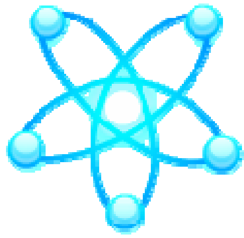


VersaDim

Version 0.6.3

User manual

Date: 19/01/2011

Copyright 2010-2011 Edgar Teufel

VersaDim is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

VersaDim is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with VersaDim. If not, see <http://www.gnu.org/licenses/>.

Manual

VersaDim is a "versatile dimensioning" tool. It implements several generic search algorithms that can be made use of by linking VersaDim to some existing legacy application. Linking means here that VersaDim will talk to these external applications via one of two interfaces: "DLL" and "named pipe". It would then vary some parameters within predefined bounds following these generic search algorithms and hopefully deliver a good parameter set in terms of high or low cost function value.

Dynamic Link Library communication interface

The DLL interface consists of exactly four functions: Init, Calc, GetHostName and GetAppVersion compiled to a dynamic link library. When configured accordingly, VersaDim will call these library functions repeatedly and use the information returned to vary the parameter set.
A dynamic link library compatible with VersaDim must export each of these four functions. The calling convention must be "standard call" ("stdcall") for all of them. In order to avoid name conflicts, "GetHostName" can also be named "vdGetHostName". Inside the compiled DLL file, all of these function names must appear in lower case letters without trailing or leading underscores. Please choose your compilers name decoration settings accordingly.

Description of the interface functions

Init
Gets called by VersaDim only once: before a new search run starts, i.e. the function "calc" is called for the first time. Therefore, it is the right place to put initialization code to.
Arguments: - nothing -
Return value: Integer value representing initialization result.
Remark: VersaDim does not react on the return value in any way. However it reports it to the user. So this can be used to update the user about the outcome of the initialization.
Calc
Gets called by VersaDim for many sets of parameters.
Arguments: Array of double precision (eight byte) floating point variables and integer value specifying the length of the array
Return value: Cost function value as double precision floating point value

GetHostName / vdGetHostName
Gets called by VersaDim during setup and once before search run starts. Must deliver host name as string and write it back to argument.
Arguments: Pointer to array of 255 bytes interpreted as Null-terminated string; memory handling is done by VersaDim (please make sure not to write beyond bound)
Return value: Integer value; 0 means successful operation, -1 means that some error has occurred
Remark: VersaDim does not react on argument or return value in any way. It only writes the argument representing the host name to the result file.

GetAppVersion
Gets called by VersaDim during setup and once before search run starts. Must deliver version of the cost function calculating code as string and write it back to argument.
Arguments: Pointer to array of 255 bytes interpreted as Null-terminated string; memory handling is done by VersaDim (please make sure not to write beyond bound)
Return value: Integer value; 0 means successful operation, -1 means that some error has occurred
Remark: VersaDim does not react on argument or return value in any way. It only writes the argument

representing the application version to the result file.

Example implementations

In the following, sample implementations for the VersaDim DLL interface are given in the languages C/C++, Object Pascal (Delphi) and FORTRAN90. These implementations have been tested with Borland Developer Studio 2006 (Version 10.0.2558.35231 Update 2, Hotfixed) and Compaq Visual Fortran Professional Edition (Version 6.6.0).

C

```
//---------------------------------------------------------------------------
#include <windows.h>
#pragma argsused

__declspec(dllexport) __stdcall int init (void);
__declspec(dllexport) __stdcall double calc (double* Arg, int Length);
__declspec(dllexport) int __stdcall vdgethostname (char* Name);
__declspec(dllexport) int __stdcall getappversion (char* Version);


int WINAPI DllEntryPoint(HINSTANCE hinst, unsigned long reason, void* lpReserved)
{
	return 1;
}
//---------------------------------------------------------------------------



int __stdcall init (void)
{
/* ---------------------------------------------
	FILL IN YOUR INITIALIZATION CODE HERE PLEASE
	--------------------------------------------- */
	return(0);
}
//---------------------------------------------------------------------------



double __stdcall calc (double* Arg, int Length)
{
/* -----------------------------------------------------
	FILL IN YOUR NUMERICAL CODE HERE PLEASE (example below)
	----------------------------------------------------- */
	double sum = 0.0;
	int i;

	for (i=0; i < Length; i++) sum += Arg[i];
	return(sum);
}
//---------------------------------------------------------------------------



int __stdcall vdgethostname (char* Name)
```

```
{
        int a = 1;
        strcpy(Name, "MyName");    /* <--- add code here to find out host name and copy it to "Name"  */
        a = 0;
        return(a);
}
//---------------------------------------------------------------------------
```

```
int __stdcall getappversion (char* Version)
{
        int a = 1;
        strcpy(Version, "0.1");
        a = 0;
        return(a);
}
//---------------------------------------------------------------------------
```

C++

```
//---------------------------------------------------------------------------
#include <windows.h>
#pragma argsused

extern "C" int __declspec(dllexport) __stdcall init (void);
extern "C" double __declspec(dllexport) __stdcall calc (double* Arg, int Length);
extern "C" int __declspec(dllexport) __stdcall vdgethostname (char* Name);
extern "C" int __declspec(dllexport) __stdcall getappversion (char* Version);

BOOL WINAPI DllMain(HINSTANCE hinstDLL, DWORD fwdreason, LPVOID lpvReserved)
{
        return 1;
}
//---------------------------------------------------------------------------
```

```
extern "C" int __declspec(dllexport) __stdcall init (void)
{
/* ---------------------------------------------
        FILL IN YOUR INITIALIZATION CODE HERE PLEASE
        ------------------------------------------- */
        return(0);
}
//---------------------------------------------------------------------------
```

```
extern "C" double __declspec(dllexport) __stdcall calc (double* Arg, int Length)
{
/* -----------------------------------------------------
        FILL IN YOUR NUMERICAL CODE HERE PLEASE (example below)
        ---------------------------------------------------- */
        double sum = 0.0;

        for (int i=0; i < Length; i++) sum += Arg[i];
        return(sum);
}
//------------------------------------------------------------------------



extern "C" int __declspec(dllexport) __stdcall vdgethostname (char* Name)
{
        int a = 1;
        strcpy(Name, "MyName");    /* <--- add code here to find out host name and copy it to "Name"  */
        a = 0;
        return(a);
}
//------------------------------------------------------------------------



extern "C" int __declspec(dllexport) __stdcall getappversion (char* Version)
{
        int a = 1;
        strcpy(Version, "0.1");
        a = 0;
        return(a);
}
//------------------------------------------------------------------------
```

