**Курсовая работа:**

**«Программная реализация модального управления для линейных стационарных систем»**

**Постановка задачи:**

1. Для объекта управления с математическим описанием

, (1) - задано,

где  - n-мерный вектор состояния, ,

- начальный вектор состояния,

- скалярное управление,

- матрица действительных коэффициентов,

- матрица действительных коэффициентов,

найти управление в функции переменных состояния объекта, т.е.

, (2)

где- матрица обратной связи, такое, чтобы замкнутая система была устойчивой.

2. Корни характеристического уравнения замкнутой системы

 (3)

должны выбираться по усмотрению (произвольно) с условием устойчивости системы (3).

**Задание:**

1. Разработать алгоритм решения поставленной задачи.

2. Разработать программу решения поставленной задачи с интерактивным экранным интерфейсом в системах Borland Pascal, Turbo Vision, Delphi - по выбору.

3. Разработать программу решения систем дифференциальных уравнений (1) и (3) с интерактивным экранным интерфейсом.

4. Разработать программу графического построения решений систем (1) и (3) с интерактивным экранным интерфейсом.

**Введение**

Наряду с общими методами синтеза оптимальных законов управления для стационарных объектов всё большее примене­ние находят методы, основанные на решении задачи о размеще­нии корней характеристического уравнения замкнутой системы в желаемое положение. Этого можно добиться надлежащим выбором матрицы обратной связи по состоянию. Решение ука­занной задачи является предметом теории *модального управ­ления* (термин связан с тем, что корням характеристического уравнения соответствуют составляющие свободного движения, называемые *модами).*

**Алгоритм модального управления.**

**Соглашения:**

* Задаваемый объект управления математически описывается уравнением

, (1)

где  и  - матрицы действительных коэффициентов,

 - n-мерный вектор состояния

- скалярное управление,

 - порядок системы (1).

* Обратная связь по состоянию имеет вид

, (2)

где- матрица обратной связи.

* Система с введенной обратной связью описывается уравнением

 (3)

* Характеристическое уравнение системы (1) имеет вид

 (4)

* Характеристическое уравнение системы (3) с задаваемыми (желаемыми) корнями имеет вид

 (5)

**Алгоритм:**

1. Для исходной системы (1) составляем матрицу управляемости



2. Обращаем матрицу , т.е. вычисляем .

Если  не существует (т.е. матрица  - вырожденная), то прекращаем вычисления: полное управление корнями характеристического уравнения (5) не возможно.

3. Вычисляем матрицу 

4. Составляем матрицу



5. Вычисляем матрицу, обратную матрице , т.е. 

6. Вычисляем матрицу  - матрицу  в канонической форме фазовой переменной:



где - коэффициенты характеристического уравнения (4).

Матрица  в канонической форме имеет вид



7. Составляем вектор  , элементам которого являются коэффициенты характеристического уравнения (4), т.е. , ,

где  - элементы матрицы .

8. Находим коэффициенты характеристического уравнения (5) (см. пояснения) и составляем из них вектор .

9. Вычисляем вектор .

 - искомая матрица обратной связи системы (3), но она вычислена для системы, матрицы которой заданы в канонической форме фазовой переменной ( и ).

10. Для исходной системы (3) матрица обратной связи получается по формуле



Матрица  - искомая матрица обратной связи.

**Пояснения к алгоритму:**

В данной работе рассматривается случай, когда управление единственно и информация о переменных состояния полная. Задача модального управления тогда наиболее просто решается, если уравнения объекта заданы в канонической форме фазовой переменной.

Так как управление выбрано в виде линейной функции переменных состояния , где  является матрицей строкой . В таком случае уравнение замкнутой системы приобретает вид . Здесь





Характеристическое уравнение такой замкнутой системы будет следующим



Поскольку каждый коэффициент матрицы обратной связи  входит только в один коэффициент характеристического уравнения, то очевидно, что выбором коэффициентов  можно получить любые коэффициенты характеристического уравнения, а значит и любое расположение корней.

Если же желаемое характеристическое уравнение имеет вид

,

то коэффициенты матрицы обратной связи вычисляются с помощью соотношений:



Если при наличии одного управления нормальные уравнения объекта заданы не в канонической форме (что наиболее вероятно), то, в соответствии с пунктами №1-6 алгоритма, от исходной формы с помощью преобразования  или  нужно перейти к уравнению  в указанной канонической форме.

