Programming in UM Environment

User's manual

2014

Contents

5. Programming in UM environment

Universal Mechanism supports the following techniques to use user's defined procedures within UM models: (1) programming in the *Control File* (see Sect. 5.1) and programming in the *External libraries* (see Sect. 5.3).

In your applied models now it is recommended to use *External libraries* technique as more up-to-date and user- and programmer-friendly. However both mentioned techniques can be used within any one model.

5.1. Programming in the Control File

As a rule, modeling of complex technical systems requires programming by the user in the UM environment. Available programming languages are: *Pascal, Delphi, C* and *C++*. Programming is a powerful tool for calculation of complex forces, description of external functions as well as for control the simulation process. The user may use a number of predefined UM procedures. Programming is carried out with the help of the *control file*, which is generated as a part of equations of motion for the object and every external subsystem presented in the object.

Programming in the UM environment concerns to the *simulation module*.

The user is responsible for correctness of programming procedures and functions. An external debugger can be used for debugging the code.

User can link any number of units to the control file.

5.1.1. Standard constants, types and variables

Consider some constants, types and variables useful for programming in the UM environment.

5.1.1.1. Unit CtvSt.pas

- *Constants*

```
{Type of multiplication of a coordinate vector by a 3x3 matrix c=Ab or
c=A'b)}
NORMAL      = 0;
TRANSPON    = 1;
```

- *Types*

```
real_       = double;             { basic floating-point type }
coordin     = array [1..3] of real_; { coordinate vector }
trans_matr = array [1..3,1..3] of real_; { 3x3 – matrix, often a direct co-
sine matrix }

{Dynamic arrays}

VectReal    = array [1..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;

{One-dimensional arrays}

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;
{Two-dimensional arrays}

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

5.1.1.2. Unit CtvDll.pas

Constants

```
{Indices of UM messages}
  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;
```

{Maximal number of external subsystems}

```
  NSubsMax    = 1000;
```

{Type of object elements}

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

{Types of user's messages}

```
_mtConfirmation = 0;
_mtInformation  = 1;
_mtError        = 2;
```

{Types of contact forces}

```
_cftSum = 0;
_cftNormal = 1;
_cftFriction = 2;
```

{Index of system of coordinates for vector components}

```
BodyCoordinateSystem = 0;
BaseCoordinateSystem = 1;
```

- *Variables*

```
t           : real_; – current time value during simulation/XVA process;
NSubSystems : integer;  – number of external subsystems for the object;
SubIndx     : VectIPtr; - array of local indices of external subsystems;


UserVars    : array [0..1000] of real_; {elements of the array are available
for plotting in a graphical window}
```

- *Types of standard UM procedures, which can be used by the user*

5.1.2. Unit DGetVars.pas

The unit contains headers of standard UM procedures, which can be used by the user.

5.1.3. Control File Structure

The control file for an object has the name *cl[NameOfObject].pas*, where *[NameOfObject]* is the object name. The file is a part of equations. It is generated by the Input Module and located in the object directory. This file is the basis for programming in the UM environment.

Consider the structure of the control file for an object *pendulum* when external functions are not presented in the object description.

```
unit Clpendulum;

interface

uses CtvSt, CtvDll;

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

procedure ControlPanelMessage( _x, _v, _a : VectRPtr; _isubs, _index : inte-
ger; _Value : double ); 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, pendulumC, _Tpendulum;

procedure TimeFuncCalc( _t : real_; _x, _v : VectRPtr; _isubs : integer );
var
  _ : _pendulumVarPtr;
begin
  _ := _PzAll[SubIndx[_isubs]];
end;

procedure ForceFuncCalc( _t : real_; _x, _v : VectRPtr; _isubs : integer );
var
  _ : _pendulumVarPtr;
begin
  _ := _PzAll[SubIndx[_isubs]];
end;

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

end.
```

The interface part of the file contains headers of 6 procedures:

- **TimeFuncCalc**
  Calculation of all time-dependent function.
- **ForceFuncCalc**
  Calculation of user's forces.
- **UserCalc**
  Processing UM messages.
- **UserConCalc**
  Additional user-defined constraint equations.
- **EstimationFuncCalc**

Estimation of the criterion function for parametric optimization of object.

- **ControlPanelMessage**
  Processing messages from a *control panel*.

The first and the second procedures are called from the unit *Al[NameOfObject].pas*. Structure of this file is important for programming.

```
unit Alpendulum;

interface

uses CtvSt, CtvDll;

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

implementation

uses
  Clpendulum, _Tpendulum, pendulumC, pendulumE, DGetVars;

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

  //First call - kinematics
  if _kinemat then begin
    _._ap := CommonData.APredVector;
    try
      TimeFuncCalc( t, _x, _v, _isubs );
    except
      _isubs := -2;
      exit;
    end;
    //Evaluation of trigonometric functions
    _._s1 := sin( _x[1] );
    _._c1 := cos( _x[1] );
    DoElement( _x, _v, 1, _isubs );
  end;
  //Second call - generalized forces
  if CommonData.ForcesCalculation then begin
    _._ap := CommonData.APredVector;
    DoElement( _x, _v, 2, _isubs );
  end;
  //Third call - mass matrix
  if CommonData.MassMatrixCalculation then begin
    _lMassMatrix := CommonData.MMatrixPtr;
    DoElement( _x, _v, 3, _isubs );
    UserPtr1 := _lMassMatrix;
    UserCalc( _x, _v, nil, _isubs, CALCMASSMATRIX_MESSAGE, UserWhatDo );
  end;
end;

end.
```

Procedure **AllCalc** organizes evaluating elements of equations of motion. The simulation module calls this procedure three times for every iteration of the simulation process to obtain

- kinematical variables (positions, velocities);
- generalized applied and inertia forces;
- mass matrix.

Evaluation of the elements above is executed in the procedure **DoElement**, which is called three times.

Call of the procedure **TimeFuncCalc** is located before evaluation of kinematical variables, therefore some kinematical quantities can be calculated or initialized in the procedure **TimeFuncCalc**. This procedure cannot be used for computing forces, which depend on the current kinematical variables.

Do not forget to make backup copies of the control file!

5.1.4. Full names of elements

The full name of an object element (body, joint etc.) contains its own name as well as names of all subsystems (external and internal), which belong to the path from the subsystem of the element to the object root.

To get the whole list of element names for an object:

1. run Input module (**UM Input**);
2. open the object;
3. create the file of element with the help of the **Tools | File of elements…** menu command.

The file *n[NameOfObject].txt* appears automatically in the built-in text and stored in the object directory.

Example of the file of elements

```
Object : vehicle
********************List of bodies**********************
  Body
  Bogie1.Frame
  Bogie1.WMSet1.Motor
  Bogie1.WMSet1.Rotor
  Bogie1.WMSet1.GearRim
……..
********************List of joints**********************
  jCarBody
  Bogie1.jFrame
  Bogie1.WMSet1.jMotor
  Bogie1.WMSet1.jRotor
  Bogie1.WMSet1.jGearRim
……..
********************List of identifiers**********************
  v0
  xsh
  mc
  icx
  icy
  Bogie1.ixframe
  Bogie1.iyframe
  Bogie1.dz2
  Bogie1.fwnl2
……..
```

5.1.5. Indices of elements

The user utilizes names of elements (bodies, joints and so on) for their identification in the Input module. But internal identification of elements uses their *indices*. Indexing is used to organize access to body kinematical variables, to value of coordinates and other data, which are necessary for programming.

Indexing starts with 1.

All elements except external subsystems have two indices:

1. Index of an external subsystem *isubs* (1..NSubSystems) the element belongs to. If an object does not contain external subsystems, this index is always 1.
2. Local index of the element within the corresponding external subsystem (object).

Indexing is foreseen for the following elements:

- Subsystems
- Graphical objects
- Coordinates
- Bodies
- Joints
- Bipolar forces
- General forces
- Linear forces
- Contact forces
- Special forces
- Identifiers

The user can determine indices in two ways.

1. The **Indices** tab of the object constructor in the **UM Input** module.

An example is shown in Figure 5.1. The object contains 2 external subsystems. Elements of every type are presented as a two-level tree. The first level corresponds to the type of element, the second level – to elements (joints in Figure 5.1) within the corresponding indices. If the element belongs to the subsystem, its name begins with the name of the subsystem and point. For instance, the graphical object called **Rotor** belonging to the subsystem Electricmotor, has the name "Elektricmotor.Rotor" and the index 4. subsystem.

This method for defining element indices has an essential disadvantage. Deleting or adding of elements or subsystems can change indices. Therefore, the second method is more reliable.

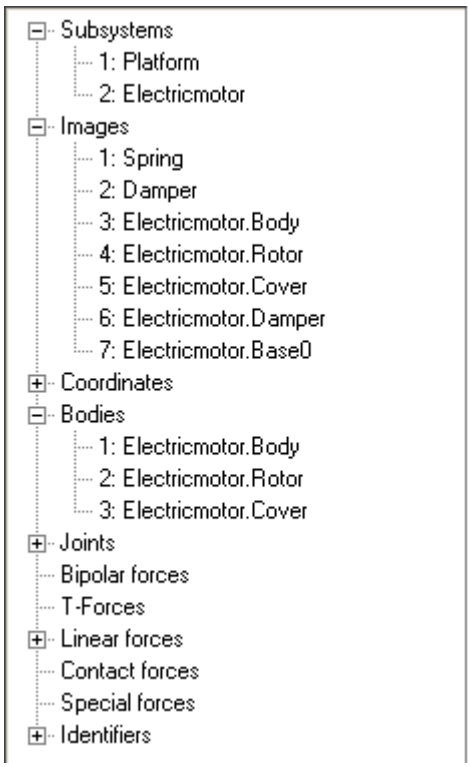


Figure 5.1.

Getting indices by element names.
The following function is used to get indices be full element name:

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

Input parameters:
elType – type of element from the following list:

Type	Comments
eltBody	Body
eltJoint	Joint
eltSubsystem	Subsystem
eltBFrc	Bipolar force
EltLFrc	Linear force element
EltCFrc	Contact force element
EltAFrc	General force
EltSFrc	Special force
EltGO	Graphical object
EltIdentifier	Identifier

Name is a full element name.

Output:

index : local index of element;

isubs : index of external subsystem;

return value : 0 (success), -1 element not found.

Instructions

Use the **UserCalc** procedure in the control file and the **FIRSTINIT_MESSAGE** to get all necessary indices. This message is sent by Simulation module directly after loading object.

Use global or static variables as identifiers of indices.

Example

Indices of two linear force elements are determined in the example.