Object Pascal

Library file:

```
library InterfaceTestDLL;

uses
  SysUtils,
  Classes,
  ServerNumerics in 'ServerNumerics.pas';

{$R *.res}

exports
  Init name 'init',
  Calc name 'calc',
  GetHostName name 'gethostname',
  GetAppVersion name 'getappversion';
begin
end.
```

Implementation file:

```
unit ServerNumerics;

interface

uses SysUtils;

type

  TStringResponse = array[0..255] of char;
  TDoubleArray = array of double;

  function Init: integer; stdcall;
  function Calc (Arg: TDoubleArray; Length: integer): double; stdcall;
  function GetHostName (var Name: TStringResponse): integer; stdcall;
  function GetAppVersion (var Version: TStringResponse): integer; stdcall;

implementation


function Init: integer; stdcall;
begin
  result := 0;
end;



function Calc (Arg: TDoubleArray; Length: integer): double; stdcall;
var i: integer;
begin
  result := 1.0;
  for i := 0 to Length - 1 do result := result * Arg[i];
end;



function GetHostName (var Name: TStringResponse): integer; stdcall;
begin
  result := -1;
  StrCopy(@Name[0], PChar('MyName'));
  result := 0;
end;



function GetAppVersion (var Version: TStringResponse): integer; stdcall;
begin
  result := -1;
  StrCopy(@Version[0], PChar('0.2'));
  result := 0;
end;

end.
```

FORTRAN

```
function Init
! user supplied initialization function; gets called once by VersaDim before parameter variation starts
  !DEC$ ATTRIBUTES DLLEXPORT, STDCALL :: Init
  implicit none
  integer          :: Init                          ! VersaDim will report this return value

  ! Variables

 ! --------------------------------------------
 ! FILL IN YOUR INITIALIZATION CODE HERE PLEASE
 ! --------------------------------------------
 Init = 0

end function



function Calc (Argument, Length)
! user supplied calculation function; called by VersaDim to demand cost function value for given
parameters
  !DEC$ ATTRIBUTES DLLEXPORT, STDCALL :: Calc
  implicit none
  double precision               :: Calc                 ! cost function value as computed here for given
parameters
  double precision, intent(in)      :: Argument(*)            ! parameter array -
  integer, intent(in)              :: Length                 ! number of parameters in array

  ! Variables
  integer  :: i

  ! ------------------------------------
  ! FILL IN YOUR NUMERIC CODE HERE PLEASE
  ! ------------------------------------

  ! Here are two examples for calculating the cost function value:

  ! 1.) Summing up all parameters and delivering the result would be done like that
  Calc = 0
  do i = 1, Length
    Calc = Calc + Argument(i)
  end do

  ! 2.) This gives a more undulating scenery (to test algorithms...)
  Calc = cos(Argument(1)) + cos(Argument(2)) + 2.0

end function



function GetHostName (HostNameStr)
! administration function; gets and returns host name
  !DEC$ ATTRIBUTES DLLEXPORT, STDCALL :: GetHostName
```

```
  !DEC$ ATTRIBUTES REFERENCE :: HostNameStr
  implicit none
  integer            :: GetHostName
  character(255)      :: HostNameStr                          ! VersaDim may display the host name of this
machine

  GetHostName = -1
  ! -------------------------------------------------
  ! FILL IN YOUR CODE TO FIND OUT HOSTNAME HERE PLEASE
  HostNameStr = "MyName"//CHAR(0) ! --> example 1
  ! example 2: call hostnm(HostNameStr)
  ! -------------------------------------------------
  GetHostName = 0
end function



function GetAppVersion(VersionStr)
! administrative function; tells VersaDim version number of this library (number string has to be set
manually!!)
  !DEC$ ATTRIBUTES DLLEXPORT, STDCALL :: GetAppVersion
  !DEC$ ATTRIBUTES REFERENCE :: VersionStr
  implicit none
  integer        :: GetAppVersion
  character(255) :: VersionStr                             ! VersaDim will mention this version number in its
result report

  GetAppVersion = -1
  !---change version string here:----
  VersionStr = "0.1"//CHAR(0)
  !----------------------------------
  GetAppVersion = 0
end function
```

Named pipe communication interface

Within this context, "named pipe" is meant to describe the Microsoft Windows extension of the pipe concept commonly found on POSIX systems. Therefore, this interface is only available on the Microsoft Windows platform and can not be ported directly to other operating systems. However, similar functionality to implement inter-process-communications are available on other platforms as well.

The named pipe interface actually is a client-server model of computing. Here, VersaDim is the client that would attach to a named pipe created by a server and send requests to query it. Thus, a user supplied application compatible with VersaDim that is to communicate via this interface must implement a named pipe server defined in the next section.