Управление возможно, если выполняется условие полной управляемости (ранг матрицы управляемости *M* должен быть равен *n*). В алгоритме об управляемости системы судится по существованию матрицы : если она существует, то ранг матрицы равен ее порядку (*n*). Для объекта управления с единственным управлением матрица  оказывается также единственной.

Для нахождения коэффициентов  характеристического уравнения (5), в работе используется соотношения между корнями  и коэффициентами  линейного алгебраического уравнения степени *n*:

, (*k = 1, 2, ... , n*)

где многочлены - элементарные симметрические функции, определяемые следующим образом:

,

,

,

**...**



где *Sk* - сумма всех  произведений, каждое из которых содержит *k* сомножителей *xj* с несовпадающими коэффициентами.

**Программная реализация алгоритма.**

Текст программной реализации приведен в ПРИЛОЖЕНИИ №1. Вот несколько кратких пояснений.

* Программа написана на языке Object Pascal при помощи средств Delphi 2.0, и состоит из следующих основных файлов:

KursovayaWork.dpr

MainUnit.pas

SubUnit.pas

Matrix.pas

Operates.pas

HelpUnit.pas

OptsUnit.pas

* KursovayaWork.dpr - файл проекта, содержащий ссылки на все формы проекта и инициализирующий приложение.
* В модуле MainUnit.pas находится описание главной формы приложения, а также сконцентрированы процедуры и функции, поддерживаюшие нужный интерфейс программы.
* Модули SubUnit.pas и Operates.pas содержат процедуры и функции, составляющие смысловую часть программной реализации алгоритма, т.е. процедуры решения задачи модально управления, процедуры решения систем дифференциальных уравнений, процедуры отображения графиков решений систем и т.д. Там также находятся процедуры отображения результатов расчетов на экран.
* В модуле Matrix.pas расположено описание класса TMatrix - основа матричных данных в программе.
* Модули HelpUnit.pas и OptsUnit.pas носят в программе вспомогательный характер.
* Для решения систем дифференциальных уравнений использован метод Рунге-Кутта четвертого порядка точности с фиксированным шагом. Метод был позаимствован из пакета программ NumToolBox и адаптирован под новую модель матричных данных.
* Обращение матриц производится методом исключения по главным диагональным элементам (метод Гаусса). Этот метод так же был позаимствован из NumToolBox и соответствующе адаптирован.

**Пориложение.**

**program KursovayaWork;**

uses

Forms,

MainUnit in 'MainUnit.pas' {Form\_Main},

OptsUnit in 'OptsUnit.pas' {Form\_Options},

SubUnit in 'SubUnit.pas',

Matrix in 'Matrix.pas',

Operates in 'Operates.pas',

HelpUnit in 'HelpUnit.pas' {Form\_Help};

{$R \*.RES}

begin

Application.Initialize;

Application.Title := 'Модальное управление';

Application.CreateForm(TForm\_Main, Form\_Main);

Application.CreateForm(TForm\_Options, Form\_Options);

Application.CreateForm(TForm\_Help, Form\_Help);

Application.Run;

end.

**unit MainUnit;**

interface

uses

Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,

ComCtrls, Tabnotbk, Menus, StdCtrls, Spin, ExtCtrls, Buttons, Grids,

OleCtrls, VCFImprs, GraphSvr, ChartFX {, ChartFX3};

type

TForm\_Main = class(TForm)

BevelMain: TBevel;

TabbedNotebook\_Main: TTabbedNotebook;

SpinEdit\_Dim: TSpinEdit;

BitBtn\_Close: TBitBtn;

BitBtn\_Compute: TBitBtn;

StringGrid\_Ap0: TStringGrid;

StringGrid\_Anp0: TStringGrid;

StringGrid\_Roots: TStringGrid;

StringGrid\_Kpp0: TStringGrid;

StringGrid\_Bp0: TStringGrid;

RadioGroup\_RootsType: TRadioGroup;

Label\_A1p0: TLabel;

Label\_Ap0: TLabel;

Label\_mBp0: TLabel;

Label\_Roots: TLabel;

Label\_Kpp0: TLabel;

BevelLine: TBevel;

Label\_Dim: TLabel;

StringGrid\_Ap1: TStringGrid;

StringGrid\_Bp1: TStringGrid;

Label\_Ap1: TLabel;

Label\_Bp1: TLabel;

StringGrid\_Kpp1: TStringGrid;

