтезе уравнений движения автоматически формируются заголовки функций в файле управления и их вызовы для соответствующих точек уравнений движения, причем функциям “по умолчанию” устанавливаются нулевые значения. Пользователю следует заполнить “тело” функции нужными вычислениями.

Если пользователь модифицирует объект, уравнения движения которого были синтезированы ранее, и в результате модификации добавил, удалил или переименовал внешнюю функцию, то необходимо скорректировать прежний файл управления. Для этого следует воспользоваться вновь синтезированным файлом управления, который будет записан с расширением new, если при синтезе уравнений не был установлен флажок **Переписать файл управления.** Если при синтезе уравнений установлен флажок перезаписи файла управления (что, вообще говоря, не рекомендуется), то старый вариант файла будет скопирован с расширением old, и следует перенести ранее написанные ранее процедуры в новый вариант файла.

5.1.7.1. Программирование шарнирных координат – функций времени

Рассмотрим модель кривошипно-ползунного механизма (рис. 5.2, объект CrankRod в каталоге Tutorial). Для исследования кинематики механизма зададим движение кривошипа в зависимости от времени, используя внешнюю функцию с идентификатором *phi* при задании угла поворота.

При синтезе уравнений движения в файл управления будет добавлена функция с введенным именем

```
procedure phi( _isubs : integer; _t : real_; var _Value, _dValue, _ddValue :
real_ );
var  _ : _crankrodVarPtr;
begin
  _ := _PzAll[SubIndx[_isubs]];
  _Value := 0;
  _dValue := 0;
  _ddValue := 0;
end;
```

в которой пользователь должен вычислить значение угла (переменная _Value) и первых двух производных по времени _dValue, _ddValue.

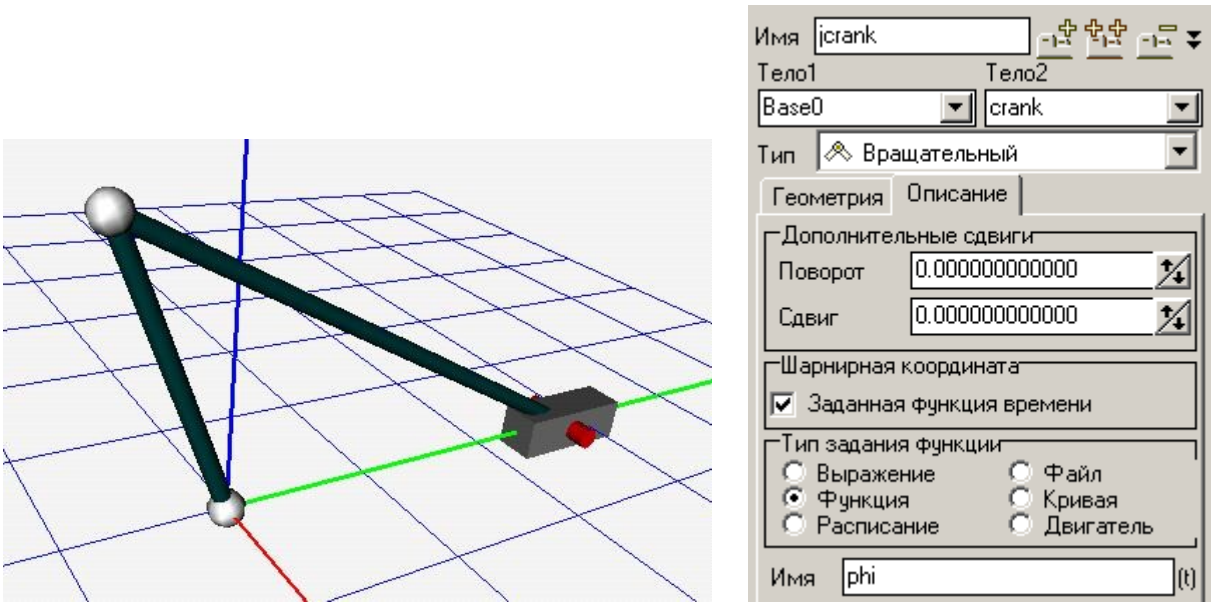


Рис. 5.2. Координата как функция времени

Кроме того, в файл AlCrankRod непосредственно после вызова функции TimeFuncCalc вставлен вызов функции phi:

```
TimeFuncCalc( t, _x, _v, _isubs );
phi( _isubs, t, _._timefunc1, _._timefunc1_1, _._timefunc1_2 );
```

При программировании внешних функций удобно использовать идентификаторы. В примере, который мы рассматриваем, введены три дополнительные идентификатора

phi0, ampl, om, mode,

которые не входят в описание ни одного элемента объекта, а предназначены исключительно для программирования зависимости угла поворота от времени. Поскольку перед

моделированием движения можно изменять значения идентификаторов, можно изменять значения и этих параметров и исследовать зависимость от них кинематики объекта.

Рассмотрим следующие вариант задания функции:

```
procedure phi(_isubs : integer; _t : real_; var _Value, _dValue, _ddValue :
real_);
var  _ : _crankrodVarPtr;
begin
  _ := _PzAll[SubIndx[_isubs]];
  case round(_.mode) of
  0 : begin  //Равномерное движение
      _Value:=_.om*_t+_.phi0;
      _dValue:=_.om;
      _ddValue:=0;
    end;
  1 : begin  //Гармонические колебания
      _Value:=_.ampl*sin(_.om*_t)+_.phi0;
      _dValue:=_.ampl*_.om*cos(_.om*_t);
      _ddValue:=-_.ampl*_.om*_.om*sin(_.om*_t);
    end;
  end;
end;
```

Обратите внимание, что переменная _ (то есть подчеркивание) используется для доступа к идентификаторам объекта (_.om – это идентификатор om). Если такой стиль кажется неудобным, можно ту же самую процедуру написать иначе:

```
procedure phi(_isubs : integer; _t : real_; var _Value, _dValue, _ddValue :
real_);
begin
  with _PzAll[SubIndx[_isubs]]^ do
    case round(mode) of
    0 : begin  //Равномерное движение
        _Value:=om*_t+phi0;
        _dValue:=om;
        _ddValue:=0;
      end;
    1 : begin  //Гармонические колебания
        _Value:=ampl*sin(om*_t)+phi0;
        _dValue:=ampl*om*cos(om*_t);
        _ddValue:=-ampl*om*om*sin(om*_t);
      end;
    end;
end;
```

Таким образом, идентификатор *mode* в процедуре использован в качестве переключателя режимов движения кривошипа: *mode*=0 – равномерное вращение кривошипа; *mode*=1 – гармонические колебания кривошипа. Идентификатор *om* в первом варианте задает угловую скорость равномерного вращения, во втором – частоту колебаний; *ampl* – игнорируется в первом режиме и задает амплитуду колебаний во втором; *phi0* определяет начальный угол поворота кривошипа.

Перед каждым интегрированием уравнений движения следует установить нужные значения идентификаторов, чтобы выбрать режим и его параметры. Некоторые результаты моделирования в различных режимах приведены на рис. 5.3 и рис. 5.4.

Рис. 5.3. Модуль силы реакции в шарнире, соединяющем кривошип с базой (слева) и ускорение ползуна при mode=1, om=1, ampl=2, phi0=0

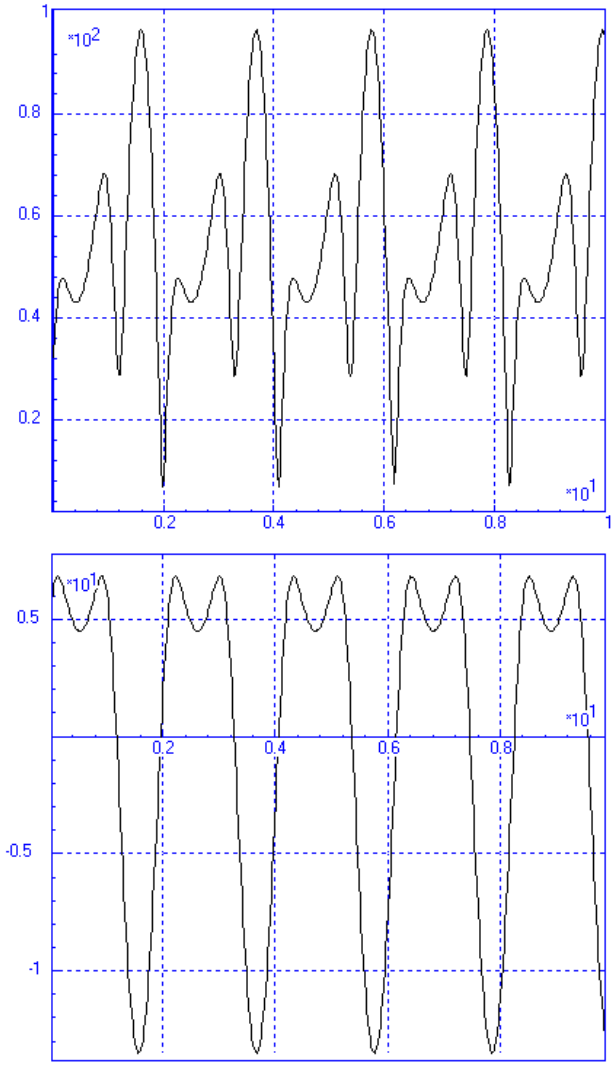


Рис. 5.4. Модуль силы реакции в шарнире, соединяющем кривошип с базой (слева) и ускорение ползуна при mode=0, om=3, phi0=0

Замечание. Будьте внимательны при программировании производных от функций времени. Ошибки в данном случае приводят к неверным результатам моделирования.

5.1.7.2. Программирование шарнирных и биполярных сил

Использование внешних биполярных функций не поддерживается в версиях старше UM 5.0.

5.1.7.3. Программирование графического элемента типа Z-поверхность

В данном разделе мы рассмотрим пример, иллюстрирующий возможности UM создания весьма сложных динамически изменяющихся графических образов (рис. 5.5, объект *Float* в каталоге *Tutorial*). В данном примере с использованием графического элемента типа Z-поверхность получено изображение движущихся волн. Поверхность задается внешней функцией, которой присвоено имя Waves. Графический объект, содержащий данный элемент, назначен в качестве образа сцены. Обратите внимание, что в программе ввода функция Wave не описана, поэтому поверхность изображена в виде прямоугольника с соответствующими размерами (3 × 10 м).

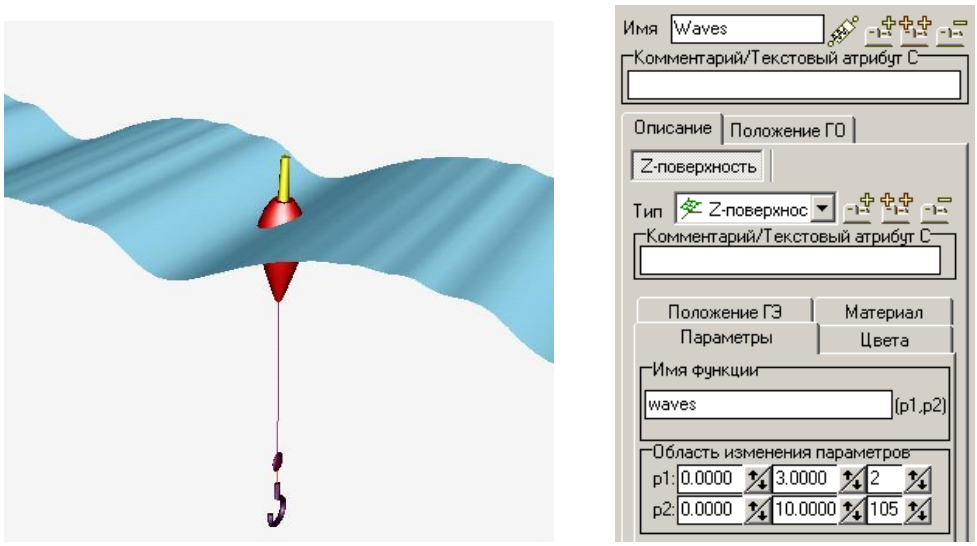


Рис. 5.5

При синтезе уравнений движения в файле управления создается “заготовка” функции *ZGraphicElementFunctions* предназначенной для вычисления всех Z-поверхностей, заданных внешними функциями. Данная функция вызывается программой моделирования всякий раз, когда необходимо обновить соответствующие графический образы (при загрузке объекта, при изменении значений идентификаторов, при вызове пользователем процедуры обновления соответствующих ГО из файла управления).

```
function ZGraphicElementFunctions(_index, _isubs: integer; _p1, _p2: real_):
real_;

var
  _ : _floatVarPtr;

begin
  _ := _PzAll[SubIndx[_isubs]];
  case _index of
    0 : begin
```

```
      { Function waves }
      Result := 0;
    end;
  end;
end;
```

Обратите внимание, что имя программируемой внешней функции вставлено в тело функции *ZgraphicElementFunctions* в виде комментария

```
{ Function waves }
```

в том разделе, где ее необходимо вычислить. По умолчанию функции присваивается нулевое значение, что соответствует образу в виде прямоугольника, так же как и в программе ввода.