Pipe server definition version 1.1

The data transformed from and to the server per request are organized in frames. Each request to the server and each response consists of exactly one frame:

Element	Data type	Size	Comment
Size	unsigned integer	4 byte	Size in byte of the frame
Count	unsigned integer	4 byte	Number of characters in Data
Data	Null-terminated string	max. 1024 byte	Command/Response

For requests from the client (VersaDim) to the server, the following commands can be contained in the Data field of the frame:

Command	Comment
calc	Calculate cost function value for these parameters; must be followed by parameter list (see explanation below)
init	Initialize calculation of cost function values; called once before first "calc" request
server version	Send version number of named pipe server protocol
application version	Send version number of cost function calculating application
hostname	Send name of host
shutdown	Shutdown server

Please note that the "calc" command has to be followed by a blank-separated list of parameters. Between "calc" and the first parameter there must also be one blank character.
The individual parameter values are given as character sequence. Each sequence must only contain the characters "+", "-", ".", "e", "E" or any digit. In the following, some valid examples are given:
calc 1.0 2 3.00 0.000
calc 1.0
calc -3.2 -3.29e2

After evaluation of the command, the server sends back one frame to the client. It writes its response to the Data field of the frame and sets the Size and Count fields accordingly.

The server application must create a named pipe using the Microsoft Windows API function

"CreateNamedPipe" with the following arguments (uppercase words represent constants defined in Microsoft Windows API):

Argument	Value
lpName	`\\.\pipe\`*`PipeName`* where *`PipeName`* is as read from command line arguments (example: `\\.\pipe\VersaDimPipeServer`)
dwOpenMode	PIPE_ACCESS_DUPLEX
dwPipeMode	PIPE_TYPE_MESSAGE || PIPE_READMODE_MESSAGE || PIPE_WAIT
nMaxInstances	PIPE_UNLIMITED_INSTANCES
nOutBufferSize	sizeof(TPipeMessage)
nInBufferSize	sizeof(TPipeMessage)
nDefaultTimeOut	NMPWAIT_USE_DEFAULT_WAIT
lpSecurityAttributes	NULL

Reference implementation

In case the definitions in the last section leave any open points, the reference application listed in the following will clarify them. It is written in Object Pascal and has been test-compiled with Borland Delphi 2006 but may be translated into C++ with little effort as it heavily relies on functionality offered by the operating system API.

```
unit PipeServer;

interface

uses Classes, Windows, SysUtils;

type

      TPipeMessage = packed record
        Size: DWORD;
        Count: DWORD;
        Data: array[0..1024] of char;
      end;

      TWorkFunction = function (Command: string): string of object;
      TInitFunction = function: string of object;

      TPipeServer = class (TThread)
        private
          FHandle: THandle;
          FPipeName: string;
          FWorkFunction: TWorkFunction;
          FInitFunction: TInitFunction;
          FRequest, FResponse: string;
          const ServerVersionStr = '1.1';
          procedure CallWorkFunction;
        public
          constructor Create (PipeName: string); reintroduce;
          destructor Destroy; override;
          procedure StartUpServer;
```

```
          procedure ShutDownServer;
          procedure Execute; override;
          property WorkFunction: TWorkFunction read FWorkFunction write
FWorkFunction;
          property InitFunction: TInitFunction read FInitFunction write
FInitFunction;
      end;


implementation




{ TPipeServer }

constructor TPipeServer.Create(PipeName: string);
begin
 FPipeName := Format('\\%s\pipe\%s', ['.', PipeName]);
 FHandle := INVALID_HANDLE_VALUE;
 FWorkFunction := nil;
 FInitFunction := nil;
 inherited Create(true);       // always create us suspended...
end;



destructor TPipeServer.Destroy;
begin
  if FHandle <> INVALID_HANDLE_VALUE then ShutDownServer;
  inherited Destroy;
end;



procedure TPipeServer.ShutDownServer;
begin
 if FHandle <> INVALID_HANDLE_VALUE then
   begin
     CloseHandle(FHandle);
     FHandle := INVALID_HANDLE_VALUE;
   end;
end;



procedure TPipeServer.StartUpServer;
begin
 if FHandle = INVALID_HANDLE_VALUE then
   begin
     FHandle := CreateNamedPipe (PChar(FPipeName), PIPE_ACCESS_DUPLEX,
       PIPE_TYPE_MESSAGE or PIPE_READMODE_MESSAGE or PIPE_WAIT,
PIPE_UNLIMITED_INSTANCES,
       sizeof(TPipeMessage), sizeof(TPipeMessage),
       NMPWAIT_USE_DEFAULT_WAIT, nil);
     if FHandle = INVALID_HANDLE_VALUE then raise Exception.Create('Could not
```

```
create PIPE');
   end;
end;



procedure TPipeServer.Execute;
var Written: Cardinal;
    InMsg, OutMsg: TPipeMessage;
    Request, Response: string;
begin
  while not Terminated do
    begin
      if FHandle <> INVALID_HANDLE_VALUE then
        begin
          if ConnectNamedPipe(FHandle, nil) then
            try
             InMsg.Size := sizeof(InMsg);
             ReadFile(FHandle, InMsg, InMsg.Size, InMsg.Size, nil);
             OutMsg := InMsg;
             Request := StrPas(InMsg.Data);
             //
             if Pos('calc', Request) > 0 then
               begin
                 if assigned(FWorkFunction) then
                   begin
                    FRequest := Request;
                    Synchronize(CallWorkFunction);
                    Response := FResponse;
                   end
                   else Response := 'server app error.';
               end
             else if Request = 'init' then
               begin
                 if assigned(FInitFunction) then Response := FInitFunction
                   else Response := 'OK';
               end
             else if Request = 'server version' then Response :=
ServerVersionStr
             else if Request = 'application version' then Response :=
FetchVersionString
             else if Request = 'hostname' then Response := FetchHostName
             else if Request = 'shutdown' then
               begin
                Terminate;
                Response := 'bye.';
               end
             else Response := 'syntax error.';
             StrPCopy(OutMsg.Data, Response);
             OutMsg.Count := length(Response);
             OutMsg.Size := sizeof(OutMsg.Size) + sizeof(OutMsg.Count) +
OutMsg.Count + 3;
             WriteFile(FHandle, OutMsg, OutMsg.Size, Written, nil);
            finally
             DisconnectNamedPipe(FHandle);
            end;
```

```
        end
        else begin
          Sleep(250);
        end;
    end;
end;



procedure TPipeServer.CallWorkFunction;
begin
 FResponse := FWorkFunction(FRequest);
end;

end.
```