Label\_Kpp1: TLabel;

StringGrid\_InCond: TStringGrid;

Label\_InCond: TLabel;

Label\_U: TLabel;

Edit\_U: TEdit;

BitBtn\_Options: TBitBtn;

BitBtn\_Help: TBitBtn;

StringGrid\_ABKpp1: TStringGrid;

Label\_ABKpp1: TLabel;

Edit\_W: TEdit;

Label\_w: TLabel;

RadioGroupChart: TRadioGroup;

ChartFX: TChartFX;

LabelW1: TLabel;

StringGrid\_Solve1: TStringGrid;

StringGrid\_Solve2: TStringGrid;

Label1: TLabel;

Label2: TLabel;

Label3: TLabel;

procedure BitBtn\_CloseClick(Sender: TObject);

procedure BitBtn\_OptionsClick(Sender: TObject);

procedure BitBtn\_ComputeClick(Sender: TObject);

procedure FormCreate(Sender: TObject);

procedure SpinEdit\_DimChange(Sender: TObject);

procedure StringGrid\_RootsSetEditText(Sender: TObject; ACol,

ARow: Longint; const Value: string);

procedure RadioGroup\_RootsTypeClick(Sender: TObject);

procedure TabbedNotebook\_MainChange(Sender: TObject; NewTab: Integer;

var AllowChange: Boolean);

procedure StringGrid\_SetEditText(Sender: TObject; ACol,

ARow: Longint; const Value: string);

procedure BitBtn\_HelpClick(Sender: TObject);

procedure RadioGroupChartClick(Sender: TObject);

private

procedure FillFixedCellsInAllGrids;

procedure FillCellsInAllGrids;

public

procedure BindGrids;

procedure UnBindGrids;

end;

var

Form\_Main: TForm\_Main;

implementation

uses Matrix, SubUnit, OptsUnit, Operates, CFXOCX2, HelpUnit;

const

DefOptions = [goFixedVertLine, goFixedHorzLine,

goVertLine, goHorzLine,

goColSizing, goEditing,

goAlwaysShowEditor, goThumbTracking];

{$R \*.DFM}

procedure TForm\_Main.FillFixedCellsInAllGrids;

var

Order : TOrder;

i: byte;

Str: string;

begin

Order := SpinEdit\_Dim.Value;

for i := 1 to Order do

begin

Str := IntToStr(i);

StringGrid\_Ap0.Cells[0, i] := Str;

StringGrid\_Ap0.Cells[i, 0] := Str;

StringGrid\_Bp0.Cells[0, i] := Str;

StringGrid\_ANp0.Cells[i, 0] := Str;

StringGrid\_ANp0.Cells[0, i] := Str;

StringGrid\_Roots.Cells[i, 0] := Str;

StringGrid\_Kpp0.Cells[i, 0] := Str;

StringGrid\_Ap1.Cells[0, i] := Str;

StringGrid\_Ap1.Cells[i, 0] := Str;

StringGrid\_Bp1.Cells[0, i] := Str;

StringGrid\_ABKpp1.Cells[i, 0] := Str;

StringGrid\_ABKpp1.Cells[0, i] := Str;

StringGrid\_InCond.Cells[i, 0] := Str;

StringGrid\_Kpp1.Cells[i, 0] := Str;

StringGrid\_Solve1.Cells[i, 0] := 'X' + IntToStr(i);

StringGrid\_Solve2.Cells[i, 0] := 'X' + IntToStr(i);

StringGrid\_Solve1.Cells[0, 0] := 'Время';

StringGrid\_Solve2.Cells[0, 0] := 'Время';

end;

end;

procedure TForm\_Main.FillCellsInAllGrids;

var

Order : TOrder;

i, j : byte;

begin

Order := SpinEdit\_Dim.Value;

for i := 1 to Order do

for j := 1 to Order do

begin

StringGrid\_Ap0.Cells[j, i] := '0';

StringGrid\_Ap0.Cells[i, i] := '1';

StringGrid\_Bp0.Cells[1, i] := '0';

StringGrid\_Roots.Cells[i, 1] := '-1';

StringGrid\_Roots.Cells[i, 2] := '0';

StringGrid\_Kpp0.Cells[i, 1] := '0';

StringGrid\_Ap1.Cells[j, i] := '0';

StringGrid\_Ap1.Cells[i, i] := '1';

StringGrid\_Bp1.Cells[1, i] := '0';