В функции *ZGraphicElementFunctions* пользователь должен описать поверхность, то есть организовать вычисление ее значения по передаваемым в функцию декартовым координатам (параметры *_p1, _p2*). В рассматриваемом примере вычисления графического образа волн реализовано в отдельной функции Wave. Форма поверхности волн зависит только от координаты y и времени и представлена суммой десяти гармоник с разными амплитудами, фазами, частотами и фазовыми скоростями.

```
function Wave(t,p : real; var deriv : real_) : real_;
var i : integer;
    phase : real_;
begin
  Result := 0; Deriv:=0;
  for i:=1 to 10 do begin
    phase := p*i*2+2*t*i*sqrt(i)+ln(i);
    Result:=Result+sin(phase)/i/i/5;
    Deriv:=Deriv+2*cos(phase)/i/10;
  end;
end;
```

Функция возвращает, во-первых, координату *z* поверхности через переменную *Result*, во-вторых, производную от функции по координате через переменную *deriv*. Последняя, конечно, не используется при расчете графического образа, но потребуется для вычисления момента от волн, действующего на поплавок при проходе волн. Эффект волнения поверхности “воды” связан с фазой гармоник, зависящей от времени.

В функции *ZGraphicElementFunctions* расчет поверхности организован очень просто через вызов функции Wave:

```
function ZGraphicElementFunctions( _index, _isubs : integer; _p1, _p2 : real_
) : real_;
var  h : real_;
begin
  _ := _PzAll[SubIndx[_isubs]];
  case _index of
    0 : begin
      { Function waves }
      Result:=Wave(t,_p2,h);
    end;
  end;
end;
```

Несмотря на то, что функция, определяющая поверхность волн, зависит от времени, при моделировании это не вызовет никакого эффекта, если не обновлять ГО перед каждой отрисовкой графических окон. Для этого используется программирование сообщения *StepEnd_Message* в процедуре *UserCalc*.

```
procedure UserCalc( _x, _v, _a : VectRPtr; _isubs, _UMMessage : integer; var
WhatDo : integer );
var   Key : integer;
```

```
begin
  Key := WhatDo;
  WhatDo := NOTHING;
  case _UMMessage of
    FORCESCALC_MESSAGE : begin
      try
        ForceFuncCalc( t, _x, _v, _isubs );
      except
        WhatDo := -1;
      end;
    end;    IntegrEnd_Message,
    StepEnd_Message : RefreshElement(eltGO,1,1);
  end;
end;
```

Вызов процедуры обновления ГО при сообщении *IntegrEnd_Message* позволяет вернуть образу его начальное значение после завершения процесса интегрирования уравнений движения.

Замечания.

5. Для того, чтобы графический образ динамически изменялся в процессе XVA-анализа следует обработать сообщения XVASTEP_MESSAGE и XVAEND_MESSAGE точно так же, как *IntegrEnd_Message, StepEnd_Message*, то есть дописать в процедуре *UserCalc* обновление ГО:

   XVAEnd_Message,
   XVAEnd_Message: RefreshElement(eltGO,1,1);

6. Если в функцию расчета графического образа внесено изменение после того, как создан XVA-файл, то в процессе XVA-анализа (“прокрутка” старого файла) ГО будет представлен в новом виде, то есть графическое изображение не будет соответствовать данным, представленным в XVA-файле.

Рассмотрим, каким образом поплавок реагирует на волны. Этот эффект достигается путем программирования дополнительных сил, действующих на поплавок, причем значения сил зависят от состояния поверхности воды в позиции поплавка. Заметим, что модель силы и момента весьма упрощенная, и ее параметры выбраны весьма условно с целью достижения реалистичного динамического поведения объекта. Ниже приведена процедура *ForceFuncCalc*, в которой рассчитываются дополнительные силы.

```
procedure ForceFuncCalc( _t : real_; _x, _v : VectRPtr; _isubs : integer );
const ro : coordin = (0,0,0);
var hWave, dWave : double; //Высота волны и производная по координате y в по-
ложении
                           //поплавка
    r,v,om : coordin;
    frc, trq : coordin;
begin
  GetPoint(1,1,ro,r);   //координаты поплавка в СК0
  GetVel(1,1,ro,v);     //скорость поплавка в СК0
  GetVelAng(1,1,om);    //угловая скорость поплавка в СК0
  hWave:=Wave(t,r[2]+5,dWave); //Высота волны в положении поплавка
  frc[1]:=0; frc[2]:=0;
  frc[3]:=(hWave-r[3]+0.3)*600-100*v[3];  // Архимедова сила и диссипация во-
ды
  trq[1]:=dWave*50-om[1]*10;               // Момент, действующий на поплавок
при
```

```
                                        //набегании волны, вызывает покачи-
вание
  trq[2]:=0;  trq[3]:=0;
// Добавляем силы
  AddForceToBody(1,1,frc,BaseCoordinateSystem);
  AddMomentToBody(1,1,trq,BaseCoordinateSystem);
end;
```

5.1.7.4. Программирование контактной Z-поверхности

Контактные силы типа Z-сфера и Z-окружность позволяют создавать объекты с весьма сложным контактным взаимодействием, особенно при использовании возможностей программирования в среде. При этом, как правило, одновременно используются графические элементы типа Z-поверхность для визуализации контактных поверхностей. В данном разделе мы рассмотрим пример модели четырехколесного автомобиля, управляемого с клавиатуры (рис. 5.6, модель *Robot* в каталоге *Tutorial*).

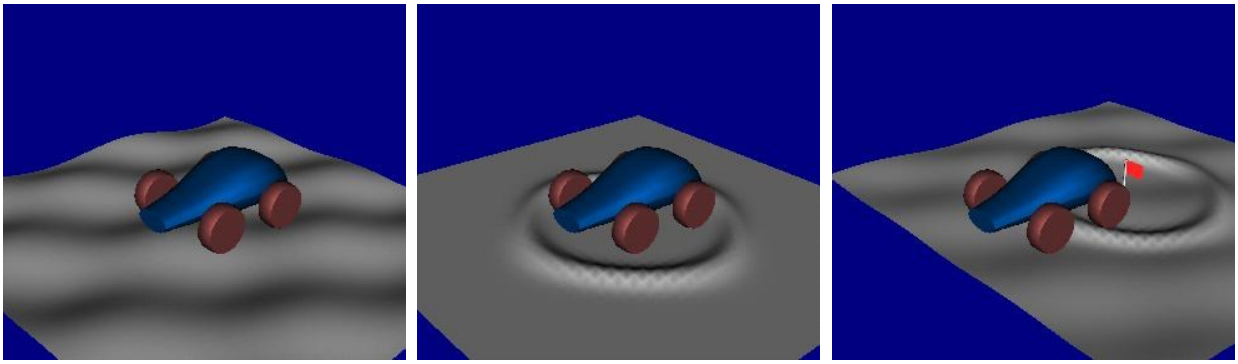


Рис. 5.6

В данном примере мы обсудим следующие вопросы:

- совмещение программирования Z-поверхностей графических элементов и контактных поверхностей;
- организация перемещения графического образа поверхности при движении объекта (“слежение” поверхности за объектом).

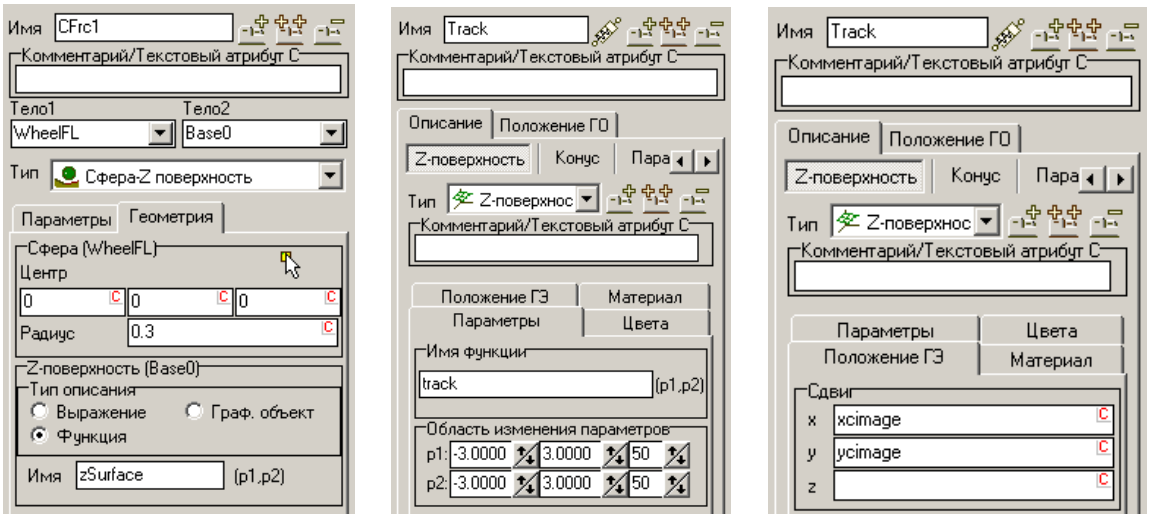


Рис. 5.7

Для расчета взаимодействия с каждым из колес модели используются четыре контактных силовых элемента типа Z-сфера каждый, тип описания – функция, имя функции – *zSurface* для каждого элемента (рис. 5.7, слева). Образ сцены определяется, в частности, элементов типа Z-поверхность с именем Track (рис. 5.7, в центре), причем для сдвига эле-

мента вдоль осей X и Y введены два идентификатора: *xcimage, ycimage* (рис. 5.7, в центре), которым присвоены нулевые значения.

Внешняя функция описания Z-контакта при синтезе уравнений представлена в файле управления функцией

```
procedure zSurface( _isubs : integer; _x, _y : real_; var _z, _dzx, _dzy :
real_ );
var  _ : _robotVarPtr;
begin
  _ := _PzAll[SubIndx[_isubs]];
  _z := 0;
  _dzx := 0;
  _dzy := 0;
end;
```

Входными параметрами являются координаты точки *_x, _y*, в зависимости от которых следует рассчитать:

- *_z* – значение функции в данной точке;
- *_dzx* – значение производной от функции по координате *x*;
- *_dzy* – значение производной от функции по координате *y*.

Поскольку описание образа поверхности задано также внешней функцией, в файле управления добавлена заготовка кода для ее расчета:

```
function ZGraphicElementFunctions( _index, _isubs : integer; _p1, _p2 : real_
) : real_;
var  _ : _robotVarPtr;
begin
  _ := _PzAll[SubIndx[_isubs]];
  case _index of
    0 : begin
      { Function Track }
      Result := 0;
    end;
  end;
end;
```

В файле управления нами написан код, поддерживающий:

- три типа образа поверхности, по которой движется робот и связанные с ними контактные поверхности, привязанные к значениям идентификатора *SurfaceType*;
- управление модели с клавиатуры;
- игровую ситуацию при *SurfaceType-1,2*.