```
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,'Bogie1.WMSet1.SpringLeft',
            LFrc1Indices[1,1], LFrc1Subs[1,1])=-1 then
        UMMessage(' Element not found: Bogie1.WMSet1.SpringLeft ', _mtError);
      If GetElementIndexByName(eltLFrc,'Bogie1.WMSet1.SpringRight',
            LFrc1Indices[2,1], LFrc1Subs[2,1])=-1 then
        UMMessage('Element not found: Bogie1.WMSet1.SpringRight ', _mtError);
    end;
  end;
end;
```

5.1.6. Procedures and functions

Access to UM procedures and functions is carried out with the help of the header unit **DGet-Vars**.

The input parameter *isubs* used below is the index of external subsystem.

5.1.6.1. Number of elements

To determine element count in object or external subsystem you should use the following function:

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

  Input parameters

  elType – type of element from the list

Type	Comments
eltBody	Body
eltJoint	Joint
eltSubsystem	Subsystem
eltBFrc	Bipolar force
eltLFrc	Linear force element
eltCFrc	Contact force element
eltAFrc	General force
eltSFrc	Special force
eltGO	Graphical object
EltIdentifier	Identifier

The function returns the number of elements of the indicated type or –1 if it is failed.

5.1.6.2. Values of coordinates

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

The function returns the value of the coordinate with index $nr \in [1..NCoordinates]$ where *NCoordinates* is the number of coordinates in external subsystem *isubs*.

- function **GetCoordDer**(nr, isubs : integer) : real_;

The function returns the value of the first derivative of coordinate with index $nr \in [1..NCoordinates]$ where *NCoordinates* is the number of coordinates in external subsystem *isubs*.

5.1.6.3. Kinematics of bodies

- procedure **GetPoint**(body, isubs : integer; ro : coordin; var r : coordin);

  Input: *body* – index of a body, *ro* – coordinates of a point in the body-fixed SC.

  Output: *r* – coordinates of the point in SC0.

- procedure **GetVel**(body, isubs : integer; ro : coordin; var v : coordin);
  Input: *body* – index of a body, *ro* – coordinates of a point in the body-fixed SC.
  Output: *v* – components of the velocity vector in SC0.

- procedure **GetAi0**(body,isubs : integer; var ai0 : Trans_Matr);
  Input: *body* – index of a body.
  Output: direct cosine matrix *ai0* for the body.

Remark. The matrix Ai0 converts vector components from SC0 to the body-fixed SC, i.e. $r^i = A_{i0} r^0$. The transposed matrix carries out the inverse transformation $r^0 = A_{0i} r^i = A_{i0}^T r^i$.

- procedure **GetVelAng**(body, isubs : integer; var om : coordin);
  Input: *body* – index of a body.
  Output: angular velocity *om* of the body in SC0.

A group of procedures determines relative position and motion of a pair of bodies. They have the following common input parameters:

bd1, bd2 – indices of bodies, motion of *bd2* relative to body *bd1* is computed;

isubs1, isubs2 – indices of external subsystems containing the bodies;

bdref, isubsref – indices of a reference body and the corresponding subsystem. Results are presented in the reference body-fixed SC. If *bdref* is 0, output vectors are resolved in SC0.

- procedure **GetRelVeloc**(bd1, isubs1, bd2, isubs2, bdref, isubsref : integer; const ro2 : coordin; var v12 : coordin);
  Output: velocity vector *v12* of point *ro2* of body *bd2* relative to body *bd1*. Vector *ro2* is resolved in SC of body *bd2*.

- procedure **GetRelAcc**(bd1, isubs1, bd2, isubs2, bdref, isubsref : integer; const ro2 : coordin; var a12 : coordin);
  Output: acceleration vector *a12* of point *ro2* of body *bd2* relative to body *bd1*. Vector *ro2* is resolved in SC of body *bd2*.
  The procedure can be used exclusively as processing the **STEPSINGLE_MESSAGE, STEPEND_MESSAGE** as well as **XVASTEP_MESSAGE** in the procedure **UserCalc**.

- procedure **GetRelAng**(bd1, isubs1, bd2, isubs2, bdref, isubsref : integer;
  var ang12 : coordin; var angle : real_);
  Output: rotation vector *ang12* as well as the value *angle* of the turning angle of body *bd2* relative to body *bd1*.

- procedure **GetRelVelAng**(bd1, isubs1, bd2, isubs2, bdref, isubsref : integer;
  var om12 : coordin);
  Output: vector of angular velocity *om12* of body *bd2* relative to body *bd1*.

- procedure **GetRelAccAng**(bd1, isubs1, bd2, isubs2, bdref, isubsref : integer; var e12 : coordin);
  Output: angular acceleration vector *e12* of body *bd2* relative to body *bd1*.

The procedure can be used *exclusively* as processing the **STEPSINGLE_MESSAGE, STEPEND_MESSAGE** as well as **XVASTEP_MESSAGE** in the procedure **UserCalc**.

- procedure **GetRelCoord**(bd1, isubs1, bd2, isubs2, bdref, isubsref : integer; const ro1, ro2 : coordin; var r12 : coordin);
  Output: vector *r12*, connecting two points of the bodies. The points are defined by vectors *ro1*, *ro2* in SC of the corresponding bodies.

- procedure **GetRelVelPoints**(bd1, isubs1, bd2, isubs2, bdref, isubsref : integer; const ro1, ro2 : coordin; var v12 : coordin; var v : real_);
  Output: vector of bipolar velocity *v12* as well as its value v. Consider two points of body *bd1* and body *bd2*, which coordinates are given by *ro1* and *ro2* in the body-fixed SC. Velocity v is the derivative of the distance between these points. Vector *v12* can be found from the formula $v_{12} = ve_{12}$ where e_{12} is the unit vector directed from the first to the second point, $\|v_{12}\| = |v|$.

- procedure **GetRelAccPoints**(bd1, isubs1, bd2, isubs2, bdref, isubsref : integer; const ro1, ro2 : coordin; var a12 : coordin; var a : real_);
  Output: vector of bipolar acceleration *a12* as well as its value a. Consider two points of body *bd1* and body *bd2*, which coordinates are given by *ro1* and *ro2* in the body-fixed SC. Acceleration *a* is the second derivative of the distance between these points. Vector v12 can be found from the formula $a_{12} = ae_{12}$ where e_{12} is the unit vector directed from the first to the second point, $\|a_{12}\| = |a|$.

The procedure can be used *exclusively* as processing the **STEPSINGLE_MESSAGE, STEPEND_MESSAGE** as well as **XVASTEP_MESSAGE** in the procedure **UserCalc**.

5.1.6.4. Operations with 3-vectors and 3x3-matrices

The following procedures and functions are useful for operations with kinematic vectors (coordinates, velocities, accelerations, the *coordin* type) and 3x3-matrices (the *trans_matr* type).

- procedure **Mult_vec_val**(var a : coordin; r : real_; var b : coordin);
  Multiplies a vector *a* by a scalar *c*, *b=ca*

- function **ScalarMult**(var a, b : coordin) : real_;
  The function returns the scalar product of two vectors.

- function **Vect_Len**(var a : coordin) : real_;
  The function returns the length of a vector a.

- function **Vect_Mult**(a, b : coordin; i : integer) : real_;
  The function returns the i-th component of the cross product $\mathbf{a}\times\mathbf{b}$.

- procedure **Mult_Vect**(a, b : coordin; var c : coordin);
  Output: cross product of two vectors **c=a×b**.

- procedure **Turning**(angle : real_; e : coordin; var a10 : trans_matr);
  The procedure realises the known formula of finite rotation. Let SC1 be the result of rotation of SC0 by the angle *phi* about the vector *e*. The procedure output is the direct cosine matrix A_{10}. If *e=0*, $A_{10}=I$ (the identity matrix).

- procedure **MkVectAng**(var angle : real_; var rvect : coordin; var a10 : trans_matr);
  This is the inverse procedure to the procedure **Turning**. Let the SC1 orientation be the result of turning the SC0 by angle *angle* around vector **e**. The procedure output is the vector *e* and the rotation angle calculated by the direct cosine matrix *a10*.

- procedure **GetRotAngles**(const A: trans_matr; i, j, k: integer; var ai, aj, ak: real_);
  Input: direct cosine matrix A, sequence of rotations $i, j, k \in [1,2,3]$.
  Output: Three orientation angles *ai*, *aj*, *ak*: $A = A_i(ai)A_j(aj)A_k(ak)$.

- procedure **Mult_m_v_3x1**(type_ :byte; var a: trans_matr; b: coordin; var c: coordin);
  Input: matrix *A*, vector *b* and type of multiplication *type_*.
  Output: $c = \begin{cases} Ab, & \text{type\_} = \text{NORMAL} \\ A^T b, & \text{type\_} = \text{TRANSPON} \end{cases}$

Remark. The procedure is usually used to resolve vectors in different SC.

- procedure **Mult_m_m**(a,b : trans_matr; var c : trans_matr);
  Output: $C = AB$ (the product of two 3x3 matrices).

- procedure **Mult_mT_m**(a,b : trans_matr; var c : trans_matr);
  Output: $C = A^T B$ (the product of two 3x3 matrices).

- procedure **Mult_m_mT**(a,b : trans_matr; var c : trans_matr);
  Output: $C = AB^T$ (the product of two 3x3 matrices).

- procedure **Mult_mT_mT**(a,b : trans_matr; var c : trans_matr);
  Output: $C = A^T B^T$ (the product of two 3x3 matrices).

5.1.6.5. Solving linear algebraic equations

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

The Gauss elimination procedure for solving the linear algebraic matrix equation *AX=B* based on the column pivoting.

Input:

a is a square $na_x na$ matrix. It is not conserved;

b is a $na_x nb$ matrix of the right-hand side. The matrix is not conserved.

Eps is a positive number determining matrix singularity. The *eps=1.0e-12* value is recommended.

Output:

b is the equation solution;

code equals 1 if the solution is obtained, 0 if the matrix A is singular.

The *MatrRPtr* type corresponds to a dynamic matrix.

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

The Gauss elimination procedure for solving the linear algebraic equation *Ax=b* with non-singular matrix A. Pivot elements are looked for columns.

Input:

a - matrix A; type - MatrRPtr; it is not conserved;

b – one-dimensional vector of the right-hand side (VectRPtr);

na- size of the matrix A equal *na*na*;

Eps is a positive number determining matrix singularity. The *eps=1.0e-12* value is recommended.

Output:

b is the equation solution;

code equals 1 if the solution is obtained, 0 if the matrix A is singular.

5.1.6.6. Additional forces and moments

The user can compute additional forces and moments, *which are not included in the object description in the Input Module*. The additional forces can be calculated either in the **ForceFuncCalc** procedure (Sect. 5.1.3) or by processing the **FORCESCALC_MESSAGE** in the **UserCalc** procedure.

Procedures below are used for adding the additional forces to the object. The *CoordSystem* in the procedures detects a SC of the force or/and moment. Values of this parameter are:

BaseCoordinateSystem – SC0;

BodyCoordinateSystem – body-fixed SC (for the second body in **AddForceToBodyAtPoint**).

For all cases except the **AddForceToBodyAtPoint** procedure, the forces should be applied to the origin of the body-fixed SC.

Functions return 0 if succeed else –1.

- function **AddForceToBody**(ibody, isubs : integer; Force : coordin; CoordSystem : integer) : integer;
  Adds the force *Force* to body (*ibody*, *isubs*).

- function **AddMomentToBody**(ibody,isubs : integer; Moment : coordin; CoordSystem : integer) : integer;
  Adds the moment *Moment* to body (*ibody*, *isubs*).

- function **AddForceToBodyAtPoint**(ibody1,isubs1, ibody2, isubs2: integer; Point, Force, Moment : coordin; CoordSystem : integer) : integer;
  Adds a force and a moment to a pair of interacting bodies (ibody1, isubs1) and (ibody2, isubs2). The *Force* and the *Moment* act on the second body (ibody2, isubs2) at the point *Point* (coordinates are given in SC of the second body).

Example

Add the applied external force (0, 1000, 0), resolved in SC0, to body (1, 1) applied to the point (0, 0, 0.5) in SC1.

```
procedure ForceFuncCalc( _t : real_; _x, _v : VectRPtr; _isubs : integer );
const
  Point  : coordin = (0, 0, 0.5);
  Force  : coordin = (0, 1000, 0);
  Moment : coordin = (0, 0, 0);
begin
  AddForceToBodyAtPoint(0, 1, 1, 1, Point, Force, Moment, BaseCoordinateSys-
tem);
end;
```

5.1.6.7. Changing identifiers

UM allows parameterization of the object description. Changing parameters gives a powerful tool for analysis of object dynamics.

The user often changes identifiers directly before starting the integration process with the help of UM interface abilities. Another way for changing is programming identifiers in the control file.

From the other hand, changing some identifiers during the simulation process can lead to wrong simulation results because it does not correspond to laws of mechanics. Here are these parameters:

- inertia parameters: masses and inertia moments;
- joint geometrical coordinates;
- identifiers, which parameterize a graphical object, if GO are used for automatic computing the inertia parameters.

It is necessary to restart the simulation process after changing these identifiers.

5.1.6.7.1. Structures of identifiers

Identifiers introduced by the user for object parameterization are included in structures (‘records’ in the Pascal notation) in the object equations. These structures are not the same for differ-

ent external subsystems. Types of the structures can be found in the file _T[NameOfObject] generated by the Input Module as a part of equations and located in the directory of the object or external subsystem.

A structure is built strictly according to the tree of *included subsystems*. The identifying names of subsystems are used as names of substructures (subrecords), which include identifiers of the corresponding included subsystem.

Example. The object **Vehicle** contains two included subsystems with identifying names WMBlock1 and WMBlock2. Each of these subsystems contains one included subsystem with identifying name Wset. In this case the object identifiers are presented by the following structure:

```
_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;
```

The user has access to the identifiers with the help of variables

_PzAll, _PzAll[NameOfObject]

which are declared in the unit [NameOfObject]C.pas. These variables are pointers on one and the same array of structures. The length of the array of structures is always 1 for the root object, whereas it is equal to number of kinematically identical external subsystems included in the object and having the same ancestor.

For instance, the object *Train* contains 11 external subsystems. One of the subsystems has the object *Locomotive* as the ancestor, the other ten subsystems – the object *Car*. In this case the length of the array is 1 for the subsystem with the ancestor *Locomotive*, the structure of identifiers can be found in the unit …\Locomotive_TLocomotive.pas, the variables _PzAll, _PzAllLocomotive are declared in the unit …\Locomotive\LocomotiveC.pas. Identifiers for all other subsystems with the ancestor *Car* are included in the arrays _PzAll and _PzAllCar with length 10. These variables are declared in the unit …\Car\CarC.pas, the type of the corresponding structure is declared in the unit …\Car_TCar.pas.

Access to the identifiers has the following syntax:

_PzAll[NameOfObject].[Element of structure of identifiers]

or
_PzAll[SubIndx[isubs]].[Element of structure of identifiers]
Examples:
_PzAllVehicle[1].xsh
_PzAll[1].WMBlock1.xmotor
_PzAll[1].WMBlock2.Wset.mw

Access to identifiers is more simple if the object does not contain any subsystem, neither external nor included. In this case the array of structures contains one element and the structure of identifiers does not contain substructures. Examples:
_PzAll[1].mass
_PzAll[1].ix
_PzAll[1].something
_PzAllMyObject[1].some_identifier

5.1.6.7.2. Standard procedures for changing identifiers

Identifier values can be changed in two ways depending on type of object elements, which description is parameterized.

The first way consists in changes identifiers *without modification their values in the kernel of the simulation module*. This method is used if the evaluation of element is concentrated exclusively in the DLL of the object. Here is the list of such elements:

- Geometric and kinematic characteristics (there exist exceptions)
- Generalized linear force elements (*not external*)
- T-forces

A new value can be simply set to the identifier in the corresponding procedure of the control file, for instance
_PzAllMyObject[1].some_identifier:=0.1

The second method changes identifier both in the DLL and in the simulation module kernel. This way is necessary if the evaluation of the corresponding element (e.g. a force) is executed in the kernel. The user should send the new value to the kernel. Elements of this type are

- Inertia parameters;
- Identifiers parameterizing graphical objects;
- Identifiers parameterizing bipolar, external linear, contact force element, special force elements. Generalized linear force elements (not external)
- T-forces

The following procedure send a new identifier value to the kernel:
procedure **SetIdentifierValue**(index,isubs : integer; value : real_)

where index, isubs are identifier indices, real_ - new value.

After that the procedure refreshing the element parameter should be called (except of T-forces and direction of gravity)

function **RefreshElement**(elType,index, isubs : integer) : integer
where *elType* is the type of element (Sect.5.1.1.2), index, isubs are indices of the element.

5.1.6.7.3. Programming T-forces

Forces of general type are used for programming of complex T-force models. (Sect. 2.4.8, 3.3.10.4). Description of these forces allows parameterizing both force and moment components and coordinates of application point. The corresponding identifiers can be arbitrary changed during the simulation process either in the procedure **ForceFuncCalc**, which is called by the **FORCESCALC_MESSAGE** in the procedure **UserCalc**.

Example:

```
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));
  //Harmonic excitation along X
end;

procedure UserCalc;
var i : integer;
begin
  WhatDo:=NOTHING;
  case CurrEvent of
    FIRSTINIT_MESSAGE : begin
        GetElementIndexByName(eltIdentifier, 'fc1x', index_frc1x, i);
   //Store the identifier index in a global variable
    end;
    FORCESCALC_MESSAGE : begin
      try
        ForceFuncCalc( t, x, v, isubs );
      except
        WhatDo := -1;
      end;
    end;
  end;
end;
```

Remark. Programming forces of general type is an old approach. Use additional forces instead (Sect. 5.1.6.6).

5.1.6.7.4. Change of identifiers parameterizing graphical objects

Sometimes it is necessary to change graphical objects depending on time. As a rule, these are GO assigned to the scene. Usage of Z-surfaces gives opportunity to do it. Another way consists in change of identifiers parameterizing graphic elements in processing the **STEPEND_MESSAGE** and **XVASTEP_MESSAGE**.

To change identifier values use the procedure **SetIdentifierValue**. After that refresh the GO with the halp of the procedure **RefreshElement**.

SetIdentifierValue(indexIdent,isubsGO; value);

RefreshElement(eltGO,indexGO, isubsGO);

Example. It is necessary to change the position of a GO assigned to the scene image. The name of the GO is *Scene*. The GO position (e.g. its coordinate X relative to SC0) is set by the identifier *xscene*. The object does not contain subsystems.