StringGrid\_ABKpp1.Cells[j, i] := '0';

StringGrid\_ABKpp1.Cells[i, i] := '1';

StringGrid\_InCond.Cells[i, 1] := '0';

StringGrid\_Kpp1.Cells[i, 1] := '0';

end;

FillFixedCellsInAllGrids;

StringGrid\_Roots.Cells[0, 1] := 'Re';

StringGrid\_Roots.Cells[0, 2] := 'Im';

StringGrid\_Bp1.Cells[1, 0] := '1';

StringGrid\_Bp0.Cells[1, 0] := '1';

end;

procedure TForm\_Main.BindGrids;

begin

CopyGrid(StringGrid\_Ap1, StringGrid\_Ap0);

CopyGrid(StringGrid\_Bp1, StringGrid\_Bp0);

CopyGrid(StringGrid\_Kpp1, StringGrid\_Kpp0);

StringGrid\_Ap1.Options := DefOptions - [goEditing];

StringGrid\_Bp1.Options := DefOptions - [goEditing];

StringGrid\_Kpp1.Options := DefOptions - [goEditing];

end;

procedure TForm\_Main.UnBindGrids;

begin

StringGrid\_Ap1.Options := DefOptions;

StringGrid\_Bp1.Options := DefOptions;

StringGrid\_Kpp1.Options := DefOptions;

end;

procedure TForm\_Main.BitBtn\_CloseClick(Sender: TObject);

begin

Close;

end;

procedure TForm\_Main.BitBtn\_OptionsClick(Sender: TObject);

var

V0, V1, V2, V3: LongInt;

LS: TCheckBoxState;

begin

with Form\_Options do

begin

V0 := SpinEdit0.Value;

V1 := SpinEdit1.Value;

V2 := SpinEdit2.Value;

V3 := SpinEdit3.Value;

LS := CheckBox\_Link.State;

ShowModal;

if ModalResult = mrCancel then

begin

SpinEdit0.Value := V0;

SpinEdit1.Value := V1;

SpinEdit2.Value := V2;

SpinEdit3.Value := V3;

CheckBox\_Link.State := LS;

end

else

if ((SpinEdit0.Value <> V0) or (SpinEdit1.Value <> V1)) or

((SpinEdit2.Value <> V2) or (SpinEdit3.Value <> V3)) then

begin

BitBtn\_Compute.Enabled := True;

case BitBtn\_Compute.Tag of

4, 5 :BitBtn\_Compute.Tag := BitBtn\_Compute.Tag - 4;

6, 7 :BitBtn\_Compute.Tag := BitBtn\_Compute.Tag - 4;

8, 9 :BitBtn\_Compute.Tag := BitBtn\_Compute.Tag - 8;

10, 11 :BitBtn\_Compute.Tag := BitBtn\_Compute.Tag - 8;

12, 13 :BitBtn\_Compute.Tag := BitBtn\_Compute.Tag - 12;

14, 15 :BitBtn\_Compute.Tag := BitBtn\_Compute.Tag - 12;

end;

end;

end;

end;

procedure TForm\_Main.BitBtn\_ComputeClick(Sender: TObject);

begin

BitBtn\_Compute.Enabled := False;

if Form\_Options.CheckBox\_Link.State = cbChecked then BindGrids;

case TabbedNotebook\_Main.PageIndex of

0 : begin

ComputeFromPage0;

BitBtn\_Compute.Tag := BitBtn\_Compute.Tag + 1;

end;

1 : begin

ComputeFromPage1;

ShowChart(Succ(RadioGroupChart.ItemIndex));

BitBtn\_Compute.Tag := BitBtn\_Compute.Tag + 14;

end;

2 : begin

ComputeFromPage2;

BitBtn\_Compute.Tag := BitBtn\_Compute.Tag + 4;

end;

3 : begin

ComputeFromPage3;

BitBtn\_Compute.Tag := BitBtn\_Compute.Tag + 8;

end;

end;

end;

procedure TForm\_Main.FormCreate(Sender: TObject);

const

FirstColWidth = 20;

begin

StringGrid\_Ap0.ColWidths [0] := FirstColWidth;

StringGrid\_Anp0.ColWidths [0] := FirstColWidth;

StringGrid\_Bp0.ColWidths [0] := FirstColWidth;

StringGrid\_Roots.ColWidths [0] := FirstColWidth;