FORTRAN90 example implementation (version 1.0 server)

This FORTRAN example implementation of a named pipe server lacks the implementation of the "init" command and is therefore version 1.0 and not version 1.1 like the reference implementation above. However, as VersaDim does not react on the return value of the "init" command, this will not narrow the usability of the server. The following code has been compiled and tested using Intel Fortran 11.0 Trial Edition.

```
!****************************************************************************
!
!  MODULE: PipeServer
!
!  PURPOSE:  Implementation of a named pipe server version 1.0.
!
!****************************************************************************

module PipeServer

use KERNEL32

implicit none

    ! variables
    type :: TPipeMessage
        integer(4) Size
          integer(4) Count
          character(1024) Text
    end type TPipeMessage
    type(TPipeMessage) :: InMsg, OutMsg

    integer(HANDLE) PipeHandle
    character(255) PipeName
    integer(DWORD) OpenMode, PipeMode, MaxInstances, OutBuffer, InBuffer,
TimeOut
```

```
    type (T_SECURITY_ATTRIBUTES) SecurityAttributes

    integer(BOOL) Response

    integer(LPVOID) Buffer
    integer(DWORD) NumberOfBytesToTransfer
    integer(LPDWORD) NumberOfBytesTransferred

    integer(LPCVOID) WriteBuffer

    character(*), parameter :: ServerVersion = '1.0'

    private :: TPipeMessage, InMsg, OutMsg, PipeHandle, PipeName, OpenMode, 
PipeMode, MaxInstances, OutBuffer, InBuffer, TimeOut, &
               SecurityAttributes, Response, Buffer, NumberOfBytesToTransfer, 
NumberOfBytesTransferred, WriteBuffer, ServerVersion

contains


    function OperatePipeServer (PipeName, UserFunction)
    ! creates a named pipe with name 'PipeName', waits for clients to connect 
and calls 'UserFunction' when received message
    ! starts with 'calc'
        implicit none
        integer                    :: OperatePipeServer
        character(255), intent(in) :: PipeName     ! name of the pipe to be 
created on localhost
        interface                                  ! 'function pointer' to 
user supplied function
            character(1024) function UserFunction (Params)
                character(1024), intent(in) :: Params
            end function
        end interface
        integer :: i, flag
        character(1024) :: Argument

        ! init
        OperatePipeServer = 0
        flag = 0

        ! start pipe server
          OpenMode = PIPE_ACCESS_DUPLEX
          PipeMode = jior(jior(PIPE_TYPE_MESSAGE, PIPE_READMODE_MESSAGE), 
PIPE_WAIT)
          MaxInstances = PIPE_UNLIMITED_INSTANCES
          OutBuffer = sizeof(OutMsg)
          InBuffer  = sizeof(InMsg)
          TimeOut = NMPWAIT_USE_DEFAULT_WAIT
          SecurityAttributes%nLength = 0
          PipeHandle = CreateNamedPipe (PipeName, OpenMode, PipeMode, 
MaxInstances, OutBuffer, InBuffer, TimeOut, SecurityAttributes)

        if (PipeHandle /= INVALID_HANDLE_VALUE) then
        do
            ! wait for client to connect to pipe server
```

```
            flag = 1
              Response = ConnectNamedPipe (PipeHandle, NULL)
              if (Response == 0) exit

              ! now let's see what the client has sent to pipe server
              flag = 2
              Buffer = loc(InMsg)
              NumberOfBytesToTransfer = sizeof(InMsg)
              Response = ReadFile (PipeHandle, Buffer,
NumberOfBytesToTransfer, loc(NumberOfBytesTransferred), NULL)
              if (Response == 0) exit

              ! process the command
              i = index(trim(InMsg%Text), 'calc', .true.)
              if ( i > 0 ) then
                  Argument = InMsg%Text(i+5:)
                  OutMsg%Text = UserFunction(Argument)
              else
                  i = index(trim(InMsg%Text), 'server version', .true.)
                  if ( i > 0) then
                      OutMsg%Text = ServerVersion
                  else
                      i = index(trim(InMsg%Text), 'application version',
.true.)
                      if ( i > 0 ) then
                          OutMsg%Text = GetAppVersion()
                      else
                          i = index(trim(InMsg%Text), 'hostname', .true.)
                          if ( i > 0) then
                              OutMsg%Text = GetHostName()
                          else