```
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; //Store the old value
      GetElementIndexByName(eltIdentifier,'xscene',IndexIdent, i]);
      If GetElementIndexByName(eltGO,'Scene',IndexGO, i])=-1 then
        UMMessage('Element scene not found, _mtError);
    end;
  STEPEND_MESSAGE : begin
 //Setting the new value and refresh GO
      PzAll[1].xscene:=0.2*sin(2*t);
      SetIdentifierValue(IndexIdent, 1, PzAll[1].xscene);
      RefreshElement(eltGO, IndexGO, 1)
    end;
  INTEGREND_MESSAGE : begin //Set back the old value
      PzAll[1].xscene:= OldIdentValue;
      SetIdentifierValue(IndexIdent, 1, PzAll[1].xscene);
      RefreshElement(eltGO, IndexGO, 1)
    end;
  end;
```

GO will oscillates along the X axis.

5.1.6.8. Animation of user's vectors

The used might create a list of vectors, which values are computed in the control file. This list is created for animation of nonstandard vectors, which cannot be obtained with the help of the wizard of variables.

The function

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

creates and adds a new vector to the list, and returns the reference index of the vector. The index is used for assignment new values to the vector during the simulation.

Input: *Name* – name of the vector; *vtype* – type of the vector. The type is one of the following list (Sect. 5.1.1.2):

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

The type should be convert to integer, for example *ord(vtVelocity)*.

Call of this function is allowed in the procedure **UserCalc** for processing the **FIRSTINIT_MESSAGE**.

To change the vector value use the procedure

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

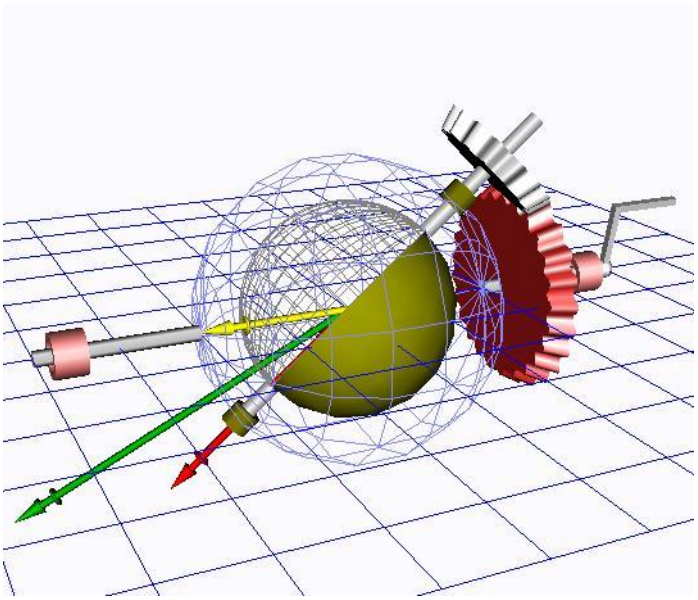
where *index* is the vector index, , *Value, Point* – their values and application points in SC0. The function returns 0 if succeed.

The function should be included in processing the **STEPEND_MESSAGE** and **XVASTEP_MESSAGE** in the procedure **UserCalc**.

Access to the user vectors is carried out in the **User | Vectors** tab of the wizard of variables.

Example.

Consider a model of a spherical crusher (the object Crusher in the Tutorial directory). This model illustrates addition of rotations of a body. User's vectors are used to animate vectors of absolute and relative angular velocities of the central sphere (not, that all these vectors can be obtained directly with the help of the wizard of variables).



Consider the corresponding control file.

The following global variables are introduced:

var avIndex : integer; //index of absolute angular velocity vector
rvIndex : integer; // index of relative angular velocity vector
evIndex : integer; // index of transient angular velocity vector
BodyIndex : integer; //index of the crusher body
CrusherIndex : integer; // index of the central sphere body

The procedure **UserCalc** looks like this:

```
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; //Absolute angular velocity
  ome : coordin; // Transient angular velocity
  omr : coordin;  //Relative angular velocity
  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
//Adding user's vectors
        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
//Calculate vectors
        GetVelAng(CrusherIndex,1,oma);
        GetVelAng(BodyIndex,1,ome);
        for i:=1 to 3 do omr[i]:=oma[i]-ome[i];
//Send vector values and positions of origins
        SetVectorValue(avIndex,oma,ro0);
        SetVectorValue(rvIndex,omr,ro0);
        SetVectorValue(evIndex,ome,ro0);
      end;
  end;
end;
```

5.1.7. Programming external function

External functions are used for description of the following elements:

- joint coordinates – time functions (rotational, translational joints, joints of generalized type, Sect.3.3.9);
- joint force and torque (joint of generalized type, Sect. 3.3.9.4);
- bipolar force element (п.3.3.10.1);
- contact force elements of types Z-sphere, Z-circle;
- special force element of the type ComboFriction;
- graphical elements Z-surface.

For all the above cases, the user enters identifiers of external functions in the Input module. Calculation of functions must be programmed in the control file.

Headers of the corresponding functions are generated automatically in the control file. The default values are zeroes.

If you modify an object, which equations of motion has already been generated before, and add, delete or rename one or several external functions, you should correct the old control file. Use the new variant of the control file (Cl[NameOfObject].*new*) to find added headers, if the option *Rewrite control file* is switched off by the generation of equations. If the option *Rewrite control file* is switched on (not recommended!), the old file is renamed to Cl[NameOfObject].*new*. Transfer old code from this file to the new version.

5.1.7.1. Programming of coordinates – time functions

Consider a slider-crank mechanism (Figure 5.2, the object *CrankRod* is located in the directory Tutorial). The external function with the name *phi* is introduced for dependence of the crank rotation as a time function.

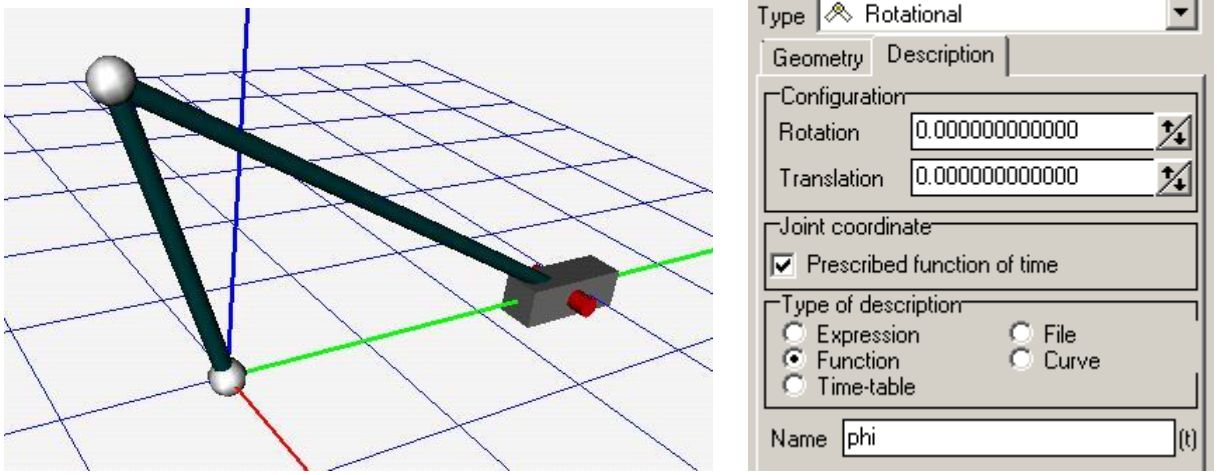


Figure 5.2.

Generator of equation of motion inserts the following procedure in the control file:

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

The user should write here a code for calculation of the angle (the variable *_Value*), its first and second time derivatives *(_dValue, _ddValue)*. In addition, the unit *AlCrankRod* contains the call of the procedure phi directly after the procedure **TimeFuncCalc**:

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

Object identifiers are very useful for programming external functions. Four additional identifiers are introduced by description of the mechanism in the Input module:

phi0, ampl, om, mode

These identifiers do not parameterized any object element. They are intended for programming of the function **phi** exclusively. The parameters can be changed either before on in process of the simulation.

Consider the following version of the procedure code:

```
procedure phi(_isubs : integer; _t : real_; var _Value, _dValue, _ddValue :
real_);
var  _ : _crankrodVarPtr;
begin
  _ := _PzAll[SubIndx[_isubs]];
  case round(_.mode) of
  0 : begin  //Uniform rotation
      _Value:=_.om*_t+_.phi0;
      _dValue:=_.om;
      _ddValue:=0;
```

```
    end;
  1 : begin  //Harmonic oscillations
      _Value:=_.ampl*sin(_.om*_t)+_.phi0;
      _dValue:=_.ampl*_.om*cos(_.om*_t);
      _ddValue:=-_.ampl*_.om*_.om*sin(_.om*_t);
    end;
  end;
end;
```

Note, that the variable '_' is for organization of success to the object identifiers (_.om corresponds to the identifier *om*). The same procedure can be rewrite in a different manner:

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

The identifier *mode* in the procedure is used as a switch, which turns on/off two modes of the crank prescribed motion: *mode*=0 the uniform rotation; *mode*=1 the harmonic oscillations. The identifier om set an angular velocity in the first mode and on oscillation frequency – in the second. The identifier *ampl* is ignored in the first mode and sets an oscillation amplitude in the second one. The identifier *phi0* sets an initial angle.

Desirable values of the identifiers can be set before each start of the simulation. Same simulation results are given in Figure 5.3 and Figure 5.4.

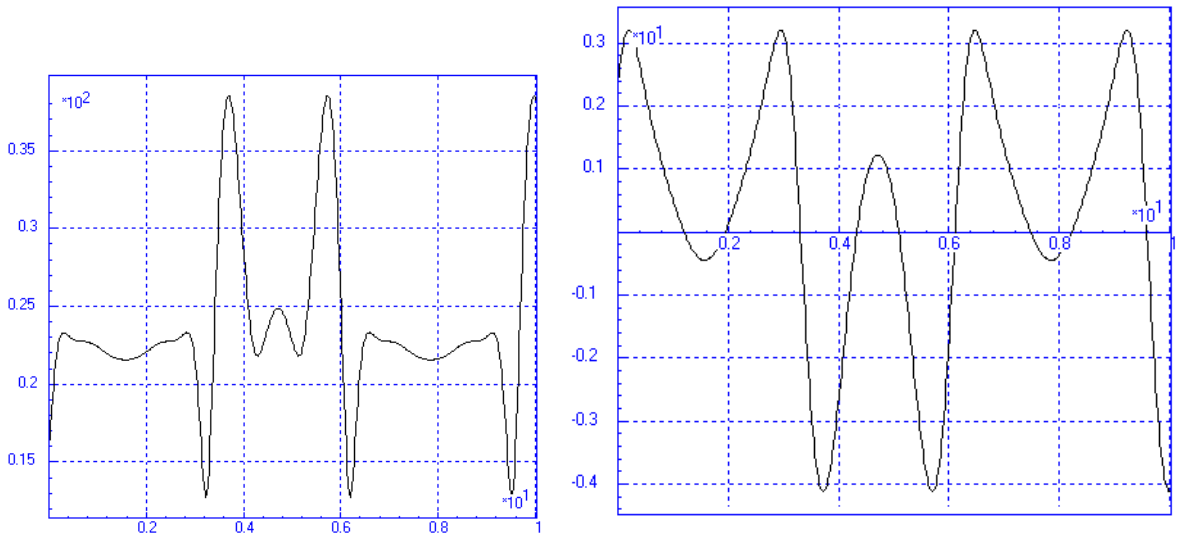


Figure 5.3. Module of reaction force in the crank-Base (left) and the slider acceleration for *mode*=1, *om*=1, *ampl*=2, *phi0*=0.

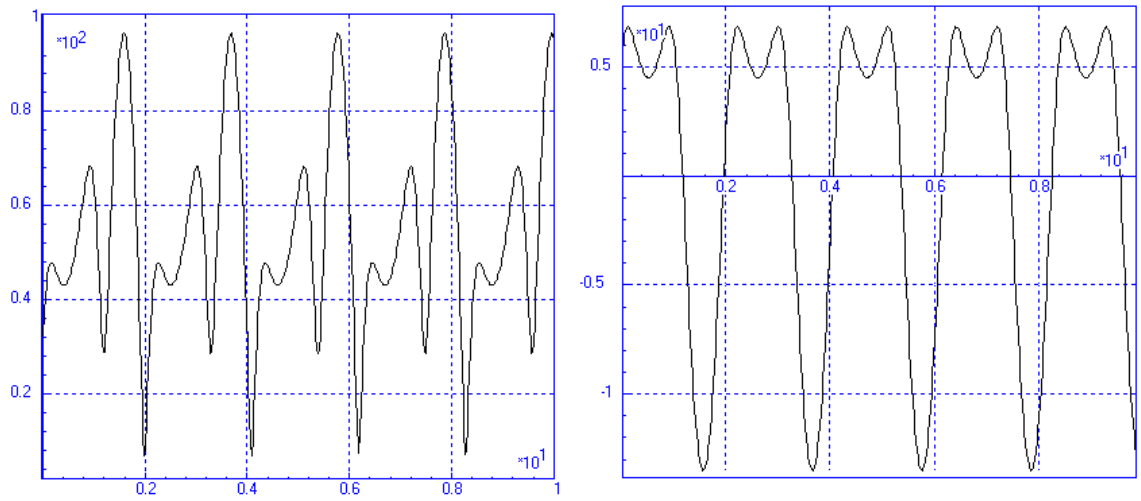


Figure 5.4. Module of reaction force in the crank-Base (left) and the slider acceleration for *mode*=0, *om*=3, *phi0*=0.

Remark. Be careful while programming the first and the second time derivatives. Errors lead to completely wrong simulation results.

5.1.7.2. Programming joint and bipolar forces

Programming of external bipolar functions in a *Control File* is not supported since UM 5.0.

5.1.7.3. Programming graphical elements: Z-surfaces

In this section we consider usage of Z-surfaces for creation of dynamic time-dependent images (Figure 5.5, the object *Float* in the directory *Tutorial*). The graphical element here is used for animation of waves. The surface is described by a graphic element Z-surface with the name *Waves*. The graphical object containing this element is assigned to the scene image. Note, that the surface is not fully described in the Input module, therefore it is seen there as a filled rectangle with the corresponding sizes (3×10 m).

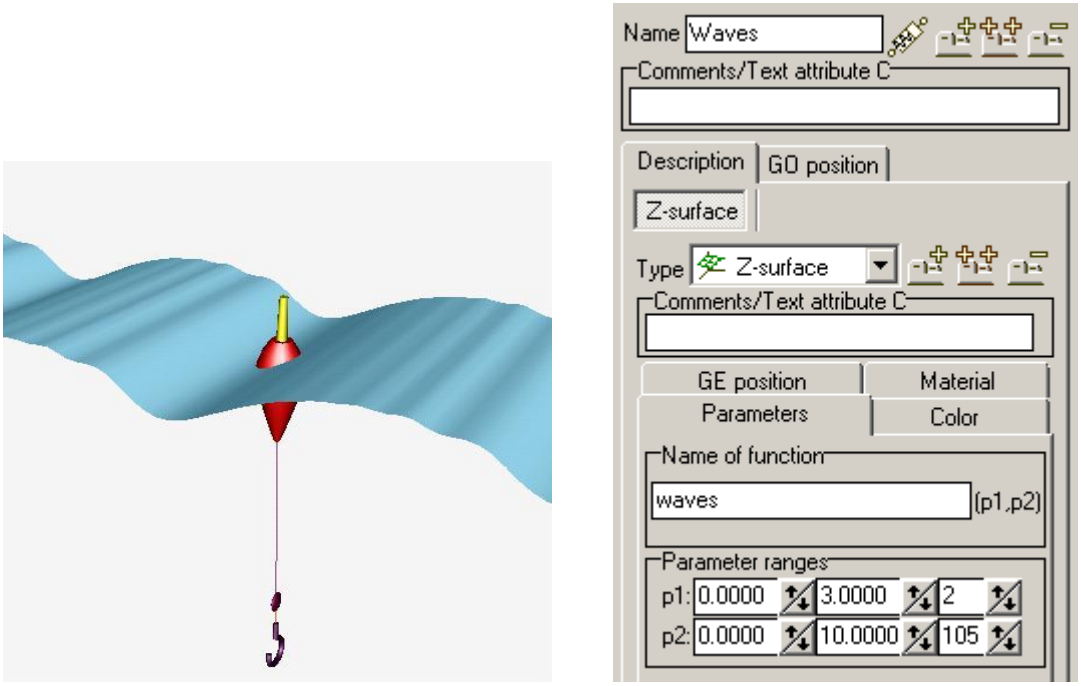


Figure 5.5.

After generation of equations the control file includes reference for the external function in the standard function **ZgraphicElementFunctions**. This function must include description of all external Z-functions. The function **ZgraphicElementFunctions** is called by the simulation module every time, when the graphic element is refreshed (by loading the object, by changing the object parameters in the corresponding window, after calling the user's procedure RefreshElement(eltGO,…)).