StringGrid\_Ap1.ColWidths [0] := FirstColWidth;

StringGrid\_ABKpp1.ColWidths [0] := FirstColWidth;

StringGrid\_Bp1.ColWidths [0] := FirstColWidth;

StringGrid\_Kpp0.ColWidths [0] := FirstColWidth;

StringGrid\_Kpp1.ColWidths [0] := FirstColWidth;

StringGrid\_InCond.ColWidths [0] := FirstColWidth;

FillCellsInAllGrids;

BindGrids;

end;

procedure TForm\_Main.SpinEdit\_DimChange(Sender: TObject);

var

Order: byte;

begin

Order := Succ(SpinEdit\_Dim.Value);

StringGrid\_Ap0.ColCount := Order;

StringGrid\_Ap0.RowCount := Order;

StringGrid\_Anp0.ColCount := Order;

StringGrid\_Anp0.RowCount := Order;

StringGrid\_Bp0.RowCount := Order;

StringGrid\_Roots.ColCount := Order;

StringGrid\_Kpp0.ColCount := Order;

StringGrid\_Ap1.ColCount := Order;

StringGrid\_Ap1.RowCount := Order;

StringGrid\_Bp1.RowCount := Order;

StringGrid\_ABKpp1.ColCount := Order;

StringGrid\_ABKpp1.RowCount := Order;

StringGrid\_InCond.ColCount := Order;

StringGrid\_Kpp1.ColCount := Order;

FillFixedCellsInAllGrids;

BitBtn\_Compute.Enabled := True;

end;

procedure TForm\_Main.StringGrid\_RootsSetEditText(Sender: TObject; ACol,

ARow: Longint; const Value: string);

var

Val : string;

begin

if (ARow = 2) and (Value <> '') then

begin

Val := StringGrid\_Roots.Cells [ACol, ARow];

if StrToFloat (Value) <> 0 then

StringGrid\_Roots.Cells[Succ(ACol),ARow]:=FloatToStr(- StrToFloat(Value));

if StrToFloat (Value) = 0 then

StringGrid\_Roots.Cells [Succ(ACol),ARow] := FloatToStr(0);

end;

end;

procedure TForm\_Main.RadioGroup\_RootsTypeClick(Sender: TObject);

var

Order: TOrder;

j: byte;

NHalf: byte;

StartAlfa, NAlfa, dAlfa: Float;

W: Float;

begin

Order := SpinEdit\_Dim.Value;

W := StrToFloat (Edit\_W.Text);

case RadioGroup\_RootsType.ItemIndex of

0 :StringGrid\_Roots.Options := DefOptions;

1 :begin

for j := 1 to Order do

begin

StringGrid\_Roots.Cells [j, 1] := FloatToStr (-W);

StringGrid\_Roots.Cells [j, 2] := '0';

StringGrid\_Roots.Options := DefOptions - [goEditing];

end

end;

2 :begin

dAlfa := Pi / Order;

StartAlfa := Pi/2 - dAlfa/2;

NHalf := Order div 2;

for j := 1 to NHalf do

begin

NAlfa := StartAlfa + dAlfa \* j;

StringGrid\_Roots.Cells [j, 1] := FloatToStr (Cos (NAlfa) \* W);

StringGrid\_Roots.Cells [Order - Pred (j), 1] := FloatToStr (Cos (-NAlfa) \* W);

StringGrid\_Roots.Cells [j, 2] := FloatToStr (Sin (NAlfa) \* W);

StringGrid\_Roots.Cells [Order - Pred (j), 2] := FloatToStr (Sin (-NAlfa) \* W);

end;

if Odd (Order) then

begin

StringGrid\_Roots.Cells [NHalf +1, 1] := FloatToStr (-W);

StringGrid\_Roots.Cells [NHalf +1, 2] := '0';

end;

StringGrid\_Roots.Options := DefOptions - [goEditing];

end;

end;

end;

procedure TForm\_Main.TabbedNotebook\_MainChange(Sender: TObject;

NewTab: Integer; var AllowChange: Boolean);

begin

with BitBtn\_Compute do

case NewTab of

0 :begin

SpinEdit\_Dim.Enabled := True;

if Tag in [1, 3, 5, 7, 9, 11, 13, 15] then Enabled := False

else Enabled := True;

BitBtn\_Compute.Caption := 'Рассчитать модальное управление';

end;