Приведем текст файла управления, сократив его лишь на “пустые” процедуры и интерфейсную часть.

```
uses
  DGetVars, robotC, _Trobot, Windows;

const
     tindex : array[1..5,1..4] of integer =
        (( 1, 1, 1, 1), (*Ahead*)
         (-1,-1,-1,-1), (*Back *)
         (-1, 1,-1, 1), (*Turn left*)
         ( 1,-1, 1,-1), (*Turn right*)
         ( 0, 0, 0, 0));(*STOP*)

(*Modes of the motion: *)
      AHEAD  = 1;
      BACK   = 2;
      LEFT   = 3;
      RIGHT  = 4;
```

```
      STOP   = 5;
(*Coordinates of center of the surface image*)
      XYCImage : array[1..2] of double = (0,0);


 var
     TorqueIndex : array[1..4] of integer;
     xcIndex, ycIndex : integer;
     StopCount : integer;
     hT : double;
     ModeIndex : integer;
     TorqueMaxIndex : integer;
     FFrIndex : integer;

procedure TimeFuncCalc;
begin
end;

procedure CalcZSurface(x,y : double; var z,dzx,dzy : double);
var x1,y1 : double;
    h : real_;
begin
  with _PzAllrobot^[1]^ do
    case round(SurfaceType) of
    0 : begin
        z:=0.1*sin(2*x)*cos(3*y);
        dzx:=0.2*cos(2*x)*cos(3*y);
        dzy:=0.3*sin(2*x)*sin(3*y);
      end;
    1 : begin
        x1:=3*sin(x*pi/6);
        y1:=3*sin(y*pi/6);
        h:=sqrt(x1*x1+y1*y1)-2;
        z:=0.1*exp(-sqr(h*4));
        dzx:=-0.8*exp(-sqr(h*4))*h*x1*pi/2*cos(x*pi/6);
        dzy:=-0.8*exp(-sqr(h*4))*h*y1*pi/2*cos(y*pi/6);
      end;
    2 : begin
        z:=0.05*sin(2*x)*cos(3*y);
        dzx:=0.1*cos(2*x)*cos(3*y);
        dzy:=0.15*sin(2*x)*sin(3*y);
        x1:=3*sin(x*pi/6);
        y1:=3*sin(y*pi/6);
        h:=sqrt(x1*x1+y1*y1)-2;
        z:=z+0.1*exp(-sqr(h*4));
        dzx:=dzx-0.8*exp(-sqr(h*4))*h*x1*pi/2*cos(x*pi/6);
        dzy:=dzy-0.8*exp(-sqr(h*4))*h*y1*pi/2*cos(y*pi/6);
      end;
  end;
end;

procedure zSurface( _isubs : integer; _x, _y : real_; var _z, _dzx, _dzy :
double );
begin
  CalcZSurface(_x,_y,_z,_dzx,_dzy);
end;

function ZGraphicElementFunctions( _index, _isubs : integer; _p1, _p2 : real_
) : real_;
var h1, h2 : double;
begin
  case _index of
    0 : begin
      { Function Track }
```

```
      CalcZSurface(_p1+XYCImage[1],_p2+XYCImage[2],Result,h1,h2);
    end;
  end;
end;

procedure ForceFuncCalc;
  function GetSingleTorque(iwheel : integer; v : real_) : real_;
  begin
    with _PzAllrobot^[1]^ do
      Result:=TorqMax*tindex[round(mode),iwheel]-CResist*v;
  end;
begin
  with _PzAllrobot^[1]^ do begin
    torquefl:=GetSingleTorque(1,_v^[8]);
    torquefr:=GetSingleTorque(2,_v^[10]);
    torquebl:=GetSingleTorque(3,_v^[12]);
    torquebr:=GetSingleTorque(4,_v^[14]);
    SetIdentifierValue(TorqueIndex[1],1, torquefl);
    SetIdentifierValue(TorqueIndex[2],1, torquefr);
    SetIdentifierValue(TorqueIndex[3],1, torquebl);
    SetIdentifierValue(TorqueIndex[4],1, torquebr);
  end;
end;

procedure EstimationFuncCalc( _optMode : integer; _ECCount : integer;
_ECVectOptPtr : TVectOptPtr; var _GoOn : boolean; var _Estimation : real_;
_Msg : pchar );
begin
  _Estimation := 0;
  case _optMode of
    optTest : begin
      _GoOn := false;
    end;
    optPreEstimation : begin
    end;
    optEstimation : begin
    end;
  end;
end;

procedure UserConCalc( _x, _v : VectRPtr; _Jacobi : MatrRPtr; _Error :
Vec3RPtr; _isubs, _ic : integer; _predict : boolean; _nright : integer );
var
  _ : _robotVarPtr;
begin
  _ := _PzAll[SubIndx[_isubs]];
end;

procedure ControlPanelMessage( _x, _v, _a : VectRPtr; _isubs, _index : inte-
ger; _Value : double ); cdecl; export;
begin
  with _pzAll[1]^ do
    case _index of
    0 : begin
        mode:=_Value;
        SetIdentifierValue(ModeIndex,1,_Value);
      end;
    1 : begin
        torqmax:=_value;
        SetIdentifierValue(TorqueMaxIndex,1,_Value);
      end;
    2 : begin
        ffr:=_Value;
        SetIdentifierValue(FFrIndex,1,_Value);
```

```
      end;
    end;
end;

 procedure UserCalc;
  const ro : coordin = (0,0,0);
        Step = 0.5;
  var key : integer;
      r : coordin;
      i : integer;
      changeFlag : boolean;
  begin
    key:=WhatDo;
    WhatDo:=NOTHING;
    case CurrEvent of
    FORCESCALC_MESSAGE : begin
      try
        ForceFuncCalc( t, _x, _v, _isubs );
      except
        WhatDo := -1;
      end;
    end;
    INTEGR_BEGIN :  begin
          Randomize;
          hT:=5;
          _PzAll[1].mode:=AHEAD;
          StopCount:=0;
        end;
    INTEGR_PROCESS : if (t>20) or (_PzAll[1].SurfaceType=0) then begin
        case Key of
        VK_UP    : _PzAll[1].mode:=AHEAD;
        VK_DOWN  : _PzAll[1].mode:=BACK;
        VK_LEFT  : _PzAll[1].mode:=LEFT;
        VK_RIGHT : _PzAll[1].mode:=RIGHT;
        ord('s') : if (StopCount<2) or (_PzAll[1].SurfaceType=0) then begin
            inc(StopCount);
            _PzAll[1].mode:=stop;
          end;
        end;
        SetIdentifierValue(ModeIndex,1,_PzAll[1].mode);
      end;

    FirstInit_Message : begin
        GetElementIndexByName(eltIdentifier, 'xcimage',xcIndex,i);
        GetElementIndexByName(eltIdentifier, 'ycimage',ycIndex,i);
        GetElementIndexByName(eltIdentifier, 'mode',ModeIndex,key);
        GetElementIndexByName(eltIdentifier, 'm_max',TorqueMaxIndex,key);
        GetElementIndexByName(eltIdentifier, 'ffr',FFrIndex,key);
        GetElementIndexByName(eltIdentifier, 'torquefl',TorqueIndex[1],key);
        GetElementIndexByName(eltIdentifier, 'torquefr',TorqueIndex[2],key);
        GetElementIndexByName(eltIdentifier, 'torquebl',TorqueIndex[3],key);
        GetElementIndexByName(eltIdentifier, 'torquebr',TorqueIndex[4],key);
      end;
    StepEnd_Message : begin
        if ((t<20) and (t>hT )) and not (_PzAll[1].SurfaceType=0) then begin
          _PzAll[1].mode:=Random(3)+1;
          hT:=hT+2;
        end;
        GetPoint(1,1,ro,r);
        changeFlag:=false;
        for i:=1 to 2 do
          if r[i]-XYCImage[i]>Step then begin
            XYCImage[i]:=XYCImage[i]+Step;
            changeFlag:=true;
```

```
          end else
            if XYCImage[i]-r[i]>Step then begin
              XYCImage[i]:=XYCImage[i]-Step;
              changeFlag:=true;
            end;
        if changeFlag then begin
          SetIdentifierValue(xcIndex,1,XYCImage[1]);
          SetIdentifierValue(ycIndex,1,XYCImage[2]);
          RefreshElement(eltGO,3,1);
        end;
      end;
    IntegrEnd_Message : begin
        XYCImage[1] := 0;
        XYCImage[2] := 0;
        SetIdentifierValue(xcIndex,1,0);
        SetIdentifierValue(ycIndex,1,0);
        RefreshElement(eltGO,3,1);
      end;
    end;
  end;
```

Для организации слежения образа поверхности за движением робота введена глобальная переменная *XYCImage*

```
XYCImage : array[1..2] of double = (0,0);
```

Первый элемент этого массива определяет координату x, второй – y; эти координаты соответствуют идентификаторам *xcimage, ycimage* (рис. 5.7, справа) и будут использованы для их изменения. Введены также глобальные переменные

xcIndex, ycIndex – индексы идентификаторов xcimage, ycimage;

StopCount – счетчик остановок модели, используется для создания игровой ситуации;

hT – время с момента старта процесса моделирования, в течение которого отключен доступ к управлению моделью, используется для создания игровой ситуации.

Процедура *CalcZSurface* рассчитывает поверхность, по которой перемещается модель. Эта процедура вызывается как из функции расчета образа поверхности *ZgraphicElementFunctions*, так и из процедуры расчета контактной поверхности *zSurface*. Тем самым достигается совпадение графического образа и контактной поверхности. Обратите внимание, что при расчете образа поверхности к значениям параметров *p1,p2* добавляется сдвиг центра образа с целью его перемещения за модель:

```
CalcZSurface(_p1+XYCImage[1],_p2+XYCImage[2],Result,h1,h2);
```

В процедуре обработки сообщений в процедуре *UserCalc* определяются и рассчитываются:

- индексы идентификаторов *xcimage, ycimage*, сообщение *FirstInit_Message*;
- положение центра образа поверхности *XYCImage*, сообщение *StepEnd_Message*; изменение происходит, если центр масс кузова смещается относительно текущего положения центра образа вдоль одной из осей на величину, большую *Step*=0.5; если происходит сдвиг, то устанавливаются новые значения идентификаторов x*cimage, ycimage* (процедура *SetIdentifierValue*) и обновляется соответствующий ГО (процедура *RefreshElement*).

Сообщение *IntegrEnd_Message* используется для установки нулевого сдвига образа поверхности.

Для того, чтобы камера в анимационном окне следила за движением модели, используется функция *Следить за…* в окне параметров анимационного окна (рис. 5.8, используйте пункт *Параметры окна* из всплывающего меню для вызова окна, представленного на рисунке) и установите тело, за которым будет следить камера.

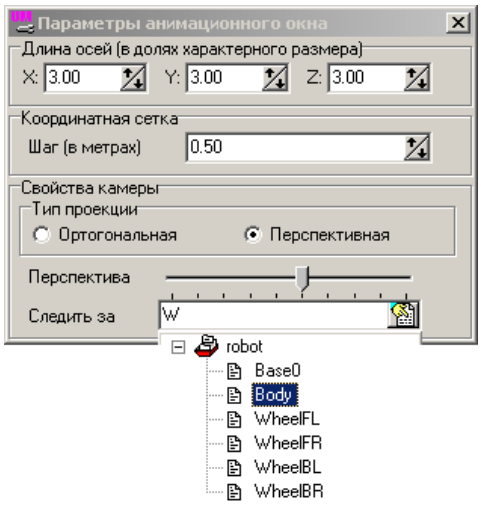


Рис. 5.8

Игровая ситуация, соответствующая значениям идентификатора SurfaceType=1,2, состоит в следующем: управляя роботом с помощью клавиш ←↑→↓, S (остановка), следует привести его *внутрь* начальной окружности (помечена флажком) и остановить (либо заставить крутиться) в ее пределах. В течение первых двадцати секунд управление с клавиатуры отключено, и модель движется случайным образом. В течение одной игры позволяется лишь дважды нажать клавишу S (остановка). Для ориентации можно использовать графическое окно, в котором строится траектория модели, причем цели соответствует начало координат. Для усложнения задачи следует уменьшать коэффициент трения (идентификатор *ffr*) и/или увеличивать движущий момент (идентификатор *torqmax*).

5.1.8. Отладка файлов управления в среде Delphi

В этом параграфе будет показано, как использовать среду программирования *Delphi* для отладки *файлов управления*.

После синтеза уравнений движения (не забудьте выбрать *Pascal* в качестве языка выходных файлов при наличии нескольких компиляторов) откройте в *Delphi* проект модели (файл *umtask64.dpr*) из каталога *ИмяМодели\Pascal*. Файл управления имеет название *Cl[ИмяМодели].pas*. Для компиляции и отладки проекта необходимо выполнить следующие настройки.

1. Опции проекта

Откройте диалог настроек проекта – пункт меню **Project | Options** – и перейдите на вкладку **Directories/Conditionals**.