```
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;
```

Note that the name of the external function waves is located as a comment in the case statement of the function **ZgraphicElementFunctions**

```
{ Function waves }
```

Directly here the function should be calculated. The default value if the function is zero, and the element looks like a filled rectangle.

The user organizes calculation of external functions in the function **ZGraphicElementFunctions** in dependence on two Cartesian coordinates (parameters *p1, p2*). In the considered example, calculation of the wave image is done in the separate function **Wave**. The surface depends

from the y-coordinate (*_p2*) and time. The function is a sum of 10 trigonometric functions with various amplitudes, frequencies and time-dependent phases.

```
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;
```

The function output is the z-coordinate of the surface (returned value) and its derivative w.r.t. the y-coordinate (*deriv*). Of course, the derivative is not used by graphic image but it is necessary for computing hydrodynamic forces acting on the float.

The function **ZgraphicElementFunctions** calls the function **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;
```

The function **Wave** depends of time, but the dependence does not affect the image if the GO is not refreshed before drawing. Use the **StepEnd_Message** in the procedure **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;
```

Calling the refresh at the message **IntegrEnd_Message** sets the initial image state (t=0) after the end of simulation.

Remarks.

1. It is necessary to process the messages **XVASTEP_MESSAGE** and **XVAEND_MESSAGE** in the same manner as the messages **IntegrEnd_Message,**

StepEnd_Message to obtain the same wave animation for the XVA-analysis as for the simulation:
XVAEnd_Message,
XVAEnd_Message : RefreshElement(eltGO,1,1);

2. If an external function is modified after creation of an XVA-file, the animation during the XVA-analysis is not match to the XVA-file.

Consider the float-wave interaction. We programmed additional forces, which values depend on the wave motion and the float position relative to the wave. Note, that we used highly simplified mathematical model of the hydrodynamic forces.

The hydrodynamic forces are computed in the procedure **ForceFuncCalc**.

```
procedure ForceFuncCalc( _t : real_; _x, _v : VectRPtr; _isubs : integer );
const ro : coordin = (0,0,0);
var hWave, dWave : double; //Wave height in the float position and its deriv-
ative
                           //
    r,v,om : coordin;
    frc, trq : coordin;
begin
  GetPoint(1,1,ro,r);   //float coordinates in SC0
  GetVel(1,1,ro,v);     //float velocity in SC0
  GetVelAng(1,1,om);    //float angular velocity in SC0
  hWave:=Wave(t,r[2]+5,dWave); // Wave height in the float position and its
                               // derivative
  frc[1]:=0; frc[2]:=0;
  frc[3]:=(hWave-r[3]+0.3)*600-100*v[3];  //Lifting force and dissipation
  trq[1]:=dWave*50-om[1]*10;              // Moment acting on the float

  trq[2]:=0;  trq[3]:=0;
//Add forces
  AddForceToBody(1,1,frc,BaseCoordinateSystem);
  AddMomentToBody(1,1,trq,BaseCoordinateSystem);
end;
```

5.1.7.4. Programming contact surfaces for contact elements

Contact force elements Z-sphere allow the user to create objects with complex contact interaction, especially with the help of programming external functions. Simultaneous description of Z-surface graphic elements is usually used in this case for visualization of contact surfaces. In this section we consider an example of a controlled wheel robot (Figure 5.6, the object *Robot* in the directory *Tutorial*).

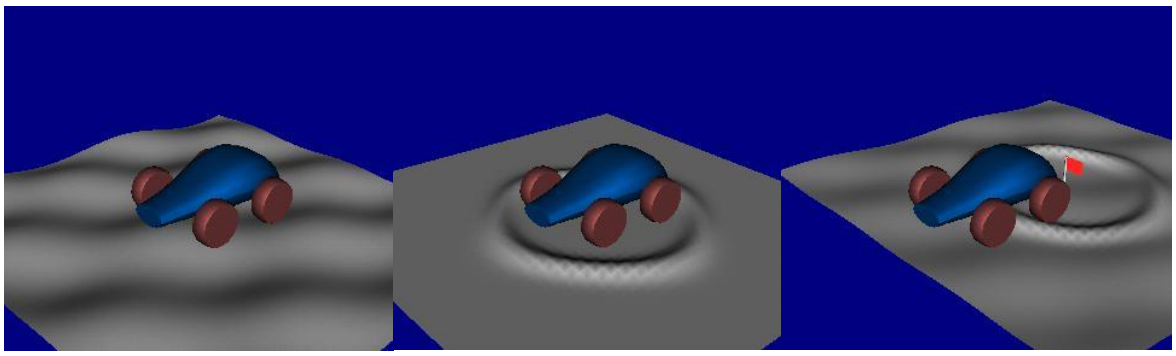


Figure 5.6.

We will discuss the following problems:

- How Z-surfaces for graphic objects and for contact surfaces can be programmed;
- How the image can follow the object motion.

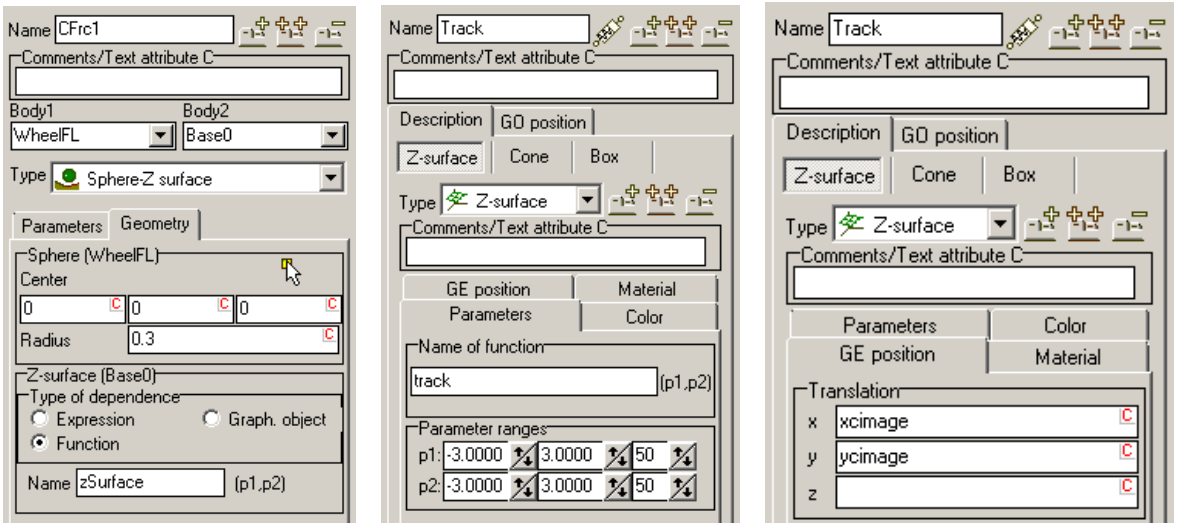


Figure 5.7.

For contact force elements of the type Z-sphere are introduced for contacts of the robot wheels with the ground. The ground contact surface is set by the external function *zSurface* for all contact elements (Figure 5.7, left). The scene image is defined in particular by an graphical element of the type Z-surface with the tame *Track* (Figure 5.7, center). Two identifiers *xcimage=0*, *ycimage=0* set the element position relative to SC0 (Figure 5.7, right).

The external Zsurface function, which describe the contact, is presented in the control file by the following function:

```
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;
```

Input parameters are: the Cartesian coordinates *_x*, *_y*

Output:

_z –function value;

_dzx – derivative w.r.t the x - coordinate;

_dzy – derivative w.r.t the y - coordinate;

Description of the surface image is defined by the external function Track, which should be computed in the procedure **ZGraphicElementFunctions**:

```
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;
```

We wrote the code in the control file, which supports

- tree mode for the ground surface image, the mode parameter is *SurfaceType*;
- control the robot motion with the help of the keyboard;
- game simulation for *SurfaceType = 1,2*.

Here is the code of the control file.

```
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 mode : integer; (*current mode of the robot*)
     xcIndex, ycIndex : integer;
     StopCount : integer;
     hT : double;

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 _PzAll[1]^ do
      Result:=TorqMax*tindex[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]);
  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;
         mode:=AHEAD;
         StopCount:=0;
       end;
   INTEGR_PROCESS : if (t>20) or (_PzAll[1].SurfaceType=0) then
       case key of
       VK_UP    : mode:=AHEAD;
       VK_DOWN  : mode:=BACK;
       VK_LEFT  : mode:=LEFT;
       VK_RIGHT : mode:=RIGHT;
       ord('s') : if (StopCount<2) or (_PzAll[1].SurfaceType=0) then begin
           inc(StopCount);
           mode:=stop;
         end;
       end;