1 :begin

SpinEdit\_Dim.Enabled := True;

if Tag in [2, 3, 6, 7, 10, 11, 14, 15] then Enabled := False

else Enabled := True;

BitBtn\_Compute.Caption := 'Решить системы дифф. уравнений ';

if Form\_Options.CheckBox\_Link.State = cbChecked then BindGrids;

end;

2 :begin

SpinEdit\_Dim.Enabled := False;

if Tag in [4, 5, 6, 7, 12, 13, 14, 15] then Enabled := False

else Enabled := True;

BitBtn\_Compute.Caption := 'Обновить результаты решений ';

end;

3 :begin

SpinEdit\_Dim.Enabled := False;

if Tag in [8, 9, 10, 11, 12, 13, 14, 15] then Enabled := False

else Enabled := True;

BitBtn\_Compute.Caption := 'Обновить диаграмму решения ';

end;

end;

end;

procedure TForm\_Main.StringGrid\_SetEditText(Sender: TObject; ACol,

ARow: Longint; const Value: string);

begin

if not BitBtn\_Compute.Enabled then

case TabbedNotebook\_Main.PageIndex of

0 :if Form\_Options.CheckBox\_Link.State = cbChecked then

BitBtn\_Compute.Tag := BitBtn\_Compute.Tag - 3

else

BitBtn\_Compute.Tag := BitBtn\_Compute.Tag - 1;

1 :BitBtn\_Compute.Tag := BitBtn\_Compute.Tag - 2;

end;

BitBtn\_Compute.Enabled := True;

end;

procedure TForm\_Main.BitBtn\_HelpClick(Sender: TObject);

begin

Form\_Help.ShowModal;

end;

procedure TForm\_Main.RadioGroupChartClick(Sender: TObject);

begin

case RadioGroupChart.ItemIndex of

0 :ShowChart(1);

1 :ShowChart(2);

end;

end;

end.**unit SubUnit;**

interface

uses

SysUtils, Matrix, Operates, Grids;

procedure CopyGrid(AGrid, BGrid: TStringGrid);

procedure LoadMatrixSolveFromStrGrd (AMatrix: TMatrix; AGrid: TStringGrid);

procedure ComputeFromPage0;

procedure ComputeFromPage1;

procedure ComputeFromPage2;

procedure ComputeFromPage3;

procedure ShowChart(NumberOfChart: Byte);

implementation

uses

MainUnit, OptsUnit, CFXOCX2;

procedure CopyGrid(AGrid, BGrid: TStringGrid);

var

i, j: LongInt;

begin

AGrid.ColCount := BGrid.ColCount;

AGrid.RowCount := BGrid.RowCount;

for j := 0 to AGrid.ColCount do

for i := 0 to AGrid.RowCount do

AGrid.Cells[j, i] := BGrid.Cells[j, i];

end;

function CropStr (Str: String): String;

var

i: Byte;

Str\_1: String;

Begin

for i := Length(Str) downto 1 do

if Str [i] = ' ' then Str := Copy(Str, 1, i-1)

else Break;

Str\_1 := Str;

for i := 1 to Length(Str) do

if Str[i] = ' ' then Str\_1 := Copy(Str, i+1, Length(Str) - i)

else Break;

CropStr := Str\_1;

End;

procedure LoadMatrixFromStrGrd (AMatrix: TMatrix; AGrid: TStringGrid);

var

i, j: Word;

begin

AMatrix.Resize (Pred(AGrid.ColCount), Pred(AGrid.RowCount));

for i := 1 to AMatrix.RowCount do

for j := 1 to AMatrix.ColCount do

begin

if CropStr(AGrid.Cells[j, i]) = '' then AGrid.Cells[j, i] := '0';

AMatrix[j ,i] := StrToFloat(AGrid.Cells[j, i])

end

end;

procedure OutPutMatrixToStrGrd (AMatrix: TMatrix; AGrid: TStringGrid);

var

i, j: Word;

begin

AGrid.ColCount := Succ(AMatrix.ColCount);

AGrid.RowCount := Succ(AMatrix.RowCount);

for i := 1 to AMatrix.RowCount do

for j := 1 to AMatrix.ColCount do

begin

AGrid.Cells[j, 0] := IntToStr (j);

AGrid.Cells[0, i] := IntToStr (i);

AGrid.Cells[j, i] := FloatToStrF(AMatrix[j ,i],ffGeneral,5,3);

end

end;

procedure OutPutMatrixSolveToStrGrd (AMatrix: TMatrix; AGrid: TStringGrid);

var

i, j, k: Word;

begin

AGrid.ColCount := AMatrix.ColCount;