- В поле **Output directory** введите путь к каталогу модели – *..\[ИмяМодели]*. Дело в том, что без указания каталога для выходной *.dll она будет создана в каталоге с файлом проекта, то есть в каталоге *..\ИмяМодели\Pascal*.
- В поле **Search Path** введите путь к каталогу com, входящему в поставку UM: **{Данные УМ}\com**.

Настроив опции проекта (см. рис. 5.9) уже можно попытаться откомпилировать проект. При необходимости исправьте ошибки в файле управления и откомпилируйте проект снова. После успешной компиляции расставьте точки останова в интересующих вас местах файла управления и выполните настройки параметров запуска.

Параметры запуска

Откройте диалог настроек параметров запуска – пункт меню **Run | Parameters** – и перейдите на вкладку **Local**.

- В поле **Host Application** выберите файл модуля моделирования – **umsimul.exe** (*C:\Program Files\UM Software Lab\Universal Mechanism\10\bin\UmSimul.exe*). **Host Application** – это приложение, которое использует процедуры *.dll, которая находится в отладке. В случае с UM **Host Application** – это модуль моделирования, а *.dll – это библиотеку с уравнениями движения модели.
- В поле **Parameters** выберите каталог модели ..\[ИмяМодели], это позволит автоматически открывать нужную модель при запуске модуля моделирования.

После выполнения этих настроек (см. рис. 5.10) запускайте проект на выполнение (пункт меню **Run | Run**). Запуститься модуль моделирования UM, который подгрузит указанную вами в поле **Parameters** модель. Файл управления готов к отладке.

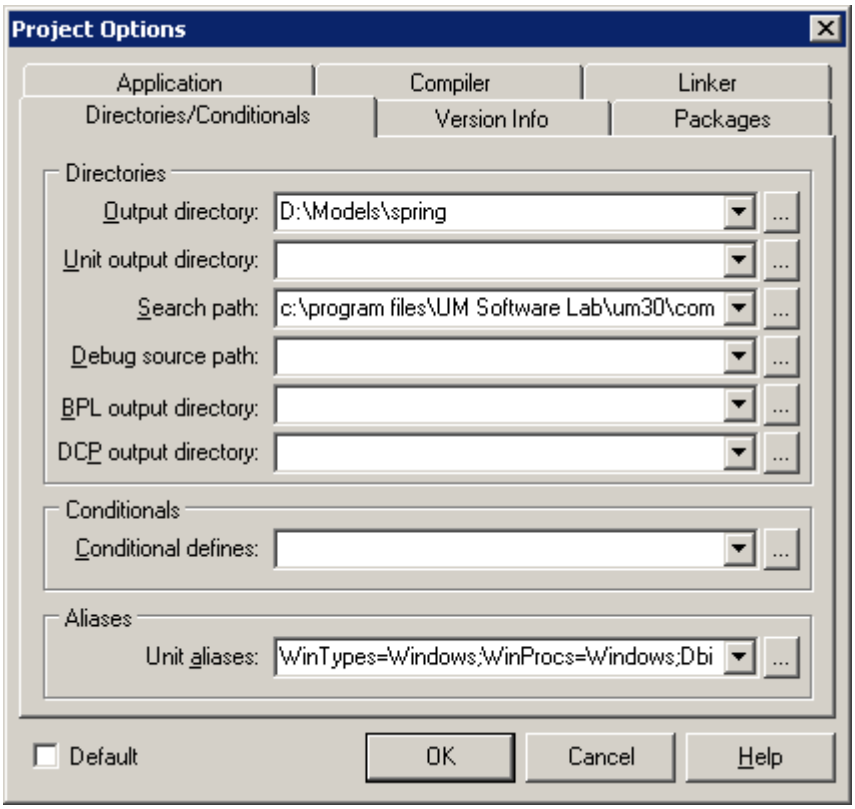


Рис. 5.9

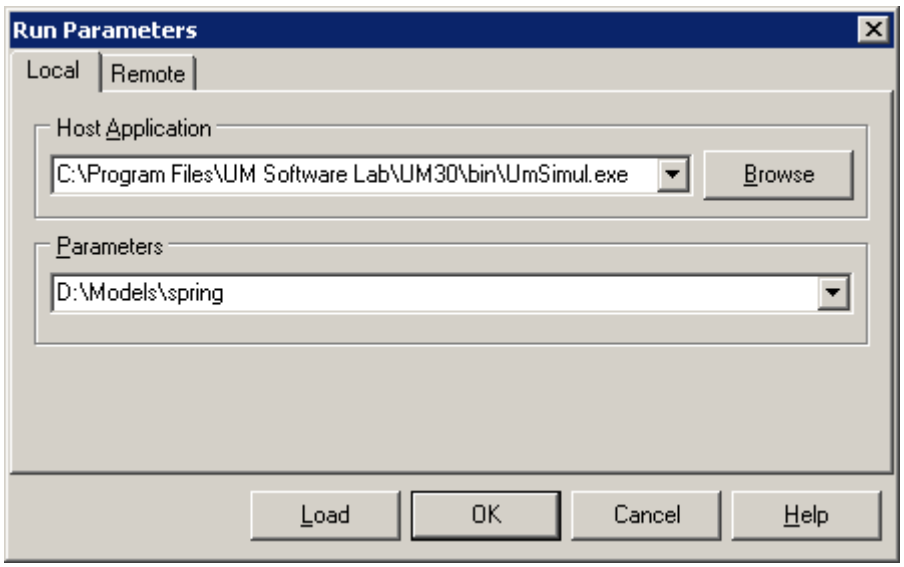

Run Parameters
Local
Remote
Host Application
C:\Program Files\UM Software Lab\UM30\bin\UmSimul.exe
Browse
Parameters
D:\Models\spring
Load
OK
Cancel
Help

Рис. 5.10

5.2. Программирование функционалов

Реализация функционалов, которые используются в *табличном процессоре* (см. п. 4.3.7 *"Процессор переменных"*), вынесена в файлы DLL, которые находятся в каталоге **Plugins**. Вместе со стандартной версией UM поставляется файл **standard.dll**, в котором находятся стандартные функционалы. Вместе с модулем моделирования ж/д экипажей поставляется файл **Railway.dll**, где сосредоточены некоторые дополнительные функционалы, используемые при моделировании ж/д экипажей.

При запуске модуля моделирования все внешние функции, находящиеся во всех **DLL** каталога **Plugins** автоматически подгружаются как функционалы.

Таким образом, разработка дополнительных функционалов может осуществляться независимо от UM самим пользователем или разработчиками, по заказу пользователя. После создания новую DLL достаточно просто поместить в каталог Plugins. После добавления новой DLL в каталог Plugins для того, чтобы новые функционалы стали доступны, необходимо перезапустить модуль моделирования. В каталоге Plugins может быть сколь угодно много DLL, причем каждая DLL может содержать произвольное число экспортируемых процедур, которые в дальнейшем будут интерпретироваться как функционалы. Все экспортные процедуры в этих DLL должны быть одного типа. Этот тип описан в файле **{Данные УМ}\com\plugin.pas**:

```
TFunctional = procedure(
    X, Y   : umPointer;        // Массивы точек по абсциссе и ординате.
                               // Указатель на массив umDouble
    N      : umInteger;        // Длина массивов X и Y
var Value  : umDouble;         // Возвращаемое значение
var Success: boolean           // Флаг успешного вычисления функционала
); cdecl;
```

Недопустимо создание экспортируемых процедур в любой из DLL несовместимых с вышеописанным типом. Пример реализации DLL на языке Pascal – в файле **{Данные УМ}\plugins\standard.dpr**.

Следующий пример иллюстрирует создание новой DLL (файл проекта Delphi). В данной DLL реализован один функционал – **Example_Min**.

```
(*********************************************************)
(*  Example library of functionals                       *)
(*  Copyright (c) 2001 UM software lab                   *)
(*                                                       *)
(*  Table of functionals:                                *)
(*                                                       *)
(*  Example_Min                                          *)
(*********************************************************)
library example;

uses
  UmTypes,
  Plugin;

type
   TUmDoubleArray = array [0..65535] of umDouble;
   TUmDoubleArrayPtr = ^TUmDoubleArray;
```

```
  procedure Example_Min( X, Y: umPointer;
                             N: umInteger;
                   var Value: umDouble;
                 var Success: boolean); cdecl; export;
  var aY: TUmDoubleArrayPtr;
      i: integer;
  begin
    Value:=0; Success:=true;
    if N>0 then begin
      aY:=Y;
      Value:=aY[0];
      for i:=1 to N-1 do
        if aY[i]<Value then Value:=aY[i];
    end else Success:=false;
  end;

exports
  Example_Min;

end.
```

При компиляции подобного проекта не забудьте установить пути поиска к файлам UmTypes и Plugin, которые находятся в каталоге **{Данные УМ}\com**.

Следующий пример иллюстрирует создание новой DLL (файл из проекта VC++ 6.0). В данной DLL реализован функционал – **Ampl** (амплитуда).

```
extern "C"
void Ampl(double* x, double* y, int n, double& value, bool& success)
{
  int i,j;
  double min = y[0];
  double max = y[0];
  if (n > 0)
    for (i = 1; i < n; i++)
      for (j = i; j < n - 1; j++)
      {
        if (y[i] < min)
          min = y[i];
        if (y[i] > max)
          max = y[i];
      }
  value = (max - min)*0.5;
  success = true;
}
```

В проект так же необходимо добавить def-файл с указанием экспортируемых функций. В данном случае экспортируется функция Ampl:

```
LIBRARY      "FuncC"
DESCRIPTION  'FuncC Dynamic Linked Library'
EXPORTS
  Ampl;
```

5.3. Создание и использование внешних библиотек

Внешние библиотеки обычно используются для подключения к «Универсальному механизму» математических моделей сил, которые невозможно описать с помощью встроенных силовых элементов. Такой метод является альтернативой программированию в *файле управления* и имеет следующие отличия:

- для разработки собственных библиотек пользователь может использовать любую инструментальную среду и любой компилятор, которые поддерживают создание динамически загружаемых библиотек (DLL);
- пользователю не обязательно разбираться в особенностях программирования в *файле управления*;
- разработанные ранее библиотеки (DLL) подключаются в «Универсальный механизм» через визуальный интерфейс пользователя, без необходимости дальнейшего программирования.

В общем случае моделирование динамики механических систем с подключением внешних библиотек предполагает выполнение следующих этапов.

- Разработка математической модели и реализация этой модели в виде программного кода в соответствии с принятыми соглашениями.
- Компиляция модели в виде динамически загружаемой библиотеки (DLL).
- Подключение DLL к модели механической системы с помощью **Мастера связи с внешними библиотеками**. Связывание входных и выходных величин модели из DLL с переменными и параметрами модели УМ.
- Моделирование динамики получившейся системы.

Внешние библиотеки имеют список входных и выходных переменных, а также список параметров. На этапе связывания внешней библиотеки и модели УМ на вход внешней библиотеки подаются переменные, созданные в *Мастере переменных*, обычно описывающие кинематику системы. Выходные переменные внешней библиотеки связываются с параметрами модели, которые обычно описывают силы и моменты, действующие на тела механической системы.

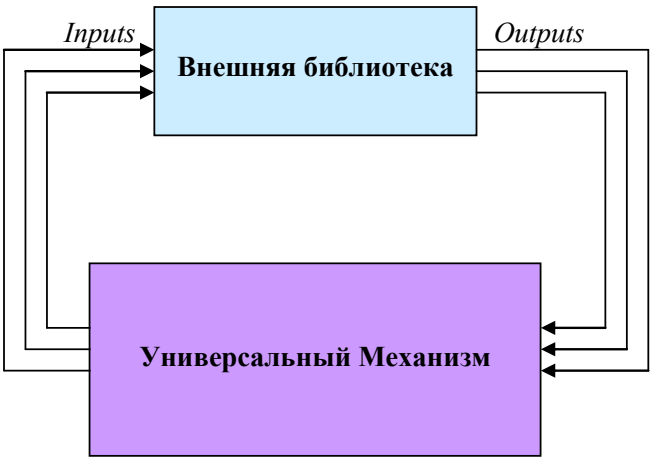


Рис. 5.11.