   FirstInit_Message : begin
       GetElementIndexByName(eltIdentifier, 'xcimage',xcIndex,i);
       GetElementIndexByName(eltIdentifier, 'ycimage',ycIndex,i);
     end;
   StepEnd_Message : begin
       if ((t<20) and (t>hT )) and not (_PzAll[1].SurfaceType=0) then begin
         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;
end.
```

Usage of the keyboard for control of a wheel robot is discussed in the lesson *Wheel robot*. Here we discuss other procedures.

The global variable *XYCImage* is used for movement of the ground image according to the robot motion.

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

The first element of this array corresponds to the x coordinate, the second one – y. Values of these coordinates coincides with the values of identifiers *xcimage, ycimage* (Figure 5.7, right). Some additional global variables:

xcIndex, ycIndex – indiced of identifiers xcimage, ycimage;

StopCount – counter of stop modes is used in the game;

hT – time moment, the robot control is switched off if t<Th, the variable is used in the game.

The procedure **CalcZSurface** computes the contact surface (the ground). This procedure is called from the two function **ZgraphicElementFunctions** and *zSurface*. In this manner we achieve the coincidence the ground image and the contact surface. Note, that elements of the array XYCImage are added to the parameters *p1,p2* to shift the image according to the robot motion:

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

The following data are determined in the procedure **UserCalc**

- indices of identifiers *xcimage, ycimage*, the message **FirstInit_Message**;
- position of the center of the ground image *XYCImage*, the message **StepEnd_Message**; changes are made if the distance between the robot and the surface center along the *x* or *y* axis is greater than *Step*=0.5. If coordinates are changed, the new values for the identifiers *xcimage, ycimage* are set (the procedure **SetIdentifierValue**), and the corresponding GO is refreshed (the procedure **RefreshElement**).

The message **IntegrEnd_Message** is used for setting initial position if the ground image.

The animation window camera follows the robot motion. To assign this mode, move the mouse cursor to the robot body image, click the right mouse button and select the command *Follow the body Body* in the pop up menu.

The game corresponds to the identifier value SurfaceType=1,2. With the help of the keyboard keys ←↑→↓, S (stop) you should bring the robot inside the initial circle (marked with the red flag) and stop it there. First 20 seconds the robot control is turned off and the robot moves in a random manner. The S (stop) button is available two times a game. To make the game more

difficult, decrease the coefficient of friction (the identifier *ffr*) and/or increase the driving torque (the identifier *torqmax*).

5.1.8. Debugging control file in Delphi

The way to use Delphi for debugging the *Control file* is shown.

Firstly, generate equations of motion of your model. Do not forget to set *Pascal* as an output language. Run *Delphi* and open there the project *umtask.dpr* from the *[ModelName]\Pascal*. Open as well the *Control file*. It is named as *Cl[ModelName].pas*. It is necessary to set up the following options.

- **Project options**

  Use the menu command **Project | Options…** and click the **Directories | Conditionals** tab.
  - Input '..\[*ModelName*]' in the **Output directory** field. Otherwise the output *.dll will be created in the project directory '..\[*ModelName*]*Pascal*.
  - Input path to com directory, which is included in the UM installation, in the field **Search Path**. For example, *'..\Program Files\UM Software Lab\um30\com'*.

Using such options (see Figure 5.8) you are ready to try to compile the project. If it is necessary, fix arisen errors and compile the project again. After successful compiling you put breakpoints to interested code positions and set up run options.

Run options

Use the menu command **Run | Parameters…** and choose the **Local** tab.

- Input path to the UM simulation module **(UmSimul.exe)** in the field **Host Application**, for example *'C:\Program Files\UM Software Lab\UM30\bin\UmSimul.exe'*. **Host Application** is the application that calls *.dll and uses its functions. In our case with UM **Host Application** is the simulation module and *.dll is the compiled dll with equations of motion.
- Input path to model directory (..\[*ModelName*]) in the **Parameters** field. It leads that simulation module loads the model right after start (see Figure 5.9).

Use the menu command **Run | Run** to run the project. You are ready to debug the *Control file*.

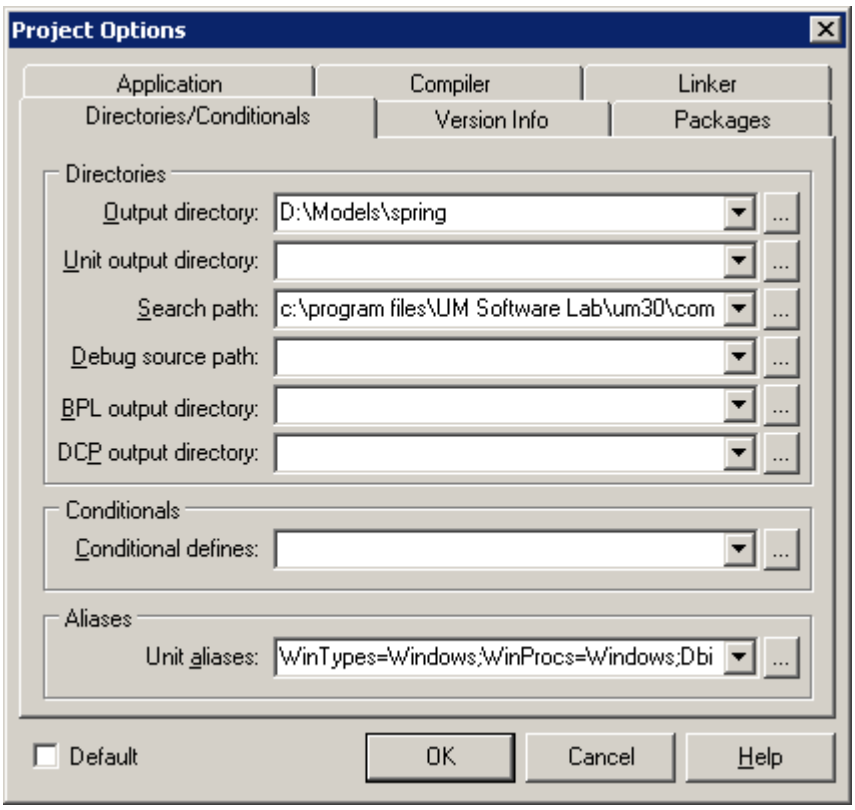


Figure 5.8.

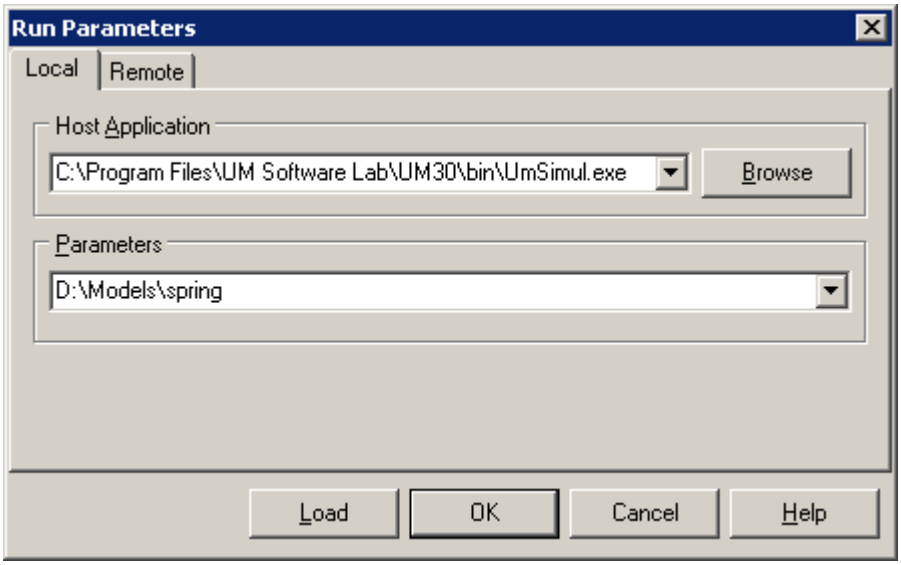


Figure 5.9.

5.2. Code implementation of functionals

Program implementations of *functionals* of *table processor* (see 4.2.6) are situated in DLL files in **Plugins** directory. Some standard functionals (min, max, rms, integral etc.) are collected in **standard.dll.** Some additional especial functionals are collected in **railway.dll.**

When simulation module starts **Plugins** directory is scanned and all functionals from DLL files are loaded.

Thus, development of new functionals can be realized by users. It needs just to copy a new DLL with new functionals to **Plugins** directory and they will be available when simulation module starts next time. **Plugins** directory can contain arbitrarily many DLL files. Every export procedure in DLL file is considered as a functional. The name of procedure is the name of functional in *table processor* window. All export procedures in DLL files in **Plugins** directory must be certain type. That type is described in {UM Data}\com\plugin.pas:

```
TFunctional = procedure(
    X, Y   : umPointer;         { Arrays of points                    }
                                { Pointers to array of umDouble       }
    N      : umInteger;         { Number of elements in X and Y arrays }
var Value  : umDouble;          { Return value                        }
var Success: boolean            { If the function succeeds return true,}
); cdecl;                       { else return false}
```

It is not allowed to include any incompatible export procedures to these DLL files. Example of implementation of functionals is situated in ..\plugins\standard.dpr.

The following example illustrates implementation of **Example_Min** functional.

```
(*********************************************************)
(*  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.
```

For successful compilation it is necessary that files **umtypes.pas** and **plugins.pas** from ..\com directory must be on search path.

The following example illustrates implementation of **Ampl** functional using Visual C++ 6.0.

```
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;
}
```

```
It is necessary to include def-file with definitions of exported functions in
the project:

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

5.3. Creating and using external libraries

External libraries are usually used for including into Universal Mechanism various mathematical models of forces that are impossible to describe with the help of the built-in force elements. Such a method is an alternative to programming in the *control file* and has the following distinctions.

- To develop your own external library with a mathematical model you can use any software environment and any program language that support creating Dynamic-Linked Libraries (DLL).
- You do not have to learn all the features concerning programming in a UM *control file*.
- You can include earlier developed external libraries to your UM model without any programming at all.

Generally simulation of dynamics of mechanical systems with connecting external libraries supposes the following steps.

- Creation of a mathematical model and its implementation as a DLL code according to regulations described below.
- Connection of the developed DLL with a UM model with the help of **Wizard of external libraries**, see **UM Simulation | Tools** menu command.
- Simulation of dynamics of a mechanical system.

External libraries have list of input and output signals, as well as list of its parameters. During the binding external library and UM model, external library input signals are connected with UM variables, that usually describe kinematical performances. Output signals are connected with UM parameters that usually parameterize forces and torques acting on a mechanical system.

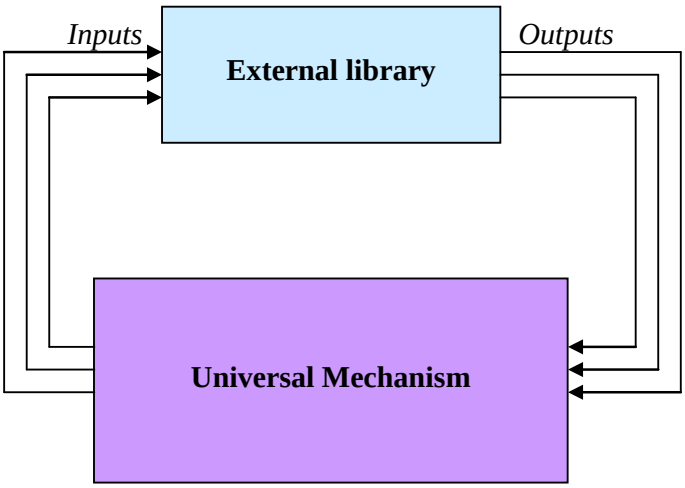


Figure 5.10.

5.3.1. Matlab/Simulink interface

Wizard of external library is also used for import models developed in Matlab/Simulink. More detailed information concerning importing *Matlab/Simulink* models into Universal Mechanism you can find in the part "Getting Started Using Universal Mechanism: Matlab/Simulink interface" of UM User's Manual. The last version of this part is available for downloading here: http://www.universalmechanism.com/download/70/eng/gs_um_control.pdf.

Note.	You can use your own external libraries in UM models and connect them with the help of **Wizard of external libraries** if only **UM Control/User-defined routines** tool is available in your UM configuration. Import Matlab/Simulink models are supported by **UM Control/Matlab Import** tool.

5.3.2. Declaration of procedures

Temples of an external library in *C* and *Pascal* languages are situated in the following directory: {UM Data}\samples\tutorial\extlibrary\templates.

UM distributive includes the sample DLL that simulates linear spring. You can find the sample model in {UM Data}\samples\tutorial\extlibrary\extlibspring directory. Source codes are located here: {UM Data}\samples\tutorial\extlibrary\source.

Let us consider functions that should be included into an external library.

Note.	Return integer parameter **status** is the exit code in many procedures considered below. Return value of 0 means no errors occurred, non-zero exit code means that some errors occurred.
Note.	To read procedure declarations in *C++* please note the following type declarations: typedef double* PDouble; typedef wchar_t* WChar;
Note.	Symbolic output parameters (model name, names of input and output signals, names of parameters) should be treated as UNICODE strings. Please note, that starting from DELPHI 2009, PChar type is equal to PWideChar (UNICODE-compatible). Earlier DELPHI versions consider PChar as PAnsiChar.
Note.	The calling convention **cdecl** is used for all functions (procedures).

`EXT_Initialize`

Delphi/Pascal:

```
procedure EXT_Initialize(var status: Integer); cdecl;
C++:
void _cdecl EXT_Initialize(int& status)
```

Initialization procedure. Here should be placed memory allocation, initialization of local variables, file opening and so on. Runs every time when simulation process in **UM Simulation** starts.

EXT_Terminate

Delphi/Pascal:

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

C++:

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

Occurs before unloading the library. Should do close operations: memory release, closing files and so on.

EXT_GetModelName

Delphi/Pascal:

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

C++:

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

Returns model name (no more than 255 chars) that is shown in the user interface. Memory is allocated on the side of the calling code. The procedure should not allocate any memory for name inside.

EXT_GetNumU

Delphi/Pascal:

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

C++:

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

Returns count of the input signals of the library.

EXT_GetUName

Delphi/Pascal:

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

C++:

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

Returns name of the input signal with i index, no more than 255 chars, uses 0-based index. Memory is allocated on the side of the calling code. Should not allocate any memory for name inside.

`EXT_GetNumY`

Delphi/Pascal:

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

C++:

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

Returns count of the output signals of the library.

`EXT_GetYName`

Delphi/Pascal:

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

C++:

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

Returns name of the output signal with i index, no more than 255 chars, uses 0-based index. Memory is allocated on the side of the calling code. Should not allocate any memory for name inside.

`EXT_GetY`

Delphi/Pascal:

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

C++:

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

It is the basic computation procedure. Having current process time (time) and current values of input signals (U) it calculates vector of output signals (Y). Memory for U, X and Y is allocated on the side of the calling code. U and Y are the pointers to first elements of the corresponding arrays. X is an array of state variables (reserved for the future use).

```
EXT_GetNumParameters
```

Delphi/Pascal:

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

C++:

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

Returns count of parameters of a model.

```
EXT_GetParameters
```

Delphi/Pascal:

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

C++:

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

Returns array of parameters of the model. Value is a pointer to the first element of the array of parameters. Memory is allocated on the side of the calling code. The procedure should not allocate any memory for value inside.

```
EXT_GetParameterName
```

Delphi/Pascal:

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

C++:

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

Returns name of the parameter with i index, no more than 255 chars, uses 0-based index. Memory is allocated on the side of the calling code. The procedure should not allocate any memory for name inside.

```
EXT_SetParameters
```

Delphi/Pascal:

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

C++:

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

Sets values of parameters of the model. Para is the pointer to the first element of the array of parameters. Memory is allocated on the side of the calling code. Procedure should not allocate any memory for value inside.

```
EXT_StepConfirmed
```

Delphi/Pascal:

```
procedure EXT_StepConfirmed; cdecl;
```

C++:

```
void _cdecl EXT_StepConfirmed()
```

The procedure is used in the case if the external library has a built-in solver. Otherwise might be empty. Program package “Universal Mechanism” uses numerical methods with variable step size and automatic accuracy control. Hence some steps of a numerical method might be cancelled and divided into several smaller ones to achieve the preset accuracy of the solution. If a step of the numerical method finishes successfully then right after calling EXT_GetY the EXT_StepConfirmed is called, otherwise several calling the EXT_GetY procedure go on one after another.