AGrid.RowCount := Succ(AMatrix.RowCount);

for i := 1 to AMatrix.RowCount do

for j := 1 to AMatrix.ColCount do

begin

if j = AMatrix.ColCount then k := 0 else k := j;

AGrid.Cells[j, 0] := 'X' + IntToStr (j);

AGrid.Cells[k, i] := FloatToStrF(AMatrix[j ,i],ffGeneral,5,3);

end;

AGrid.Cells[0, 0] := 'Время';

end;

procedure LoadMatrixSolveFromStrGrd (AMatrix: TMatrix; AGrid: TStringGrid);

var

i, j, k: Word;

begin

AMatrix.Resize (AGrid.ColCount, Pred(AGrid.RowCount));

for i := 1 to AMatrix.RowCount do

for j := 0 to AMatrix.ColCount do

begin

if j = 0 then k := AMatrix.ColCount else k := j;

if CropStr(AGrid.Cells[j, i]) = '' then AGrid.Cells[j, i] := '0';

AMatrix[k ,i] := StrToFloat(AGrid.Cells[j, i])

end

end;

procedure ComputeFromPage0;

var

Order : TOrder;

i, j : byte;

K : ShortInt;

mDummy1, mDummy2,

mA, mB, mKp,

mM, mN, mN1: TMatrix;

cvRoots: TComplexVector;

begin

with Form\_Main do

begin

Order := SpinEdit\_Dim.Value;

mA := TMatrix.Create(Order, Order);

mB := TMatrix.Create(1, Order);

mM := TMatrix.Create(Order, Order);

mDummy1 := TMatrix.Create(Order, Order);

mN1 := TMatrix.Create(Order, 1);

mN := TMatrix.Create(Order, Order);

mDummy2 := TMatrix.Create(Order, Order);

mKp := TMatrix.Create(Order, 1);

LoadMatrixFromStrGrd (mA, StringGrid\_Ap0);

LoadMatrixFromStrGrd (mB, StringGrid\_Bp0);

for j := 1 to Order do

begin

mDummy1.Assign(mA);

mDummy1.NthPower(j - 1);

mDummy1.MultFromRight(mB);

for i := 1 to Order do

mM[j, i] := mDummy1[1, i];

end;

if not mM.Inverse then

Raise ESingularMatrix.Create('Система неполностью управляема:' +

'матрица M - вырожденная !!!'#10 +

'Измените значения коэффициентов матриц А и B');

mN1.SetNull;

mN1[Order, 1] := 1;

mN1.MultFromRight(mM);

for i := 1 to Order do

begin

mDummy2.Assign(mA);

mDummy2.NthPower(i-1);

mDummy1.Assign(mN1);

mDummy1.MultFromRight(mDummy2);

for j := 1 to Order do mN[j, i] := mDummy1[j, 1];

end;

mDummy1.Assign(mN);

if not mDummy1.Inverse then

Raise ESingularMatrix.Create('Не могу обратить матрицу N !!!'#10 +

'(не разбрасывайтесь порядками коэффициентов матриц)');

mA.MultFromLeft(mN);

mA.MultFromRight(mDummy1);

OutPutMatrixToStrGrd(mA, StringGrid\_Anp0);

cvRoots.Dim := Order;

for j := 1 to Order do

begin

cvRoots.Data[j].Re := StrToFloat(StringGrid\_Roots.Cells[j, 1]);

cvRoots.Data[j].Im := StrToFloat(StringGrid\_Roots.Cells[j, 2]);

end;

for j := 1 to Order do

begin

if Odd (j) then K := -1 else K := +1;

mKp[Order-Pred(j), 1] := - mA[Order-Pred(j), Order] -

K \* SymmetricalFunction(cvRoots, j);

end;

mKp.MultFromRight(mN);

OutPutMatrixToStrGrd (mKp, StringGrid\_Kpp0);

mDummy1.Free;

mDummy2.Free;

mA.Free;

mB.Free;

mKp.Free;

mM.Free;

mN.Free;

mN1.Free;

end;

end;

procedure ComputeFromPage1;

var

Order: TOrder;

mA, mB, mABKp, mInCond, mKp: TMatrix;

mSolutionValues: TMatrix;