5.3.1. Мастер связи с внешними библиотеками

Мастер связи с внешними библиотеками позволяет пользователю подключить к УМ внешние библиотеки различной природы, которые логически объединены в модуль **UM Control**. Ниже кратко рассмотрим эти библиотеки.

- Структурные схемы, созданные в программных комплексах Matlab/Simulink или SimInTech и экспортированные в виде DLL. Для поддержки таких DLL необходимs инструмент **UM Control/Matlab Import** или **UM Control/SimInTech Import** соответственно. Более подробно об импорте структурных схем, импортированных из Matlab/Simulink см. в следующей главе руководства пользователя УМ: см. раздел «Начинаем работать: интерфейс с Matlab/Simulink и SimInTech»[2], пп. «3.1. Моделирование с использованием Matlab Import» и «4.2. Моделирование с использованием SimInTech Import». Скачать последнюю версию этого руководства можно по адресу https://www.universalmechanism.com/download/10/rus/gs_um_control.pdf.
- Библиотеки пользователя, написанные на любом языке программирования и откомпилированные в виде УМ-совместимых динамически-загружаемых библиотек, то есть библиотек, предоставляющих вызывающему приложению – УМ – предопределенный набор экспортных функций, описанных в п. 5.3.2. *"Декларация процедур"*, с. 5-56. Для поддержки таких DLL необходим инструмент **UM Control/User-defined routines**, описанный ниже.
- Структурные схемы, созданные с помощью встроенного редактора схем. Более подробно о встроенном редакторе схем читайте в разделе «Редактор структурных схем[3]». Скачать последнюю версию этого руководства можно по адресу www.universalmechanism.com/download/10/rus/24_um_blockeditor.pdf.

Описанные ниже в п. 5.3.4. *"Подключение внешних библиотек"*, с. 5-67 принципы подключения внешних библиотек не зависят от способа их реализации и являются общими для всех типов внешних библиотек.

5.3.2. Декларация процедур

Шаблоны внешних библиотек на языках *C* и *Pascal* находятся в каталоге {Данные УМ}\samples\tutorial\extlibrary\templates.

В поставку УМ входит пример реализации линейной пружины с использованием внешних библиотек. Пример находится в каталоге {Данные УМ}\samples\tutorial\extlibrary\extlibspring. Исходные тексты этой внешней библиотеки даны в каталоге {Данные УМ}\samples\tutorial\extlibrary\source.

Рассмотрим процедуры, которые должны содержаться в DLL внешней библиотеки и типы этих процедур.

Замечание. Возвращаемый целочисленный параметр **status** – код завершения процедуры.

[2] *{Данные УМ}\manual\getting started\gs_um_control.pdf*
[3] *{Данные УМ}\manual\24_um_blockeditor.pdf*

Возвращаемое значение:
0 – нет ошибок;
> 0 – процедура выполнена с ошибками. При обнаружении ненулевого кода возврата код ошибки выводится в сообщении.
Далее по тексту смысл этого параметра более не поясняется.

Замечание. Для чтения объявлений функций на C++ обратите внимание на описание следующих типов:
typedef double* PDouble;
typedef wchar_t* PChar;

Замечание. Символьные значения, возвращаемые из библиотеки (имя модели, имена входных и выходных сигналов и параметров) должны быть описаны как UNICODE-строки.
Обратите внимание, что, начиная с DELPHI 2009, тип PChar тождественно равен PWideChar (строки UNUCODE), в более ранних версиях Delphi, PChar тождественно равен PANSIChar.

Замечание. Для всех функций используется модель вызова/передачи параметров (Calling convention) **cdecl**.

`EXT_Initialize`

Delphi/Pascal синтаксис:

```
procedure EXT_Initialize(var status: Integer); cdecl;
```

C++ синтаксис:

```
void _cdecl EXT_Initialize(int& status)
```

Процедура начальной инициализации. Используется для выделения памяти под данные библиотеки, инициализации переменных, открытия файлов и т.д. Вызывается каждый раз перед запуском процесса интегрирования уравнений движения в программе **UM Simulation (Инспектор моделирования объекта/Интегрирование)**. Обратите внимание, что при загрузке библиотеки и инициализации параметров процедура может вызываться последовательно несколько раз без «парного» вызова *EXT_Terminate*. При реализации процедуры учитывайте этот факт для того, чтобы не «замусоривать» оперативную память компьютера многократным выделением памяти под одни и те же данные.

`EXT_Terminate`

Delphi/Pascal синтаксис:

```
procedure EXT_Terminate(var status: Integer); cdecl;
```

C++ синтаксис:

```
void _cdecl EXT_Terminate(int& status )
```

Процедура вызывается перед выгрузкой библиотеки из памяти. Предназначена для совершения заключительных действий: освобождения памяти, закрытия файлов и т.д.

`EXT_GetModelName`

Delphi/Pascal синтаксис:

```
procedure EXT_GetModelName(name: PWideChar; var status: Integer); cdecl;
```

C++ синтаксис:

```
void _cdecl EXT_GetModelName(WChar name, int& status)
```

Процедура возвращает имя модели (не более 255 символов), реализованной в данной внешней библиотеке, которое отображается в интерфейсе пользователя. Память выделяется на стороне вызывающего кода. В процедуре выделять память не нужно.

`EXT_GetNumU`

Delphi/Pascal синтаксис:

```
procedure EXT_GetNumU(var num: integer; var status: Integer); cdecl;
```

C++ синтаксис:

```
void _cdecl EXT_GetNumU(int& num, int& status)
```

Процедура возвращает количество входных параметров для математической модели, реализованной в данной внешней библиотеке.

`EXT_GetUName`

Delphi/Pascal синтаксис:

```
procedure EXT_GetUName(i: integer; name: PWideChar; var status:
Integer); cdecl;
```

C++ синтаксис:

```
void _cdecl EXT_GetUName(int i, WChar name, int& status)
```

Процедура возвращает имя *i*-ой входного параметра, где *i* – индекс параметра, не более 255 символов. Индексация начинается с 0. Память выделяется на стороне вызывающего кода. В процедуре выделять память не нужно.

`EXT_GetNumY`

Delphi/Pascal синтаксис:

```
procedure EXT_GetNumY(var num: integer; var status: Integer); cdecl;
```

C++ синтаксис:

```
void _cdecl EXT_GetNumY(int &num, int& status)
```

Процедура возвращает количество выходных параметров для математической модели, реализованной в данной внешней библиотеке.

`EXT_GetYName`

Delphi/Pascal синтаксис:

```
procedure EXT_GetYName(i: integer; name: PWideChar;
var status: Integer); cdecl;
```

C++ синтаксис:

```
void _cdecl EXT_GetYName(int i, WChar name, int& status)
```

Процедура возвращает имя *i*-ой выходного параметра, где *i* – индекс параметра, не более 255 символов. Индексация начинается с нуля. Память выделяется на стороне вызывающего кода. В процедуре выделять память не нужно.

`EXT_GetY`

Delphi/Pascal синтаксис:

```
procedure EXT_GetY(time: double; U, X, Y: PDouble;
var status: integer); cdecl;
```

C++ синтаксис:

```
void _cdecl EXT_GetY(double time, PDouble U, PDouble X, PDouble Y, int& sta-
tus)
```

Основная расчетная процедура, которая на основании текущего времени (*time*) и входных параметров (*U*) определяет вектор выходных параметров (*Y*). Память под массивы *U, X, Y* выделяется на стороне вызывающего кода. *U* и *Y* – указатели на первый элемент соответствующих массивов. *X* – указатель на массив переменных состояния, зарезервирован для будущего использования, в настоящей версии программы не используется.

`EXT_GetNumParameters`

Delphi/Pascal синтаксис:

```
procedure EXT_GetNumParameters(var num: integer;
var status: Integer); cdecl;
```

C++ синтаксис:

```
void _cdecl EXT_GetNumParameters(int& num, int& status)
```

Возвращает число параметров модели.

`EXT_GetParameters`

Delphi/Pascal синтаксис:

```
procedure EXT_GetParameters(value: PDouble; var status:
integer); cdecl;
```

C++ синтаксис:

```
void _cdecl EXT_GetParameters(PDouble value, int& status)
```

Возвращает значения параметров модели. *Value* – указатель на первый элемент вектора параметров. Память выделяется на стороне вызывающего кода. В процедуре выделять память не нужно.

EXT_GetParameterName

Delphi/Pascal синтаксис:

```
procedure EXT_GetParameterName(i: integer; name: PWideChar;
var status: Integer); cdecl;
```

C++ синтаксис:

```
void _cdecl EXT_GetParameterName(int i, WChar name,
int& status)
```

Процедура возвращает имя *i*-ой выходного параметра модели, где *i* – индекс параметра. Индексация начинается с нуля. Память под переменную *name* выделяется на стороне вызывающего кода. В процедуре выделять память не нужно.

EXT_SetParameters

Delphi/Pascal синтаксис:

```
procedure EXT_SetParameters(numpara: integer; para: PDouble;
var status: integer); cdecl;
```

C++ синтаксис:

```
void _cdecl EXT_SetParameters(int numpara, PDouble para, int& status)
```

Устанавливает значения параметров модели, *para* – указатель на первый элемент вектора параметров. Память выделяется на стороне вызывающего кода. В процедуре выделять память не нужно.

EXT_StepConfirmed

Delphi/Pascal синтаксис:

```
procedure EXT_StepConfirmed; cdecl;
```

C++ синтаксис:

```
void _cdecl EXT_StepConfirmed()
```

Функция опциональна и может быть опущена при разработке библиотеки пользователя. Имеет смысл только в том случае, если внешняя библиотека в процедуре **EXT_GetY** использует данные с предыдущих шагов, например, включает собственный решатель или

использует другие алгоритмы расчета с памятью. В противном случае может быть пустой или вовсе исключена из библиотеки. ПК «Универсальный механизм» для интегрирования уравнений движения использует численные методы с переменным шагом и контролем точности. Вследствие этого некоторые шаги численного метода могут отменяться и дробиться на более мелкие для достижения заданной точности интегрирования. Если очередной шаг численного метода выполнен успешно, то вслед за вызовом **EXT_GetY** следует вызов **EXT_StepConfirmed**, который и подтверждает, что шаг до момента времени, с которым был сделан последний вызов **EXT_GetY**, выполнен успешно.

Описанные ниже функции **EXT_GetNumStoredVars, EXT_GetStoredVars** и **EXT_SetStoredVars** используются для обеспечения поддержки расчета нескольких однотипных силовых элементов с помощью одной динамической библиотеки, подробнее см. п. 5.3.4.3. *"Особенности подключения нескольких библиотек"*, с. 5-69. Функции имеют смысл только в том случае, если внешняя библиотека в процедуре **EXT_GetY** не может вычислить выходные сигналы по входным сигналам и значениям параметров. Другими словами, если **EXT_GetY** использует данные с предыдущих шагов или состояние системы на предыдущих шагах. В противном случае перечисленные выше функции могут отсутствовать в библиотеке.

```
EXT_GetNumStoredVars
```

Delphi/Pascal синтаксис:

```
procedure EXT_GetNumStoredVars
 (var num: integer; var status: Integer); cdecl;
```

C++ синтаксис:

```
void _cdecl EXT_GetNumStoredVars
```

Функция опциональна и может быть опущена при разработке библиотеки пользователя. Функция возвращает количество сохраняемых переменных. Подробнее о сохраняемых переменных см. п. 5.3.4.3. *"Особенности подключения нескольких библиотек"*, с. 5-69.

```
EXT_GetStoredVars
```

Delphi/Pascal синтаксис:

```
procedure EXT_GetStoredVars (value: PDouble;
var status: integer); cdecl;
```

C++ синтаксис:

```
void _cdecl EXT_GetNumStoredVars
```

Функция опциональна и может быть опущена при разработке библиотеки пользователя. Функция записывает сохраняемые переменные по указанному адресу, value – указатель на первый элемент вектора сохраняемых переменных. Количество записываемых переменных должно соответствовать количеству, возвращаемому функцией

EXT_GetNumStoredVars. Память выделяется на стороне вызывающего кода. В процедуре выделять память не нужно.

```
EXT_SetStoredVars
```

Delphi/Pascal синтаксис:

```
procedure EXT_SetStoredVars (numpara: integer; para: PDouble; var status: integer); cdecl;
```

C++ синтаксис:

```
void _cdecl EXT_GetNumStoredVars
```

Функция опциональна и может быть опущена при разработке библиотеки пользователя. Функция восстанавливает значения сохраняемых переменных. Входные параметры: *numpara – количество сохраняемых переменных*, *para* – указатель на первый элемент вектора сохраненных переменных.