                              i = index(trim(InMsg%Text), 'shutdown', .true.)
                              if ( i > 0) then
                                  OutMsg%Text = 'bye.'
                              else
                                  OutMsg%Text = 'syntax error.'
                              end if
                          end if
                      end if
                  end if
              end if

              ! process the client command and send back response
              flag = 3
              WriteBuffer = loc(OutMsg)
              OutMsg%Count = len_trim(OutMsg%Text)
              NumberOfBytesToTransfer = sizeof(OutMsg%Size) +
sizeof(OutMsg%Count) + OutMsg%Count + 3
              Response = WriteFile (PipeHandle, WriteBuffer,
NumberOfBytesToTransfer, loc(NumberOfBytesTransferred), NULL)
              if (Response == 0) exit

              ! finish transaction
              flag = 4
              Response = DisconnectNamedPipe (PipeHandle)
              if ( (Response == 0) .or. (OutMsg%Text == 'bye.') ) exit
```

```
        end do
        i = CloseHandle(PipeHandle)
        end if

        ! update return value
        if (Response == 0) Response = GetLastError()
        OperatePipeServer = 256*Response + flag
    end function



    function GetAppVersion()
    ! retrieves file version information from this executable and returns
content of 'FileVersion' key
    ! add 'version' resource to project in order to use this feature
        use dfwin
        implicit none
        character(256) :: GetAppVersion
        logical(4) bret
        integer(4) iret
        integer    dwVerHnd
        integer    dwVerInfoSize
        integer    uVersionLen
        integer(4) lpstrVffInfo
        integer(4) hMem
        character(256) szFullPath
        character(256) szGetName
        character(256) lpVersion

        ! set default return value
        GetAppVersion = '?'

        iret = GetModuleFileName (null, szFullPath, len(szFullPath))
        dwVerInfoSize = GetFileVersionInfoSize (szFullPath, loc(dwVerHnd))

        if (dwVerInfoSize /= 0) then
            hMem = GlobalAlloc (GMEM_MOVEABLE, INT(dwVerInfoSize))
            lpstrVffInfo = GlobalLock(hMem)
            iret = GetFileVersionInfo (szFullPath, dwVerHnd, dwVerInfoSize,
lpstrVffInfo)
            if (iret /= 0) then
                szGetName = "\\StringFileInfo\\040004b0\\FileVersion"C
                bret = VersionQueryValue (lpstrVffInfo, loc(szGetName),
loc(lpVersion), loc(uVersionLen))
                if (bret /= 0) GetAppVersion = lpVersion
            end if
            iret = GlobalUnlock(hMem)
            iret = GlobalFree(hMem)
        end if
    end function



    function GetHostName()
    ! gets and returns host name
        implicit none
```

```
        character(256)     :: GetHostName

        GetHostName = '?'
        call hostnm(GetHostName)
    end function



    function GetArgValue (Argument, iArg)
    ! parses Argument for the iArg argument, converts to real and delivers
result
        implicit none
        real                      :: GetArgValue
        character(*), intent(in) :: Argument
        integer, intent(in)      :: iArg
        integer                   :: Nr
        character(1024)           :: Arg
        integer                   :: i, j, k

        ! init
        GetArgValue = 1.0
        Arg = adjustl(Argument)  ! get rid of leading blanks
        if (iArg < 1) then       ! only positive values for iArg are allowed
            Nr = 1
        else
            Nr = iArg
        end if

        ! loop Argument character-by-character looking for blanks
        i = 0
        do k = 1, Nr
            i = i + 1
            if (i > 1024) then       ! we are running out of arguments so just
return the last one that was found
                i = i - 1
                exit
            end if
            j = i
            do while( (Arg(i:i) /= ' ') .and. (Arg(i:i) /= achar(0)) )
                i = i + 1
                if (i > 1024) then   ! we have reached the end of the
'Argument' string
                    i = 1024         !    looking for a blank in vain - so
let's exit both loops...
                    exit
                end if
            end do
        end do
        i = i - 1

        ! convert to real and deliver result
        read (Arg(j:i), *) GetArgValue
    end function

end module
```