5.3.3. Features of compiling external libraries

Developers of external libraries should care that all functions (procedures) listed above should be declared as exporting in DLL and names of exporting functions should not be decorated. Decorated names are modified (in our case by compiler) names of functions that include information about types of arguments and results right in the name of function.

5.3.3.1. Compiling external libraries using C/C++

Please note that Microsoft Visual C++ and Microsoft Visual Studio decorate exporting function names by default. It is recommended to use DEF-file to control exporting functions and their names, see example in {UM Data}\SAMPLES\TUTORIAL\extlibrary\source\c directory.

To use DEF-file working in Microsoft Visual Studio simply copy that sample file to the project directory, add it into your project and set **Configuration properties | Linker | Input | Module Definition File to <Your DEF-file name>.def**, as it is shown in the Figure 5.11.

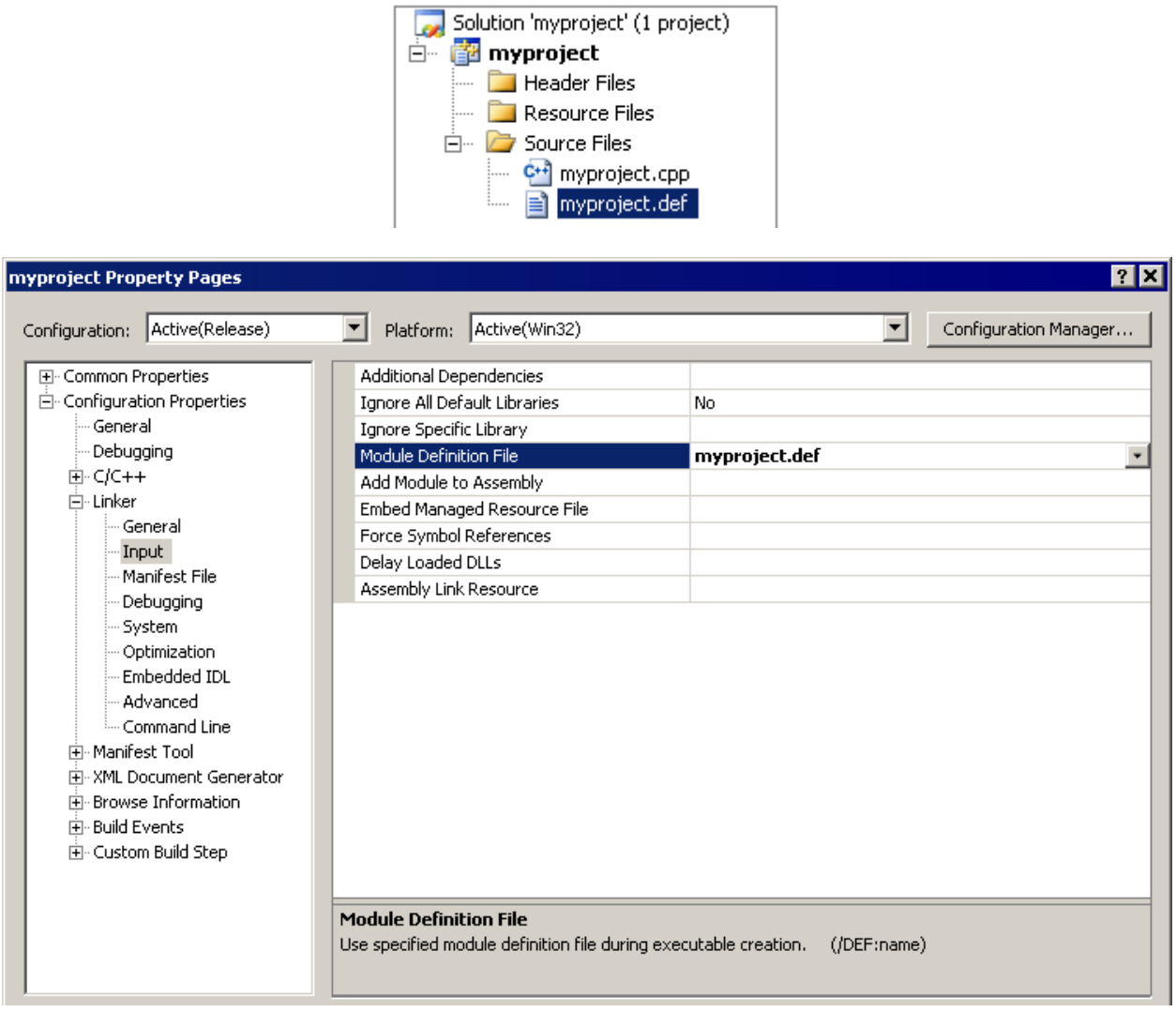


Figure 5.11. Project settings in MS Visual C++ 2008

Contents of the DEF-file:

```
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
```

Please note that the way how to define exporting functions and provide non-decorated function names depends on used C/C++ compiler. The way for MS Visual C++ is shown above. Using other compilers you probably should use declaration of functions like extern "C" and/or __declspec(dllexport) as it is shown below:

```
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. Compiling external libraries using Pascal

To define a list of exporting procedures you should use **Exports** clause, see {UM Data}\SAMPLES\TUTORIAL\extlibrary\source\pascal\UMLinearSpring.dpr for details. Typical **Exports** clause for UM external library is shown below.