5.3.3. Особенности компиляции внешних библиотек

Разработчики внешних библиотек должны позаботиться о том, чтобы перечисленные выше функции (процедуры) были объявлены как экспортируемые. Кроме того, нужно следить за тем, чтобы имена экспортных функций в DLL не были декорированы. Декорированные имена функций – это модифицированные по некоторому соглашению имена, которые кроме имени функций содержат информацию о количестве и типе аргументов.

5.3.3.1. Компиляция внешних библиотек на C/C++

Обратите внимание, что по умолчанию имена экспортируемых функций при компиляции DLL с помощью Microsoft Visual C++ и Microsoft Visual Studio декорируются. Для управления списком экспортируемых функций и отмены декорирования их имен рекомендуется пользоваться DEF-файлом, см. пример исходного текста и DEF-файла в каталоге {Данные УМ}\samples\tutorial\extlibrary\source\c.

Для использования DEF-файла при работе в Microsoft Visual Studio скопируйте файл в каталог проекта, добавьте его в проект и в настройках проекта в поле **Configuration properties | Linker | Input | Module Definition File** установите DEF-файл вашего проекта, как это показано на рис. 5.12.

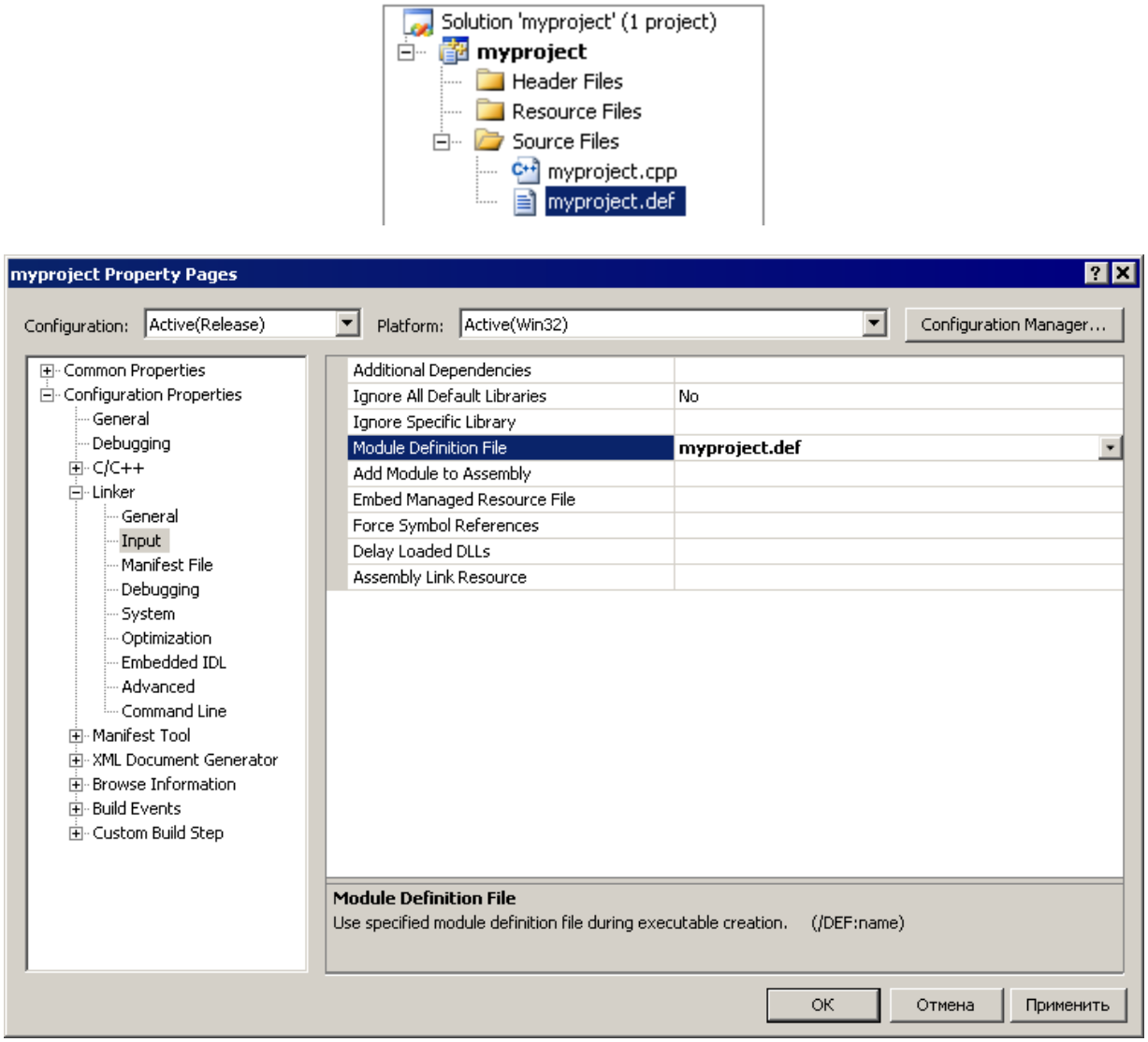


Рис. 5.12. Настройки проекта в MS Visual Studio

Содержание DEF-файла:

```
EXPORTS
  EXT_Initialize
  EXT_Terminate
  EXT_GetModelName
  EXT_GetNumU
  EXT_GetUName
  EXT_GetNumY
  EXT_GetYName
  EXT_GetY
  EXT_GetNumParameters
  EXT_GetParameters
  EXT_GetParameterName
  EXT_SetParameters
  EXT_StepConfirmed
```

Обратите внимание, что проблема с описанием экспортируемых функций в DLL не имеет единого решения и зависит от используемого компилятора C/C++. При использовании ряда компиляторов C/C++ возможно следует использовать декларации функций extern "C" и/или __declspec(dllexport) как показано на примере ниже:

```
extern "C" __declspec(dllexport) void _cdecl EXT_Initialize(int& status);
```

```
extern "C" __declspec(dllexport) void _cdecl EXT_GetModelName(WChar name,
int& status);
```

5.3.3.2. Компиляция внешних библиотек на Pascal

Для определения списка экспортируемых процедур используйте ключевое слово **Exports**, см. пример в файле

{Данные УМ}\samples\tutorial\extlibrary\source\pascal\UMLinearSpring.dpr.

```
...
Exports
  EXT_Initialize,
  EXT_Terminate,
  EXT_GetModelName,
  EXT_GetNumU,
  EXT_GetUName,
  EXT_GetNumY,
  EXT_GetYName,
  EXT_GetY,
  EXT_GetNumParameters,
  EXT_GetParameters,
  EXT_GetParameterName,
  EXT_SetParameters,
  EXT_StepConfirmed;
```

При компиляции DLL с помощью средства разработки Delphi имена процедур по умолчанию не декорируются и предпринимать специальные усилия по этому поводу не требуется.

5.3.3.3. Устранение ошибок

В случае возникновения ошибок при подключении внешней DLL до обращения в службу поддержки убедитесь

- в том, что ваша конфигурация программного комплекса «Универсальный механизм» включает инструмент **UM User-Defined Routines** из модуля **UM Control**;
- в том, что ваша DLL и вызывающий исполняемый файл umsimul.exe (**UM Simulation**) имеют одинаковую разрядность (64 бита); **UM Simulation** имеет разрядность 64 бит;
- а также в том, что разработанная Вами DLL имеет необходимый список экспортных функции и что имена этих функций не декорированы.

Проверить список экспортных функций, описанных в DLL можно с помощью утилит сторонних разработчиков:

- утилита **dumpbin.exe** из комплекта установки Microsoft Visual Studio;
- утилита **tdump.exe** из комплекта Embarcadero RAD Studio;
- бесплатная (freeware) утилита **DLL Export Viewer**,
  описание: www.nirsoft.net/utils/dll_export_viewer.html,
  ссылка на программу: www.nirsoft.net/utils/dllexp.zip.

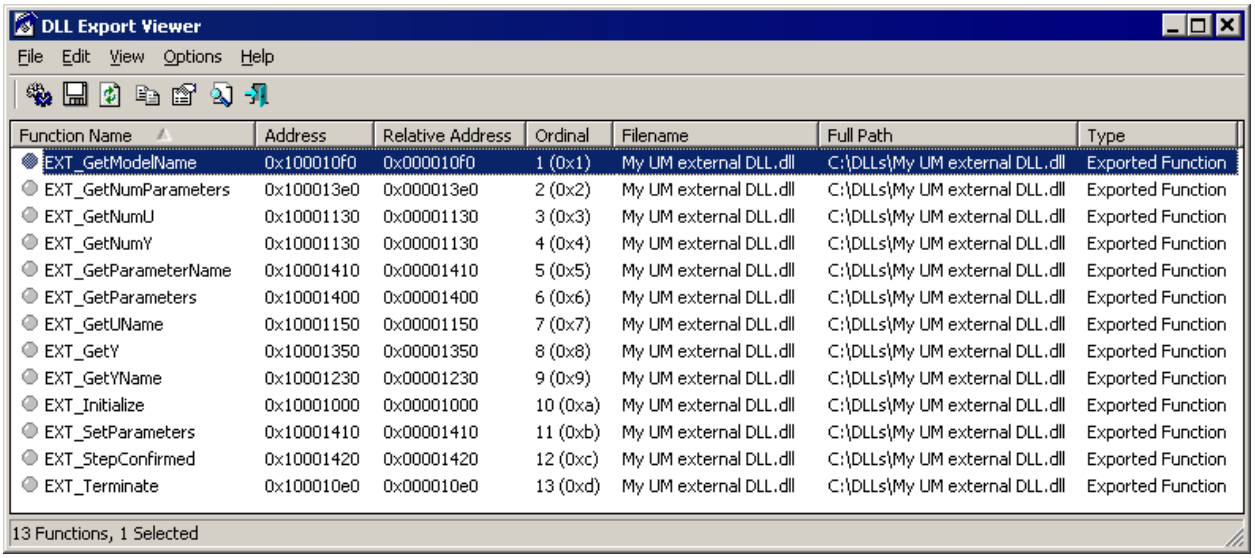


Рис. 5.13. Отчет об экспортных функциях DLL в программе **DLL Export Viewer**

5.3.4. Подключение внешних библиотек

В программном комплексе «Универсальный механизм» предусмотрено два способа подключения внешних библиотек[4]. Первый, более общий способ, заключается в использовании **Мастера связи с внешними библиотеками** программы **UM Simulation**. Второй способ является частным случаем первого и заключается в использовании скалярных сил типа *Библиотека (DLL)* в программе **UM Input**. Правила создания DLL в обоих случаях одинаковы (см. п. 5.3.2. *"Декларация процедур"*, с. 5-56), отличаются лишь способы подключения математических моделей из внешних библиотек к расчетным схемам в УМ, подробности см. ниже.

5.3.4.1. Мастер связи с внешними библиотеками

В общем случае подключение внешних библиотек выполняется в программе **UM Simulation**, пункт меню **Инструменты | Внешние интерфейсы | Интерфейс с внешними библиотеками**. Откроется окно **Мастера связи с внешними библиотеками**, см. рис. 5.14. Слева находится список внешних библиотек, справа – настройки активной (выделенной в списке) библиотеки.

На вход внешних библиотек назначаются переменные модели (см. список **Входные величины** на рис. 5.14). Для назначения переменной на некоторый вход внешней библиотеки, её нужно просто перетащить мышкой, пользуясь обычной технологией *Drag-and-Drop* из **Мастера переменных** или **Списка переменных**. Таким образом, любая переменная, которую можно создать в **Мастере переменных**, доступна для назначения на вход внешней библиотеки.