Search algorithms

Gradient descent

Given an initial parameter set, the partial deviation of the cost function with respect to each parameter is determined numerically. This gives the search direction. Along this search direction a new parameter set is sought that delivers an extreme cost function value and is within the parameter bounds.
Normally, the initial parameter set is chosen randomly. However, the search can also start with some predefined parameter set. This is called "resume".

Nelder-Mead simplex algorithm

The idea is that each parameter set describes a point in multidimensional space. The algorithm then works with a set of parameter sets, i.e. a "cloud" of points. By repeatedly applying heuristic rules to the set of points, thereby replacing "bad" points (i.e. parameter sets with non-extreme cost function values) with better ones, the points cloud moves through space and converges to a valley or summit of the cost function.

Details can be found in:
Press, W. H., Teukolsky, S. A., Vettering, W. T., Flannery, B. P., 2002. *"Numerical recipes in C++"*, 2nd Edition. Cambridge University Press

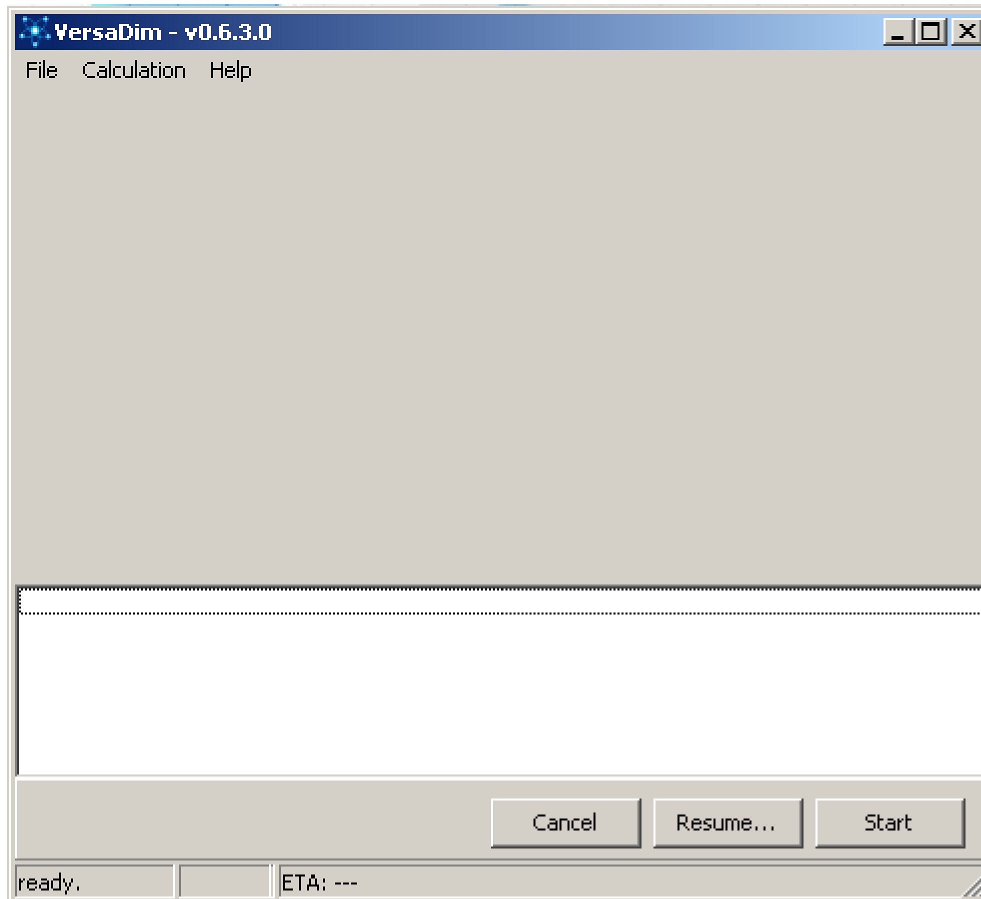
Simulated annealing for continuous control space

This algorithm works similar to the Nelder-Mead simplex algorithm except for the decisive detail that cost function values are not deterministic but have a probabilistic part added to them. This probabilistic part is to model "thermal noise" in an analogy to annealing processes found in nature. At the beginning of each search run, the probabilistic part added to the cost function values is high, analog to high temperatures leading to noise. As the search progresses, the probabilistic part is decreased and the "pure" cost function value determines the search direction. This is to make the algorithm less likely to converge to a poor local extremum.

Details can be found in:
Press, W. H., Teukolsky, S. A., Vettering, W. T., Flannery, B. P., 2002. *"Numerical recipes in C++"*, 2nd Edition. Cambridge University Press

User interface description

Main dialog



Main menu

Via the main menu several actions can be issued. These comprise opening and saving configuration files, managing search runs and invoking the help...

Buttons

Cancel: Will cancel a search run currently under way (note: VersaDim will wait for the current epoch of the search to be finished before accepting new user input, this may need some time).

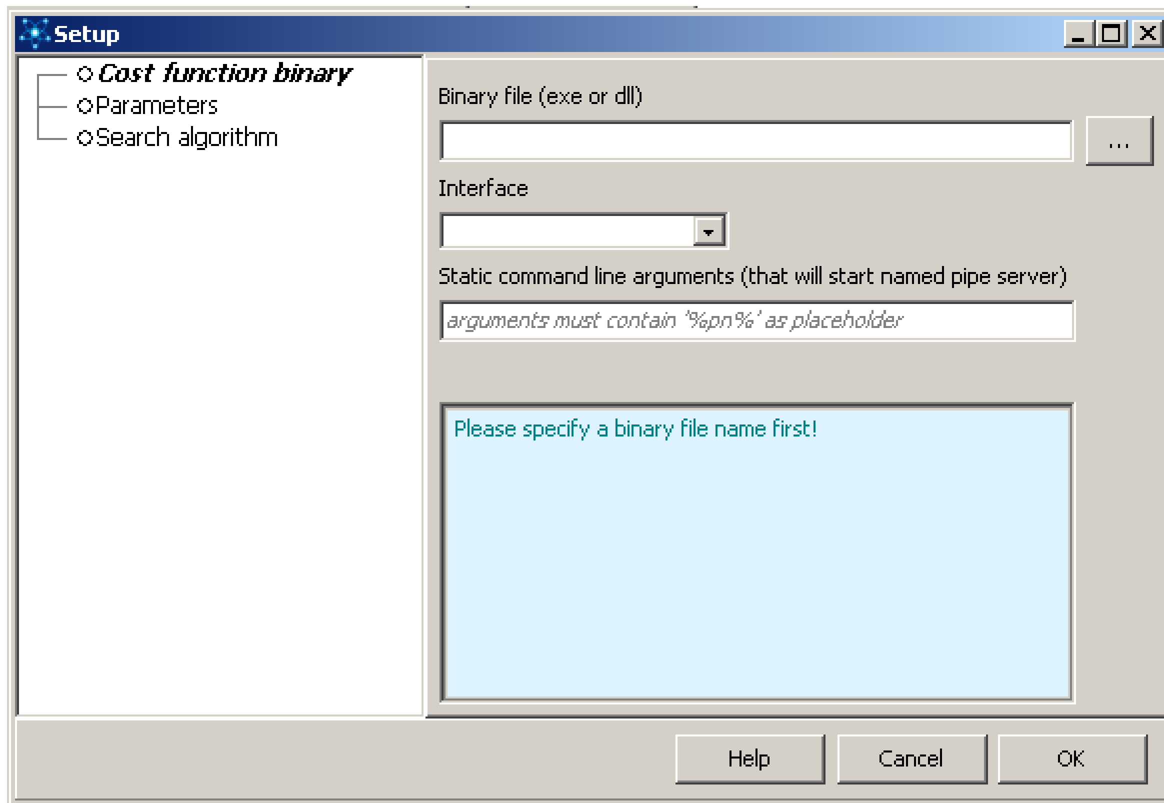
Resume... Starts a file dialog in which a result file can be choosed. The last entry of the result file, i.e. the best parameter set found so far, will be used as initial value for the gradient descent algorithm (note that this is not available for the other search algorithms).