```
...
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;
```

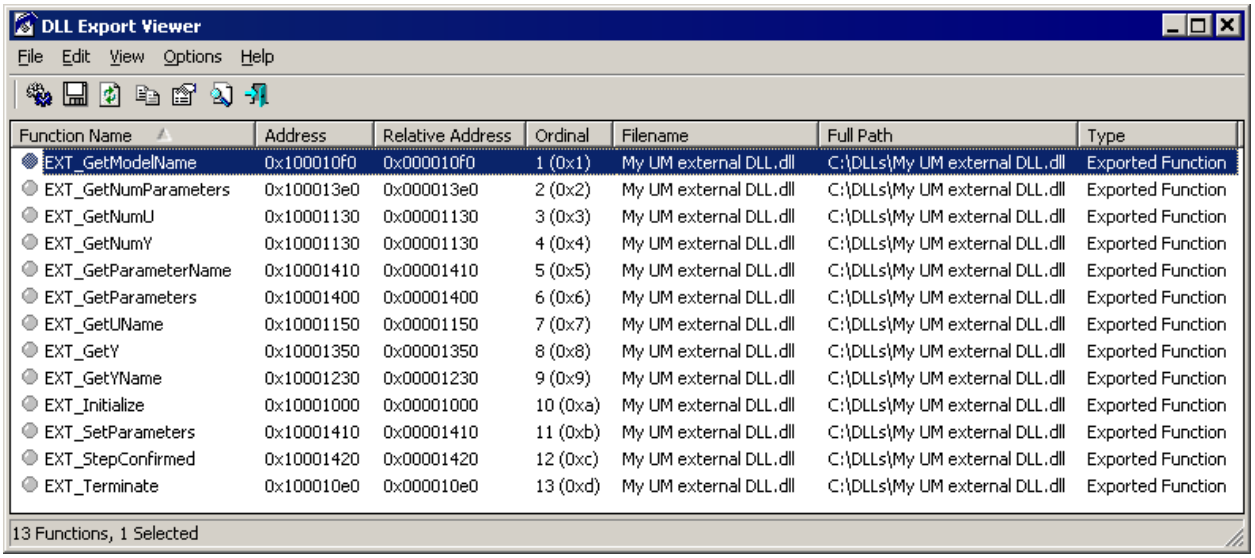
By default Delphi (Pascal) compiler does not decorate names of functions so it is not necessary to worry about that.

5.3.3.3. Troubleshooting

In the case of errors during loading DLL into UM before contacting the technical support make sure that

- your UM configuration includes **UM User Routines** tool from **UM Control** module, use **UM Simulation | Help | About…** menu command;
- DLL has all necessary exporting functions and names of these functions are not decorated. You can check the list of exporting functions and their names with the following utilities:
- **dumpbin.exe** from **Microsoft Visual Studio** package;
- **tdump.exe** from **Borland Developer Studio / Embarcadero RAD Studio** package;
- freeware utility **DLL Export Viewer**,

description: http://www.nirsoft.net/utils/dll_export_viewer.html,
link to download: http://www.nirsoft.net/utils/dllexp.zip.



Function Name	Address	Relative Address	Ordinal	Filename	Full Path	Type
EXT_GetModelName	0x100010f0	0x000010f0	1 (0x1)	My UM external DLL.dll	C:\DLLs\My UM external DLL.dll	Exported Function
EXT_GetNumParameters	0x100013e0	0x000013e0	2 (0x2)	My UM external DLL.dll	C:\DLLs\My UM external DLL.dll	Exported Function
EXT_GetNumU	0x10001130	0x00001130	3 (0x3)	My UM external DLL.dll	C:\DLLs\My UM external DLL.dll	Exported Function
EXT_GetNumY	0x10001130	0x00001130	4 (0x4)	My UM external DLL.dll	C:\DLLs\My UM external DLL.dll	Exported Function
EXT_GetParameterName	0x10001410	0x00001410	5 (0x5)	My UM external DLL.dll	C:\DLLs\My UM external DLL.dll	Exported Function
EXT_GetParameters	0x10001400	0x00001400	6 (0x6)	My UM external DLL.dll	C:\DLLs\My UM external DLL.dll	Exported Function
EXT_GetUName	0x10001150	0x00001150	7 (0x7)	My UM external DLL.dll	C:\DLLs\My UM external DLL.dll	Exported Function
EXT_GetY	0x10001350	0x00001350	8 (0x8)	My UM external DLL.dll	C:\DLLs\My UM external DLL.dll	Exported Function
EXT_GetYName	0x10001230	0x00001230	9 (0x9)	My UM external DLL.dll	C:\DLLs\My UM external DLL.dll	Exported Function
EXT_Initialize	0x10001000	0x00001000	10 (0xa)	My UM external DLL.dll	C:\DLLs\My UM external DLL.dll	Exported Function
EXT_SetParameters	0x10001410	0x00001410	11 (0xb)	My UM external DLL.dll	C:\DLLs\My UM external DLL.dll	Exported Function
EXT_StepConfirmed	0x10001420	0x00001420	12 (0xc)	My UM external DLL.dll	C:\DLLs\My UM external DLL.dll	Exported Function
EXT_Terminate	0x100010e0	0x000010e0	13 (0xd)	My UM external DLL.dll	C:\DLLs\My UM external DLL.dll	Exported Function

Figure 5.12. List of exporting functions by **DLL Export Viewer** utility

5.3.4. Including external libraries into UM models

There are two approaches to include external library into a UM model. The first one is the most common approach consists in using **Wizard of external libraries**, see **UM Simulation** program. The second one consists in using scalar forces of the *Library (DLL)* type, see **UM Input** program. Principles of developing DLLs are the same in both cases, see Sect. 5.3.2. There are differences in only approaches of including external libraries into UM models.

5.3.4.1. Wizard of external libraries

Connection of external libraries is fulfilled in **UM Simulation** program, see **Tools / External library interface** menu item. **Wizard of external libraries** opens, see Figure 5.13. You can see list of external libraries in left part of the window and settings of the selected library in the right part.

To assign a *variable* to an *input signal* of the external library it is necessary to create the variable in the **Wizard of variables** and then simply *drag* the variable to the necessary *input signal* and drop it there using the common *Drag-and-Drop* technology, see Figure 5.13. You can drag the variable from the **Wizard of variables**, **List of variables** or **Graphical window**. So any variable that could be created in the **Wizard of variables** can be assigned to *input signal* of external library.

Output signals of external libraries are usually assigned to parameters of the model. Therefore regardless of the fact that external libraries are connected to a UM model in **UM Simulation** program only you should create corresponding force elements and parameterize them in the step of preparing data in **UM Input** module.

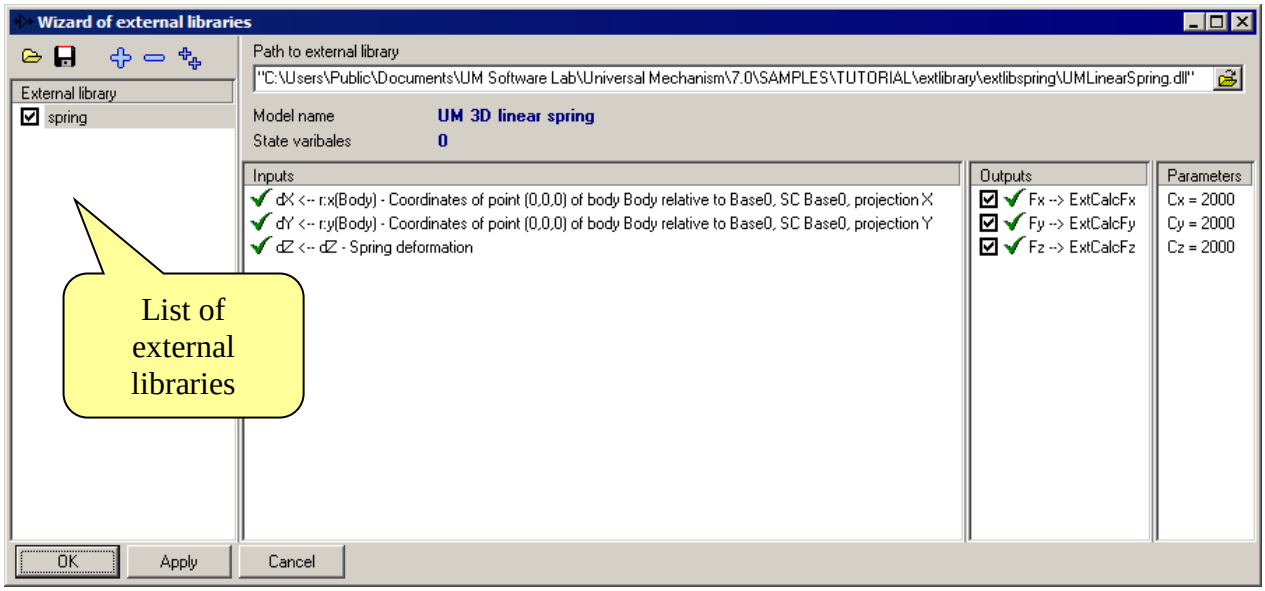


Figure 5.13. Wizard of external libraries

To attach an *output signal* of an external library to a model *parameter* double click on the necessary item to open the dialog window, see Figure 5.14, where you can select the model *parameter*. Some of *output signals* have a test/proving sense. In this case you may not to attach them to model parameters but simply create corresponding variables in the **Wizard of variables | Ext**.(ernal) **lib**.(raries) and plot them in graphical windows.

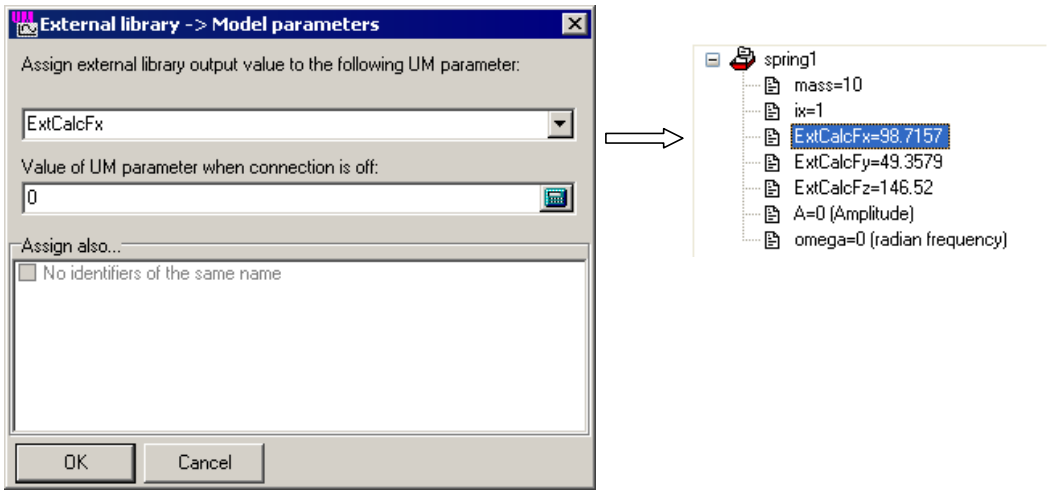


Figure 5.14. Attaching output values of an external library to UM model parameter

5.3.4.2. Scalar force of Library (DLL) type

Calculation of scalar (bipolar and joint) forces of the Library (DLL) type is also fulfilled in the external libraries. There are following differences between external libraries for scalar forces of the *Library (DLL)* type and for using via **Wizard of external libraries**.

- External library which is intended to be used as a scalar force of the *Library (DLL)* type should has three input signals (*x*, *v*, *t*) and one output signal, where x and v are coordinate and its time derivative for joint forces and are length of the element and its time derivative for bipolar forces. *t* is current time. External libraries which are intended to be used via **Wizard of external libraries** may have arbitrary number of input and output signals.
- Description external libraries for scalar forces of the *Library (DLL)* type is implemented in **UM Input** program, **Wizard of external libraries** works in **UM Simulation** program.
- Ready-to-use DLLs for scalar forces of the *Library (DLL)* type should be placed in the {UM Data}\lib\bfrc folder. External libraries that are attached via **Wizard of external libraries** might be placed anywhere.

Note. From point of view of computation efforts it is recommended that the external libraries that model joint or bipolar forces should be included into UM model as scalar forces of the *Library (DLL)* type instead using **Wizard of external libraries**.

5.3.4.3. Simultaneous connection of several libraries

Generally with the help of **Wizard of external libraries** you can load an arbitrary count of external libraries. For example, if a model of some mechanical system includes several identical force elements described with the help of a single external library then you should add this library in the **Wizard of external libraries** so many times as identical force elements are in the model.

Using simultaneous connection please note the following features. During loading several copies of the same *.dll file its data area remains the same, in other words, several copies of the library uses the same data segment. So if procedures, implemented in a library, change values of global variables and then use them for subsequent calculations, then several such interfaces will work and rewrite the same global variables. It finally leads to incorrect functioning of the external library.

Thus you can use the same DLL file for several interfaces if only EXT_GetY procedure for calculating output signals uses just input signals and model parameters without using global variables. Otherwise it is recommended to simply copy DLL file as many times as many interfaces you need.

Let us consider the following example. Let the *library.dll* realizes mathematical model of a force element that after each step saves some data in global variables to use them later during the next step. If we need to add, for example, four such force elements in our model we simple may copy the *library.dll as library1.dll, library2.dll, library3.dll* and *library4.dll* and then use them via the **Wizard of external libraries** or scalar forces of the *Library (DLL)* type.

5.3.5. Creating variables for input and output signals

Please note that with the help of the **Wizard of variables** you can create so called constant-variables (see Figure 5.15) as well as time-function-variable and identifier-variable (see Figure 5.16 and Figure 5.17 correspondingly).

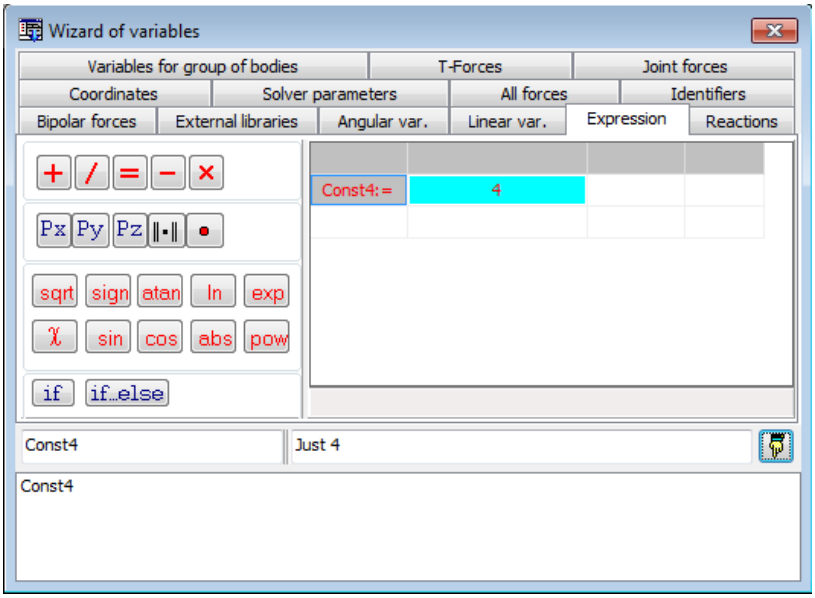


Figure 5.15. Constant-variable

Figure 5.16, Figure 5.17 show how to create variables for identifiers (parameters) with the help of the **Wizard of variables**. Then these variables will be used in the **Expression** tab for creating time-functions as variables.

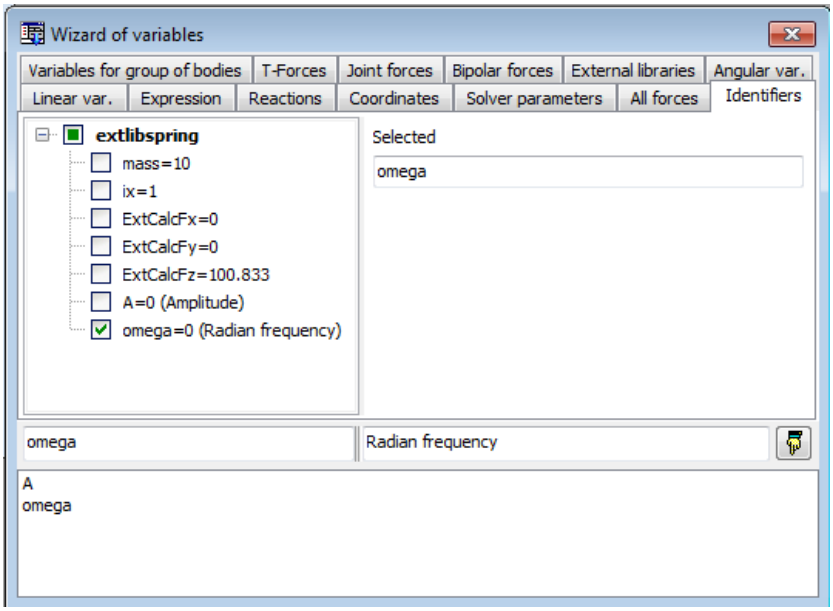


Figure 5.16. Identifier-variable (*A* and *omega*)

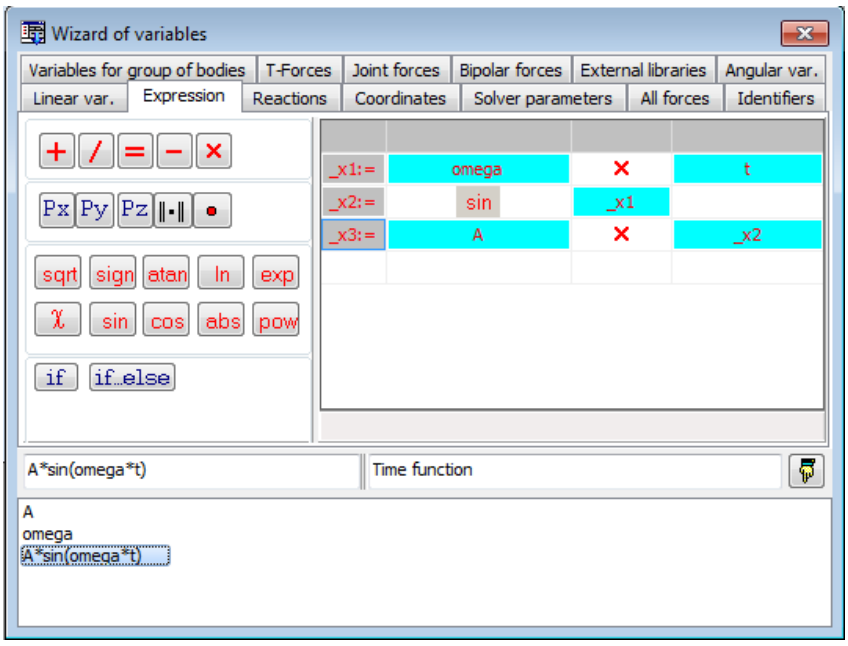


Figure 5.17. Time-function-variable *A*sin(omega*t)*

You can create variables that reflect *input* and *output signals* of external libraries with the help of the **Wizard of variables**, see **Ext**.(ernal) **lib**.(rary). It helps create necessary variables for plotting in a graphical window quickly. For example, during debugging external library or carrying out research.

Please see Sect. “4.3.2. Wizard of variables” of the UM User’s Manual for more detailed information.

5.3.6. Cascading external libraries

Cascading is a technique when *output signals* of one external library are assigned *to input signals* of other libraries. It let the user a possibility to create, for example, elements of electric or hydraulic circuits as separate external libraries and then connect them into some circuits.

Using cascading do not forget that *output signals* will be assigned to *input signals* with delay in one step size of a numerical method. Moreover at initial time (t=0), when *output signals* are not yet defined, such input elements will be set to zero. So, values of *output signals*, obtained at step *i* of a numerical method, will be given to *input* ones on step *i+1*.

5.3.7. Example of using external libraries

Let us consider an example of creating the external library that defines a mathematical model of a linear spring. Considered here mathematical model is a simplified one and takes into account linear displacements only, angular displacements are ignored.

Please find a model of the mechanical system that is considered in this section in {UM Data}\SAMPLES\TUTORIAL\extlibrary\extlibspring directory, see Figure 5.18. Source codes in C and Pascal are in {UM Data}\SAMPLES\TUTORIAL\extlibrary\source directory, see umlinearspring.dpr and umlinearspring.cpp. Let us not to explain in details all the features of their implementation. Examples are very simple and obvious for anybody who familiar with programming.

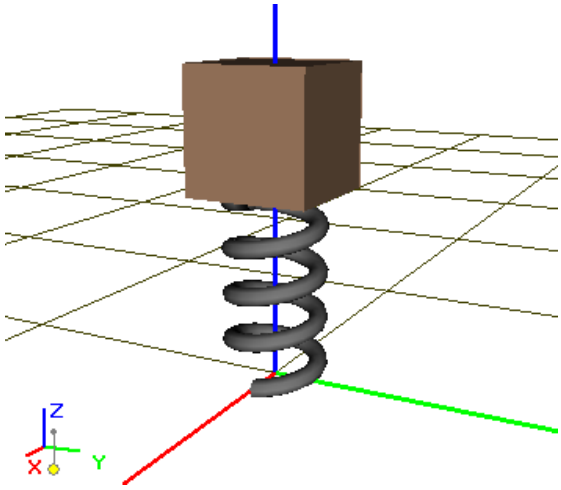


Figure 5.18. View of the model

Let us consider some features of description of this model. Run **UM Input** program and load the model from the {UM Data}\SAMPLES\TUTORIAL\extlibrary\extlibspring directory.

Firstly note that the bipolar force **FictitiousSpring** is introduced into the model. This force has zero parameters and introduced to visualize the spring only. Force itself will be calculated in the external library.

Secondly note that the **SpringForce** of **T-Force** type is also introduced into the model. Projections of this force on X, Y and Z axes are described with the help of **ExtCalcFx**, **ExtCalcFy** and **ExtCalcFz** parameters correspondingly. *Output signals* (in fact, calculated forces) from the external library will be assigned to these parameters of the model.

Now please run the **UM Simulation** program and load the same model. Window of the **Wizard of external libraries** is shown in Figure 5.13. Please note that deformations of the spring along each axis should be assigned to input signals of the external library. Deformations along X and Y axes are equal to position of the body in SC0. To calculate a spring deformation along Z axis it needs to consider the length of the unloaded spring (0.3 m). How to create necessary variables with the help of the **Wizard of variables** are shown in Figure 5.19, Figure 5.20.

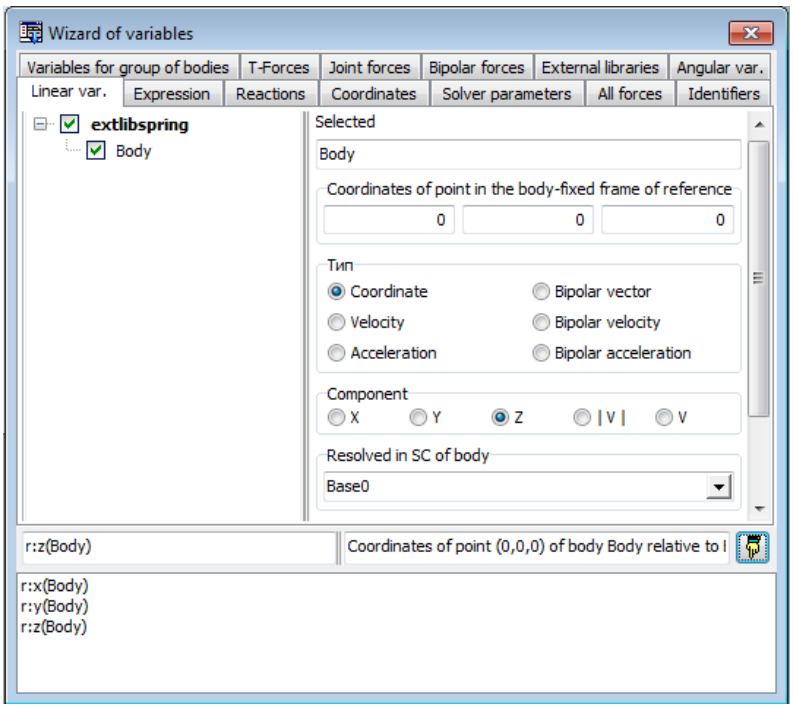


Figure 5.19. Coordinates of the center mass of the body in projection on X and Y axis

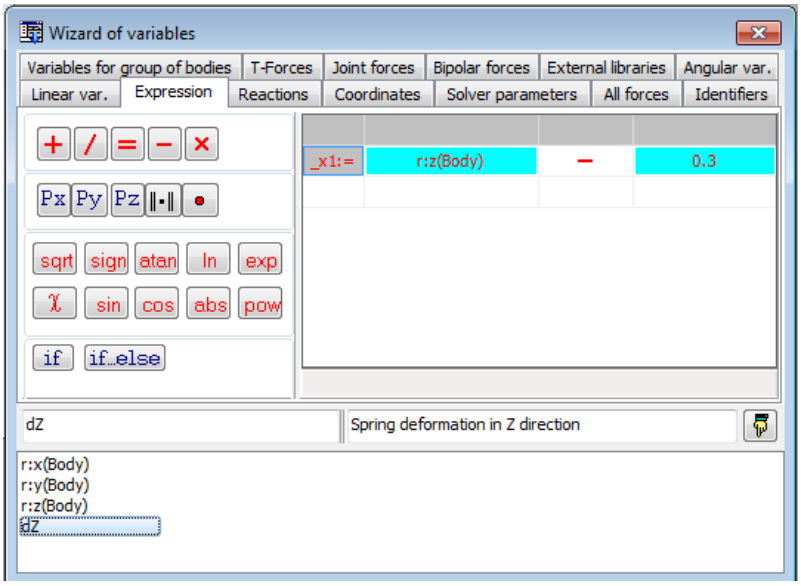


Figure 5.20. Spring deformation in Z direction

5.4. Using of external DLLs for creep force calculation

Creep forces can be calculated in UM by using external user-defined DLLs. Such external DLL must have as minimum four functions (procedures)[1]: InitProc, Contact1Proc, Contact2Proc, FinProc. The description of these functions is given below.

InitProc is the initialization function. It is called directly before the integration start. The parameters for this function are paths to files of wheel and rail profiles assigned to the vehicle model. All preliminary operations such as load of profiles and so on should be made in this function.

Function declaration:

C