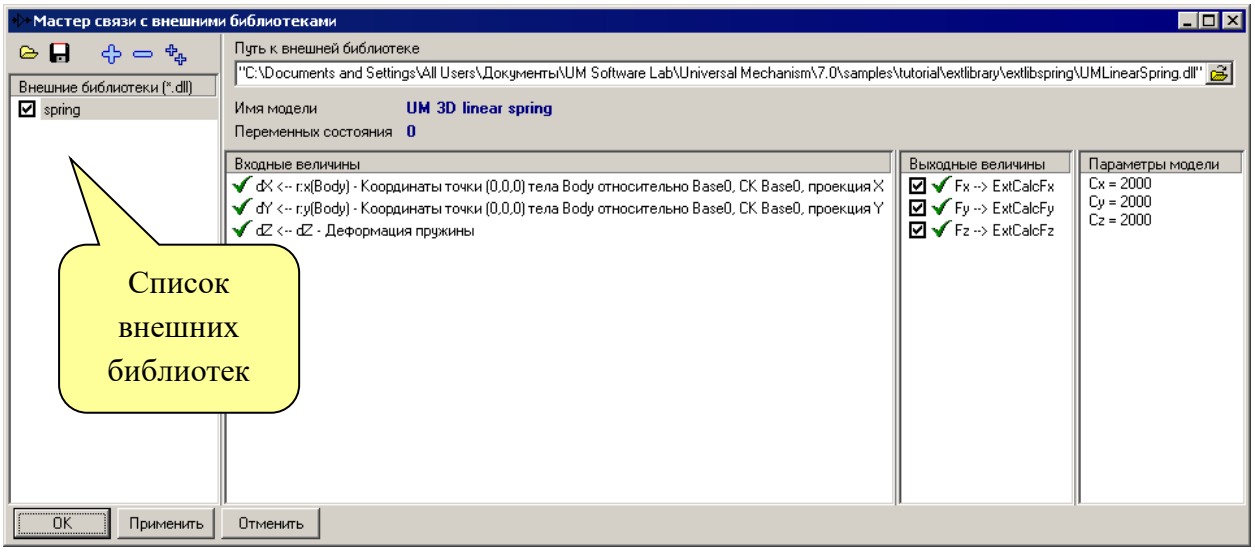


Рис. 5.14. Мастер связи с внешними библиотеками

Выходные величины внешних библиотек обычно связываются с параметрами модели. Поэтому, несмотря на то, что внешние библиотеки подключаются к модели только в про-

[4] Здесь речь идет о внешних библиотеках, поддерживаемых инструментом **User-Defined Routines** – библиотеках, реализованных на некотором языке программирования и откомпилированных в виде УМ-совместимых динамически-загружаемых библиотек (DLL)

грамме моделирования динамики (**UM Simulation**), создать и должным образом параметризовать соответствующие силовые элементы нужно еще на этапе подготовки модели в программе **UM Input**.

Для выбора параметра модели, которому будет присваиваться значение, получаемое из внешней библиотеки, сделайте двойной щелчок на нужном элементе в списке **Выходные величины**, см. рис. 5.14. Откроется диалоговое окно выбора параметров, см. рис. 5.15. Некоторые выходные величины могут носить информационный или отладочный характер. В этом случае можно не связывать их с параметрами модели, а сразу, с помощью **Мастера переменных**, выводить на графики с помощью закладки **Внешние библиотеки**.

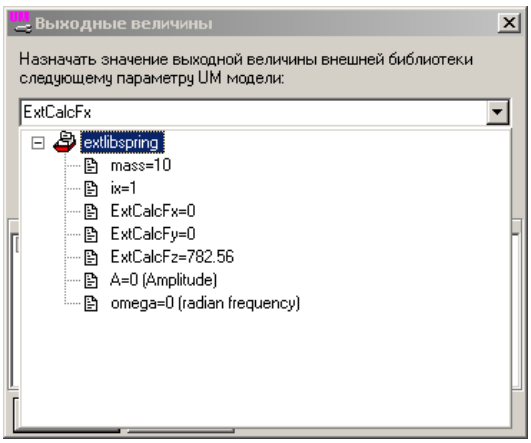


Рис. 5.15. Связь выходных параметров внешней библиотеки с параметрами UM модели

5.3.4.2. Скалярные силы типа *Библиотека (DLL)*

Расчет скалярных (биполярных и шарнирных) сил типа *Библиотека (DLL)* также происходит во внешних библиотеках. Формат и экспортируемые функции динамически загружаемой библиотеки в обоих случаях одинаковый. Вместе с тем, для библиотек, подключаемых через **Мастер связи с внешними библиотеками** и через скалярные силы типа *Библиотека (DLL)*, существует ряд отличий.

- Внешняя библиотека для сил типа *Библиотека (DLL)* должна иметь два входных сигнала (x, v) и один выходной. Для шарнирной силы x и v – значения координаты и ее производной по времени, для биполярного силового элемента – это длина элемента и его производная по времени. Кроме того, в процедуру **EXT_GetY**, кроме значений входных сигналов в любом случае передается текущее значение времени. Внешняя библиотека, предназначенная для подключения через **Мастер связи с внешними библиотеками** может иметь произвольное число входных и выходных сигналов.
- Подключение внешних библиотек для сил типа *Библиотека (DLL)* происходит в программе описания моделей **UM Input**, а внешних библиотек, предназначенных для подключения через **Мастер связи с внешними библиотеками** – в программе моделирования **UM Simulation**.
- Готовые для использования DLL для внешних библиотек сил типа *Библиотека (DLL)* должны располагаться в каталоге **{Данные УМ\lib\bfrc** (bfrc – от английского bipolar forces – биполярные силы). Внешние библиотеки, предназначенные для подключения

через **Мастер связи с внешними библиотеками** могут находиться по произвольному пути.

При прочих равных обстоятельствах, подключение сил как скалярных типа *Библиотека (DLL)* предпочтительнее с точки зрения быстродействия процесса моделирования, чем подключение тех же самых внешних библиотек через интерфейс **Мастера связи с внешними библиотеками**. Разумеется, речь идет только тех силах, которые по их физическому смыслу можно моделировать как шарнирные или биполярные. Иначе, в любом случае, необходимо использовать **Мастер связи с внешними библиотеками**.

Примеры:
{Данные УМ}\Samples\Library\DLL\BfrcSample.dpr;
{Данные УМ}\Samples\Library\DLL\BfrcSample1.dpr.

5.3.4.3. Особенности подключения нескольких библиотек

С помощью **Мастера связи с внешними библиотеками** одновременно можно подключить произвольное число внешних библиотек произвольное число раз. Например, если модель некоторой механической системы включает несколько одинаковых силовых элементов, описанных с помощью некоторой внешней библиотеки, то в **Мастере связи…** необходимо добавить эту внешнюю библиотеку столько раз, сколько одинаковых силовых элементов существует в механической системе.

Вместе с тем, при подключении одной и той же DLL через разные интерфейсы с помощью **Мастера связи…** или при использовании одной и той же DLL для описания сил типа *Библиотека (DLL)* следует обратить внимание на следующее обстоятельство. Область данных (глобальных переменных) DLL остается уникальной даже в случае загрузки нескольких копий одной и той же библиотеки. Таким образом, если процедуры, реализованные в библиотеке, изменяют значения глобальных переменных, а затем используют их в дальнейших вычислениях, то различные интерфейсы одной и той же библиотеки будут переписывать общую область данных, что в итоге потенциально приведет к некорректной работе внешней библиотеки.

В этих случаях использовать одну и ту же DLL можно только тогда, когда расчет выходных величин в процедуре **EXT_GetY** зависит только от параметров математической модели и текущих значений вектора входных значений или, другими словами, только если расчет выходных величин замкнут в процедуре **EXT_GetY** и не использует глобальных переменных. В иных случаях рекомендуется создавать (копировать) столько DLL, сколько однотипных внешних библиотек используется в модели.

Рассмотрим следующий пример. Пусть для использования в модели автомобиля разработана внешняя библиотека *library.dll*, которая описывает некоторый силовой элемент, который для расчета значений сил на следующем шаге сохраняет в глобальных переменных значения сил на предыдущем шаге. Допустим, что в модель автомобиля требуется добавить четыре таких элемента. В этом случае самое простое решение – скопировать библиотеку *library.dll* четыре раза, например, *library1.dll, library2.dll, library3.dll* и *library4.dll*, и затем в **Мастере связи…** или при описании скалярных сил использовать эти библиотеки.

5.3.5. Создание переменных для внешних библиотек

Обратите внимание, что с помощью закладки **Выражение** в **Мастере переменных** можно создавать переменные-константы, переменные-функции времени, а также переменные-функции параметров модели[5].

Так, например, на рис. 5.16 показано, как можно создать переменную-константу, а на рис. 5.17 и рис. 5.18 как создаются переменные-идентификаторы и функции времени.

С помощью **Мастера переменных** можно создавать *переменные*, отражающие входные и выходные величины, см. закладку **Внешние библиотеки**. Это позволяет оперативно сформировать необходимые величины для вывода в графические окна, например, при отладке внешней библиотеки или проведении исследований.

Подробнее о создании переменных для внешних библиотек с помощью **Мастера переменных** см. Главу 4 п. *Мастер переменных* настоящего руководства пользователя.

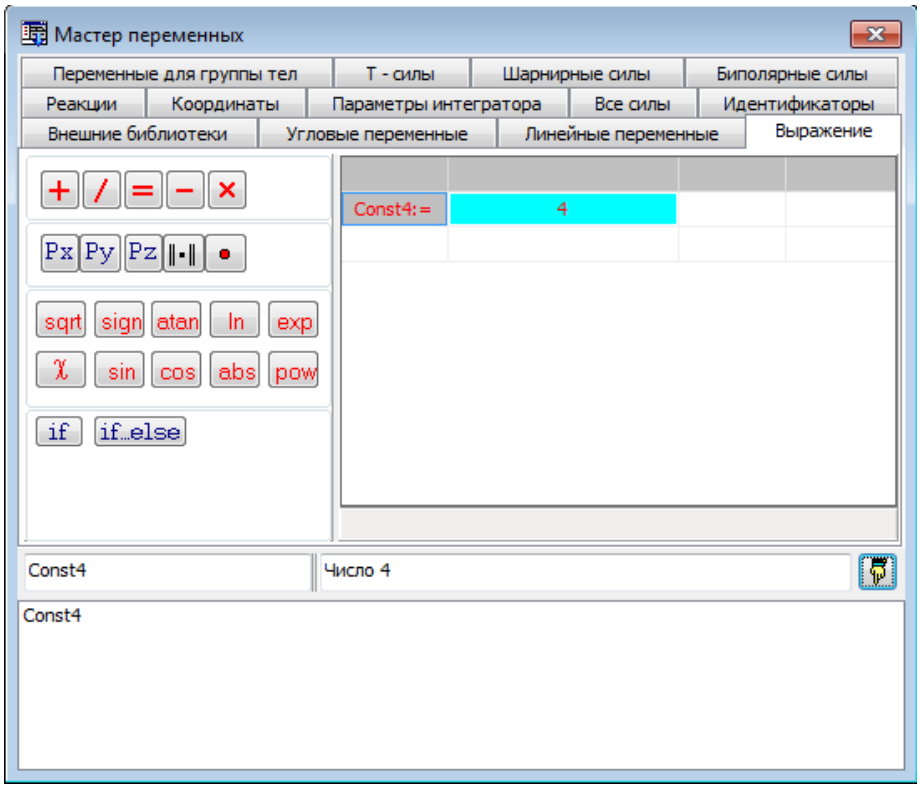


Рис. 5.16. Переменная-константа (число 4)

На рис. 5.17 показано, как с помощью **Мастера переменных** можно создать переменные для идентификаторов, которые затем будут использованы на закладке **Выражение** для описания параметризованных функций времени (рис. 5.18). При создании переменной A*sin(omega*t) используйте технологию Drag-and-Drop, чтобы вставить ранее созданные переменные A и omega в выражение.