Start: Will start the search process as configured.

Status bar

The left panel indicates the state VersaDim is in (search run under way or not). The right panel gives an estimation when the current search run will be finished ("expected time of arrival").

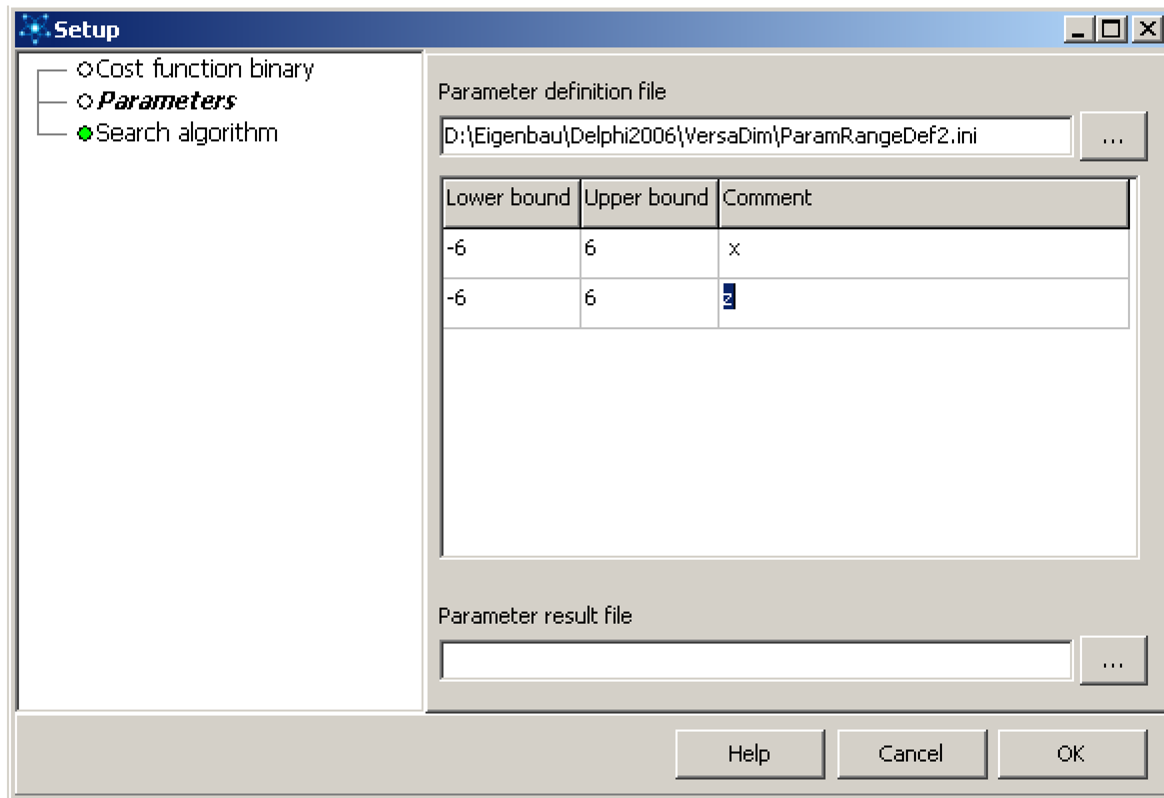
Setup dialog



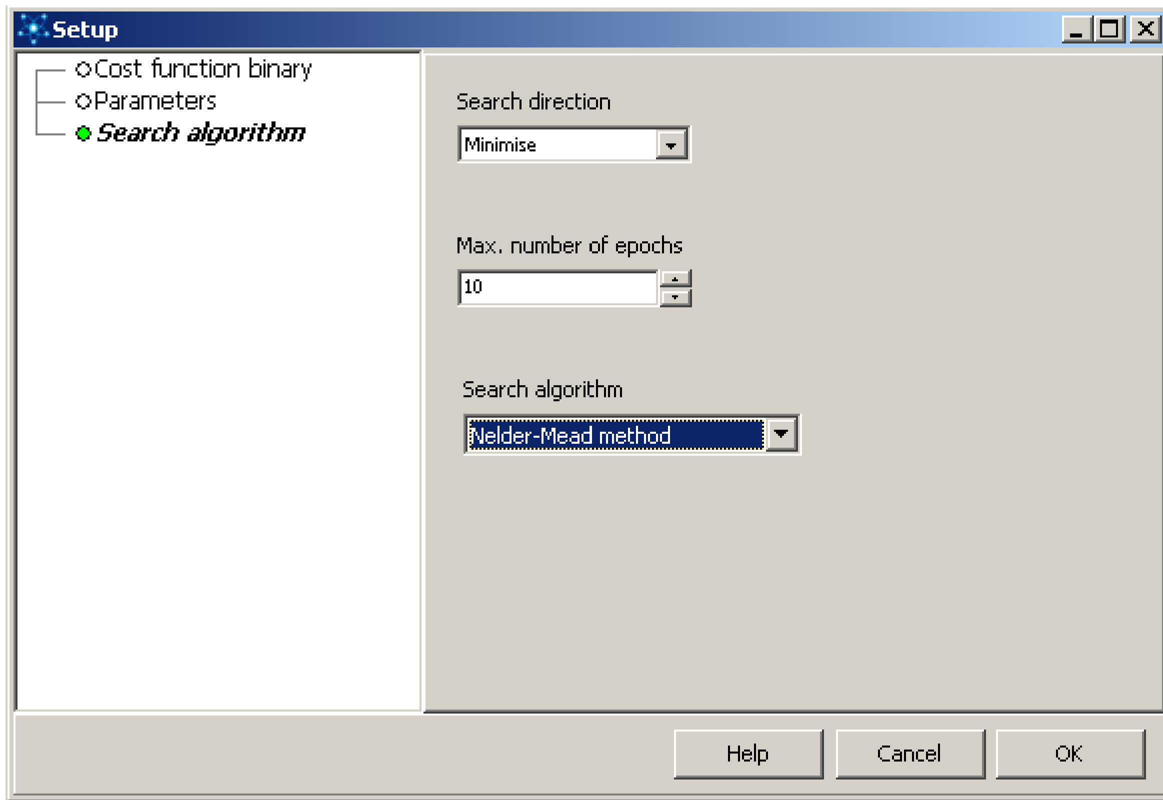
Via the setup dialog users can configure how VersaDim should communicate with which external application (or library) to demand cost function evaluations. Moreover, it can be specified what the parameters to be varied are and within which bounds values should be searched for. Finally, a search algorithm can be chosen, too.