```
void __cdecl InitProc(
  const int N,
  const WChar WheelProfileFiles[24],
  const WChar RailProfileFiles[2])
```

Pascal

```
TWheelProfileFiles  = array [0..23] of PChar;
TRailProfileFiles  = array [0..1] of PChar;

TInitProc = procedure(
 const N: Integer;
 const WheelProfileFiles: WheelProfileFiles;
 const RailProfileFiles: TRailProfileFiles
); cdecl;
```

where

N is count of wheels;

WheelProfileFiles is array of paths to wheel profiles;

RailProfileFiles is array of paths to rail profile.

Only first N elements of the *WheelProfileFiles* array are filled, others are empty. The first element (array index = 0) contains the path (file name) to the profile of the first left wheel, the second element (array index = 1) – the path to first right wheel, the third element – the path to the second left wheel, the fourth element – the path to the second right wheel and so on. The first element (array index = 1) of the *RailProfileFiles* array contains the path to the left rail profile, the first one – the path to the right rail profile.

Contact1Proc, *Contact2Proc* are functions for calculation of creep forces for the first and the second contact points respectively.

[1] It is called *FUNCTION* in C and *PROCEDURE* in Pascal. We will use *FUNCTION* term below.

Function declaration:

C

```
void __cdecl Contact1Proc(
  const int iws,
  const int iwheel,
  const double ywc,
  const double yrc,
  const double alpha,
  const double ksix,
  const double ksiy,
  const double phi,
  const double N,
  const double f,
  const double v0,
  const double wr,
  double &Fx,
  double &Fy)
```

Pascal

```
procedure TContact1Proc(
 const iws,
 iwheel : Integer;
 const ywc, yrc,
 alpha,
 ksix, ksiy, phi,
 N, f,
 v0,
 wr : Double;
 var Fx, Fy : Double);
 cdecl;
```

where

iws	wheelset index (starts from 0);
iwheel	wheel index (0 – left, 1 – right);
ywc, yrc	coordinates of contact points on wheel and rail;
alpha	rotate angle of wheel profile relative to rail profile;
ksix, ksiy, phi	longitudinal and lateral creep, spin;
N, f	normal force, friction coefficient;
v0	vehicle velocity;
wr	wheel radius in contact point;
returned parameters:	
Fx, Fy	calculated creep forces

Functions *Contact1Proc*, *Contact2Proc* should contain algorithm which calculates creep forces by using input data: wheelset and wheel indices, lateral coordinates of contact points on wheel and rail profiles (in SC of wheel and rail profiles, see Chapter 8 of UM manual), rotate

angle of wheel profile relative to rail profile, longitudinal and lateral creeps, spin, normal contact force, friction coefficient, vehicle velocity and wheel radius in contact point. In case of single-point contact, only the *Contact1Proc* function is called. If the contact is the double-point one, then the *Contact2Proc* procedure is called for the second point as well. Calculated creep forces as a result of the work of the procedures should be assigned to variable parameters *Fx* and *Fy*.

All formal parameters of the functions are in System International units excepting wheel and rail profiles which are set in millimeters.

FinProc is the finalization function. It is called after finishing the integration process. The function is intended for the execution of necessary final actions when the integration stops.

Function declaration:

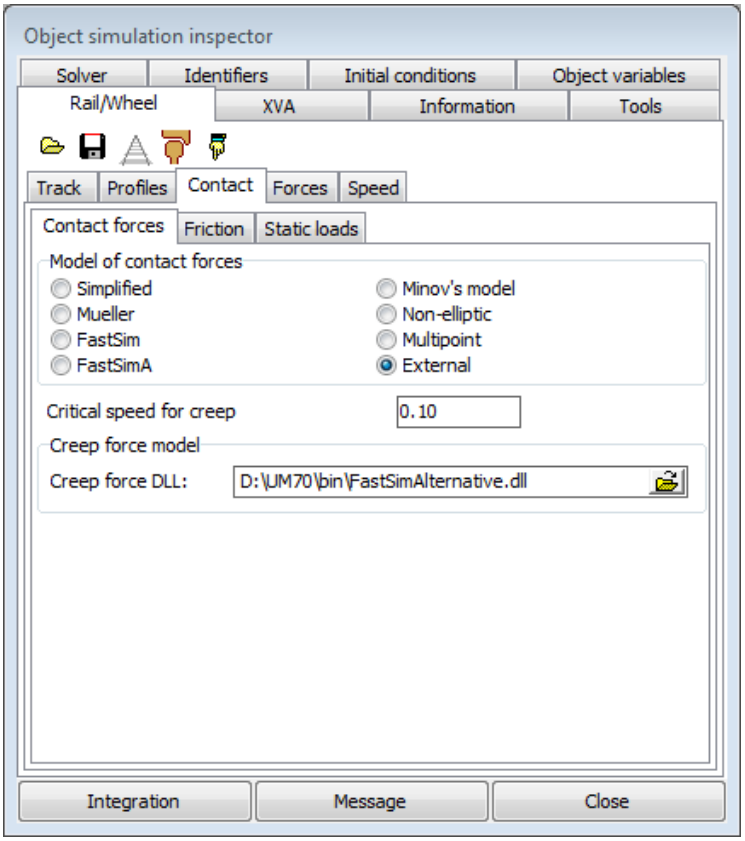
C

```
void __cdecl FinProc()
```

Pascal

```
Procedure FinProc; cdecl;
```

To set the external creep force model in UM, the user should select the **External** option in the **Creep force model** group on the **Wheel/rail | Contact | Creep forces** tab. The external DLL which will be used for creep force calculation must be set by using the **Creep force DLL** edit box.



Example of calculation tangential contact forces according to Mueller model implemented in C and Pascal are presented in Sect. 5.4.1 and 0 correspondingly.

5.4.1. External creep force DLL implemented in C

Sample code is situated here: {UM Data}\SAMPLES\TUTORIAL\extcreeplibrary\C.

```
#include "stdafx.h"
#include "math.h"

typedef wchar_t*  WChar;

void __cdecl InitProc(const int N, const WChar WheelProfileFiles[24], const
WChar RailProfileFiles[2])
{
// Some code might be here
}

void __cdecl FinProc()
{
// Some code might be here
}

void MuellerModel(
  const double ksix,
  const double ksiy,
  const double N,
  const double f,
  double &Fx, double &Fy)
{
      double ksi, P, Kc, Fxy;
      int m;

      m = 3;
      ksi = sqrt(ksix*ksix + ksiy*ksiy);
      P = 0.001*N;
      Kc = P*(235 - P*(2.4 - 0.01*P));
    Fxy = Kc/pow((1.0 + pow((Kc*ksi/(f*P)), m)), 1/m)*1000;
    Fx = -ksix*Fxy;
    Fy = -ksiy*Fxy;
}

void __cdecl Contact1Proc(
  const int iws,
  const int iwheel,
  const double ywc,
  const double yrc,
  const double alpha,
  const double ksix,
  const double ksiy,
  const double phi,
  const double N,
  const double f,
  const double v0,
  const double wr,
  double &Fx, double &Fy)
{
      MuellerModel(ksix, ksiy, N, f, Fx, Fy);
}

void __cdecl Contact2Proc(
  const int iws,
  const int iwheel,
  const double ywc,
  const double yrc,
  const double alpha,
```

```
  const double ksix,
  const double ksiy,
  const double phi,
  const double N,
  const double f,
  const double v0,
  const double wr,
  double &Fx, double &Fy)
{
      MuellerModel(ksix, ksiy, N, f, Fx, Fy);
}
```

5.4.2. External creep force DLL implemented in Pascal

Sample code is situated here: {UM Data}\SAMPLES\TUTORIAL\extcreeplibrary\Pascal.

```
library ExternalContactDLL;

uses
  SysUtils,
  Classes,
  Math;

type
  TWheelProfileFiles  = array [0..23] of PChar;
  TRailProfileFiles  = array [0..1] of PChar;

  TInitProc = procedure(
    const N: Integer;
    const WheelProfileFiles: TWheelProfileFiles;
    const RailProfileFiles: TRailProfileFiles); cdecl;

  TFinProc = procedure; cdecl;

  TContactProc = procedure(
    const iws, iwheel : Integer;
    const ywc, yrc, alpha, ksix, ksiy, phi, N, f, v0, wr : Double;
    var Fx, Fy : Double); cdecl;

  procedure MuellerModel(
    const ksix, ksiy, N, f: Double;
    var Fx, Fy: Double);
  var
    ksi, P, Kc, Fxy: Double;
    m: Integer;

  begin
    m := 3;
    ksi := sqrt(ksix*ksix + ksiy*ksiy);
        P := 0.001*N;
        Kc := P*(235 - P*(2.4 - 0.01*P));
    Fxy := Kc/power((1.0 + power((Kc*ksi/(f*P)), m)), 1/m)*1000;
    Fx := -ksix*Fxy;
    Fy := -ksiy*Fxy;
  end;

  procedure InitProc(
    const N: Integer;
    const WheelProfileFiles: TWheelProfileFiles;
    const RailProfileFiles: TRailProfileFiles); cdecl;
  begin
// Some code might be here
  end;


procedure FinProc;
  begin
// Some code might be here
  end;

  procedure Contact1Proc(
    const iws, iwheel: Integer;
    const ywc, yrc, alpha, ksix, ksiy, phi, N, f, v0, wr: Double;
    var Fx, Fy: Double); cdecl;
  begin
```

```
    MuellerModel(ksix, ksiy, N, f, Fx, Fy);
  end;

  procedure Contact2Proc(
    const iws, iwheel: Integer;
    const ywc, yrc, alpha, ksix, ksiy, phi, N, f, v0, wr: Double;
    var Fx, Fy: Double); cdecl;
  begin
    MuellerModel(ksix, ksiy, N, f, Fx, Fy);
  end;

exports
  InitProc,
  Contact1Proc,
  Contact2Proc,
  FinProc;
end.
```