[5] Данный раздел относится только для случая подключения библиотек через **Мастер связи с внешними библиотеками**

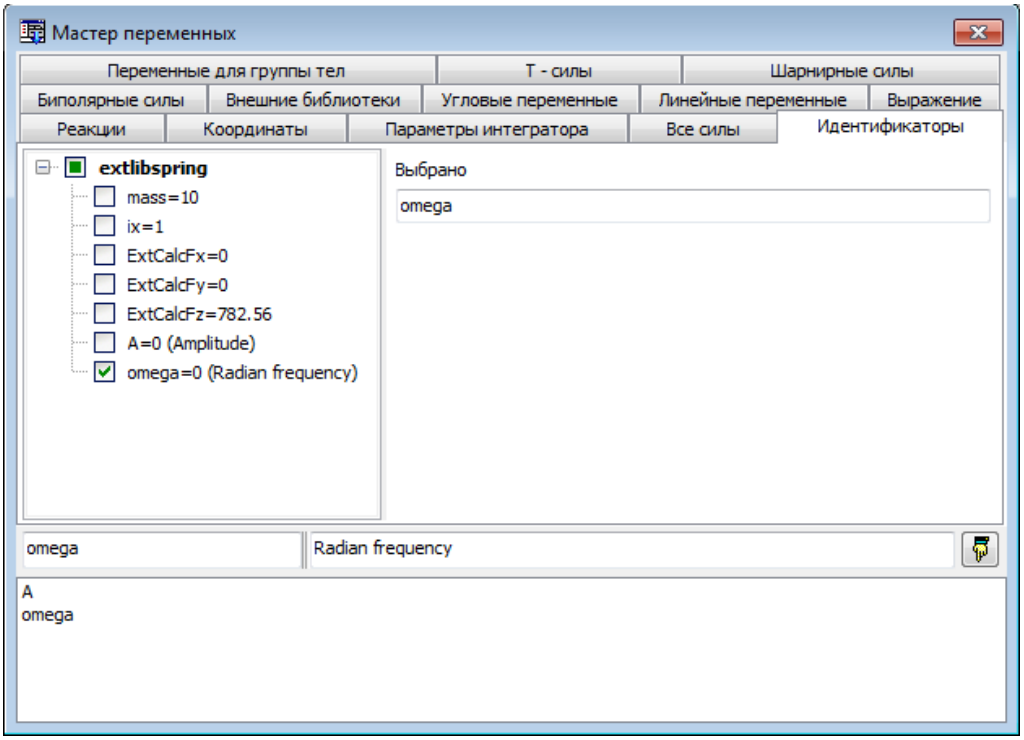


Рис. 5.17. Переменные-параметры (A и omega)

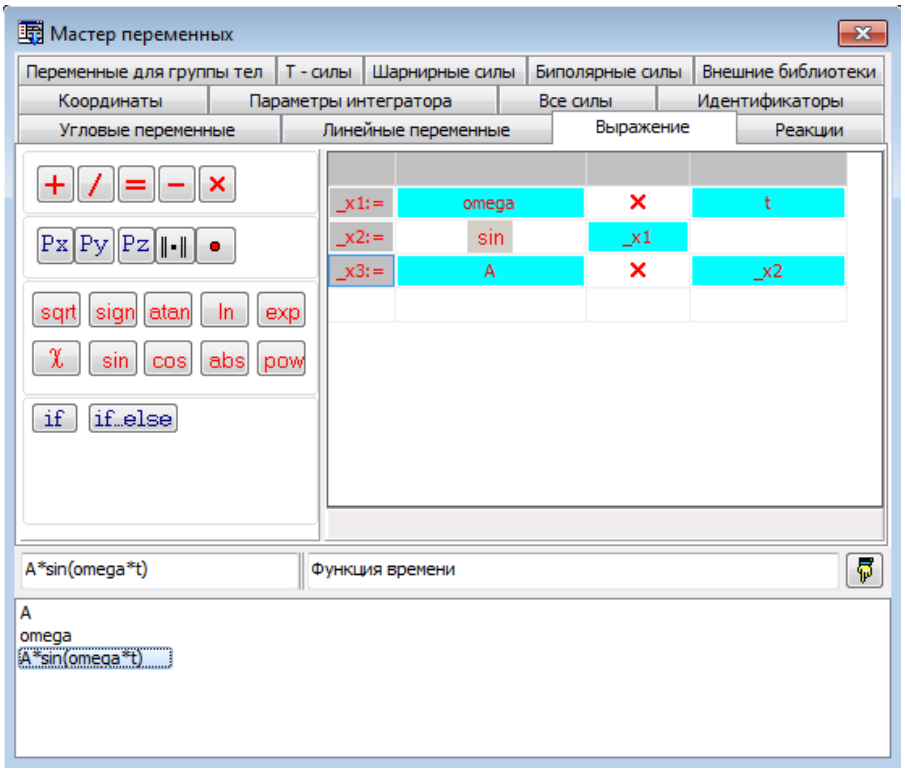


Рис. 5.18. Переменная A*sin(omega*t)

5.3.6. Каскадирование внешних библиотек

Каскадированием называется прием, когда выходные величины одной библиотеки назначаются на вход другой. Это позволяет создавать элементы, например, электрических

или гидравлических схем в виде отдельных внешних библиотек, а затем, соединяя их в определенном порядке, получать те или иные схемы.

При использовании каскадирования нужно помнить о том, что выходные сигналы подаются на вход других библиотек с запаздыванием по времени в один шаг численного метода. Кроме того, в начальный момент времени выходные сигналы внешние библиотек не определены и на вход других библиотек будут поданы нулевые значения. Таким образом, значения выходных величин, полученные на *i-ом* шаге численного метода, будут поданы на вход других внешних библиотек на *i+1* шаге.

5.3.7. Пример создания внешних библиотек

Рассмотрим пример создания внешней библиотеки, описывающей линейную пружину. Модель пружины, реализованная в этом примере, носит упрощенный характер. Рассматриваются только компоненты сил, моменты игнорируются.

Модель механической системы груза на пружине в каталоге {Данные УМ}\ samples\tutorial\extlibrary\extlibspring.

Примеры этих библиотек на C и Pascal даны в файлах в каталоге {Данные УМ}\samples\tutorial\extlibrary\source, см. *umlinearspring.dpr* и *umlinearspring.cpp*. Подробно останавливаться на особенностях реализации этих библиотек не будем – примеры очевидны для любого человека, знакомого с программированием.

Рассмотрим некоторые особенности подготовки этой модели. Запустите программу описания моделей **UM Input** и откройте модель из каталога {Данные УМ}\ samples\tutorial\extlibrary\extlibspring.

Во-первых, обратите внимание, что в модель введена биполярная сила **FictitiousSpring** (фиктивная пружина) с нулевыми параметрами и пружиной в качестве графического образа. Таким образом, мы просто визуализируем силу, которую на самом деле будем рассчитывать во внешней библиотеке.

Во-вторых, для приложения рассчитанных во внешней библиотеке сил в модель введена сила **SpringForce** типа **Т-сила**. Значения этой силы заданы параметрами ExtCalcFx, ExtCalcFy и ExtCalcFz. В дальнейшем при подключении внешней библиотеки мы свяжем ее выходные параметры – суть рассчитанные упругие силы – с этими параметрами.

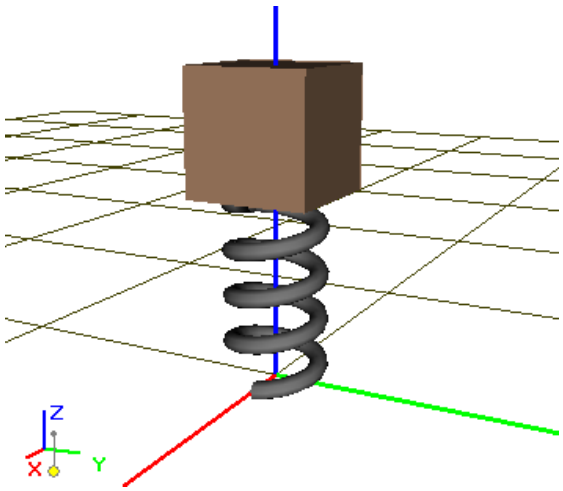


Рис. 5.19. Общий вид модели

Далее откройте эту модель в программе моделирования динамики **UM Simulation**. Окно **Мастера связи с внешними библиотеками** показано выше на рис. 5.14.

Обратите внимание, что на вход математической модели, реализованной во внешней библиотеке, нужно подавать деформации пружины по направлениям X, Y, Z. Для координат X и Y деформации пружины равны текущему положению центра масс тела в глобальной системе координат. Что касается деформации вдоль оси Z, то в этом направлении необходимо еще учесть длину нерастянутой пружины (0,3 м).

Необходимые переменные можно создать с помощью **Мастера переменных**, см. рис. 5.20 для деформаций по осям X и Y и рис. 5.21 для деформации вдоль оси Z.

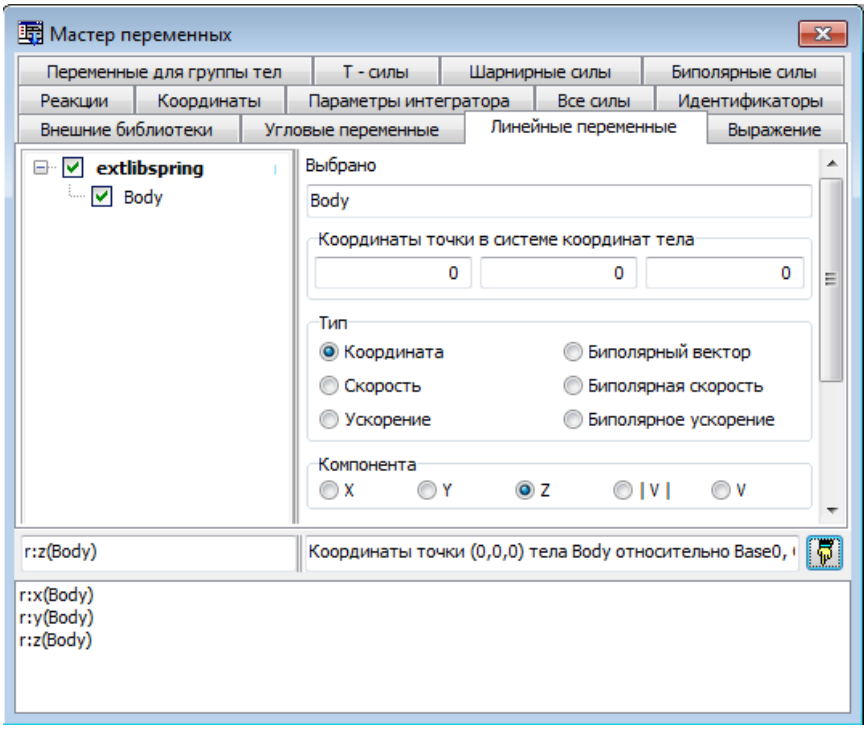


Рис. 5.20. Положение тела в проекции на ось X и Y

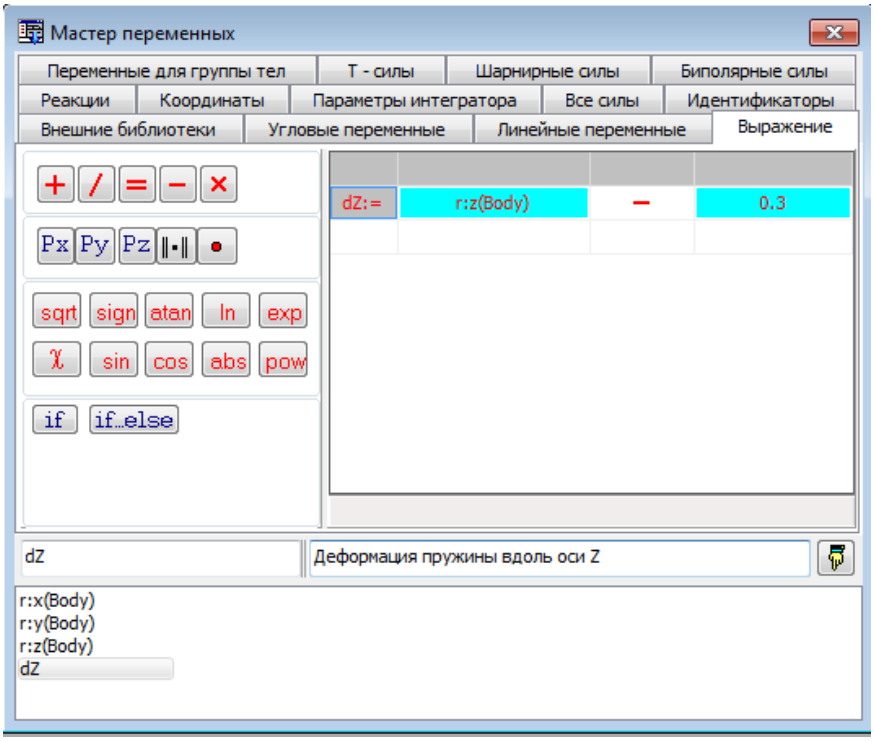


Рис. 5.21. Деформация пружины вдоль оси Z