The "**Cost function binary**" tab offers a text box which can take the name of a binary file chosen via the open-file-dialog accessible through the button right of the box. Using the combobox below the text box, the communication interface can be selected (this must be either "DLL" or "Named PIPE".
More interesting is the static command line argument text box below. It is only available when the "Named PIPE" interface has been selected and offers a mechanism for VersaDim to tell the external application that it is expected to start and operate a named pipe server soon. Note that the external application might not only implement such a server for VersaDim but instead may implement a wealth of other functionality as well. What is even more, VersaDim needs to have the possibility to tell the external application the name of the named pipe server that must be created. Therefore, "%pn%" must be contained in the static command line argument (without the quotes). So, if the external application takes a command line argument "-PipeName" followed by the pipe name, i.e. VersaDim is supposed to invoke the external

application "MyApp" to prepare it for repeated cost function evaluations like this:
MyApp -PipeName MyPipeName,
the static command line argument that must be inserted into the text box would be:
-PipeName %pn%
VersaDim will then replace %pn% with the pipe name it will internally work with and invoke the external application accordingly.
Right clicking the blue box below pops up a menu that offers a command to check the interface. VersaDim will then send commands and echo the results. The comments that are printed out to the blue box may help in finding a correct configuration.



The "**Parameters**" tab deals with the parameters VersaDim is to vary. Choosing a parameter definition file by clicking on the button in the upper right corner of the dialog and answering the file open dialog will display and populate the string grid in the central part of the dialog. The parameters may be edited. When changes are made, a confirmation dialog will be displayed after clicking on "OK" asking if the modified parameter list shall be written to the parameter file.
With the text box at the bottom the user can specify the parameter result file. VersaDim will write parameter values and resulting cost function values for each cost function evaluation to this file using one line per evaluation. After the search run, this file will contain the best parameter set in the last line.



The "**Search algorithm**" tabs allows to configure the actual search run. In the "Search direction" combo box it can be selected whether a maximum or a minimum of the cost function is to be searched for. Clicking on the up/down-arrows below users can set how many epochs are to be calculated. One epoch generally is one complete search algorithm step, i.e. the calculation of the next best parameter set. Finally, the search algorithm to be used can be selected. Users can choose between gradient descent, Nelder-Mead and simulated annealing algorithm.

File format description

Parameter file

Purpose
The parameter configuration file is used to define number, order and limits of the parameters to be varied.

File name
No restrictions as to the file name have to be considered.

File content
The parameter configuration file is a text file. Each parameter occupies one line. The parameters will be transferred to the cost function binary in the order listed in this file (top line as leftmost function argument or string entry). "'#" (without quotes) serves as comment character. This means that everything behind and including the "#" character is ignored when numeric values are to be accessed.
Upper and lower bounds of parameters are entered as numeric values separated by a blank, TAB (#9) or ",". The decimal separator sign must be ".".

Example parameter file
In the following, an example parameter definition file is given:

```
-------------- begin listing -------------------------------
# Parameter count and range definition file;
# created by VersaDim version 0.6.3.0
# date: 19.01.2011 11:05:47
#
-5.2    6       #  x
-6      8.1     #  y
-------------- end listing -------------------------------
```

This would define a parameter called "x" which will be varied in the interval [-5.2, 6.0] and a parameter "y" element of the interval [-6.0, 8.1]. When DLL is configured as communication interface, VersaDim would call the calc-function like this: calc(x,y). In the case of communication over the named pipe interface, VersaDim would compose a command string that would for example look like "calc -4.654 8.0".

Configuration file

Purpose
The configuration file is used to define the settings that can be made via the setup dialog

File name
No restrictions as to the file name have to be considered.

File content
The configuration file follows the commonly used ini-file structure. It consists of sections filled with key/value-pairs.

The following keys are used:

Section "Cost function"			
FileName	string	name of binary to be consulted for cost function evaluation	FileName=D:\Programs\Delphi2006\HelioTest\HelioTest.exe
StaticArguments	string	the static command line arguments (must contain "%pn%")	StaticArguments=-OperatePipeServer %pn% -ProjectFileName Projects\MyProject.hlx
ParamDefinitionFileName	string	file name of the parameter definition file	ParamDefinitionFileName=D:\Projects\Delphi2006\HelioTest\ParameterDefinition.ini
Interface	string	must be one of the following: • NamedPipe • DLL • undefined	Interface=NamedPipe
SearchDirection	string	must be one of the following: • maximise • minimise	SearchDirection=maximise
Section "Solver"			
MaxEpochs	integer	range: 10..10000	MaxEpochs=10
SearchAlgorithm	string	must be one of the following • NumericalGradientDescent • Nelder-Mead • SimulatedAnnealing	SearchAlgorithm=NumericalGradientDescent
ResultFile	string	name of the result file	ResultFile=D:\Data\Test\ParameterResults.dat

Configuration files can be easily created by choosing "File -> Save configuration as..." from the main form's main menu.