LowerLimit, UpperLimit, NumReturn, NumIntervals: Word;

begin

with Form\_Main do

begin

Order := SpinEdit\_Dim.Value;

mA := TMatrix.Create(Order, Order);

mB := TMatrix.Create(1, Order);

mKp := TMatrix.Create(Order, 1);

mInCond := TMatrix.Create(Order, 1);

LoadMatrixFromStrGrd(mA, StringGrid\_Ap1);

LoadMatrixFromStrGrd(mB, StringGrid\_Bp1);

LoadMatrixFromStrGrd(mKp, StringGrid\_Kpp1);

LoadMatrixFromStrGrd(mInCond, StringGrid\_InCond);

mABKp := TMatrix.Create(Order, Order);

mABKp.Assign(mB);

mABKp.MultFromRight(mKp);

mABKp.AddMatrix(mA);

OutPutMatrixToStrGrd(mABKp, StringGrid\_ABKpp1);

mB.MultConst(StrToFloat(Edit\_U.Text));

with Form\_Options do

begin

LowerLimit := SpinEdit0.Value;

UpperLimit := SpinEdit1.Value;

NumReturn := SpinEdit2.Value;

NumIntervals := SpinEdit3.Value;

end;

mSolutionValues := TMatrix.Create(1, 1);

try

DiffSystemSolve (mA, mB,

LowerLimit, UpperLimit,

mInCond,

NumReturn, NumIntervals,

mSolutionValues);

OutPutMatrixSolveToStrGrd(mSolutionValues, StringGrid\_Solve1);

mSolutionValues.ReSize(1, 1);

DiffSystemSolve (mABKp, mB,

LowerLimit, UpperLimit,

mInCond,

NumReturn, NumIntervals,

mSolutionValues);

OutPutMatrixSolveToStrGrd(mSolutionValues, StringGrid\_Solve2);

except

on EO: EOverflow do

begin

EO.Message := 'Не буду считать !!!'#10 +

'С уменьшите разброс коэффициентов в матрицах'#10 +

'либо измените опции (уменьшите их pls.)';

Raise;

end;

end;

mA.Free;

mB.Free;

mABKp.Free;

mInCond.Free;

mKp.Free;

mSolutionValues.Free;

end;

end;

procedure ShowChart(NumberOfChart: Byte);

var

Order, Serie: TOrder;

NumReturn, Point: Word;

mSolutionValues: TMatrix;

procedure SetAdm;

const

Divisor = 3.4E+38;

var

i, j: LongInt;

Greatest, Least: Float;

begin

Greatest := mSolutionValues[1, 1];

Least := Greatest;

for j := 1 to Order do

for i := 1 to NumReturn do

begin

if mSolutionValues[j, i] > Greatest then Greatest := mSolutionValues[j, i];

if mSolutionValues[j, i] < Least then Least := mSolutionValues[j, i];

end;

Form\_Main.ChartFX.Adm[CSA\_MAX] := Greatest;

Form\_Main.ChartFX.Adm[CSA\_MIN] := Least;

Form\_Main.ChartFX.Title[CHART\_TOPTIT] := 'Y = Y '' \* ';

end;

begin

with Form\_Main do

begin

Order := SpinEdit\_Dim.Value;

NumReturn := Form\_Options.SpinEdit2.Value;

mSolutionValues := TMatrix.Create(1, 1);

ComputeFromPage1;

case NumberOfChart of

1 :begin

LoadMatrixSolveFromStrGrd(mSolutionValues, StringGrid\_Solve1);

SetAdm;

ChartFX.OpenDataEx(Cod\_Values, Order, Pred(NumReturn));

for Serie := 1 to Order do

begin

ChartFX.SerLeg[Pred(Serie)] := 'X ' + IntToStr(Serie);

ChartFX.ThisSerie := Pred(Serie);

for Point := 0 to Pred(NumReturn) do

ChartFX.Value[Point] := mSolutionValues[Serie, Succ(Point)];

end;

ChartFX.CloseData(Cod\_Values);

{

ChartFX.OpenDataEx(Cod\_XValues, Order, Pred(NumReturn));

for Serie := 1 to Order do

begin

ChartFX.ThisSerie := Pred(Serie);

for Point := 0 to Pred(NumReturn) do

ChartFX.XValue[Point] := mSolutionValues[1, Succ(Point)];

end;

ChartFX.CloseData(Cod\_XValues);

}

end;

2 :begin

LoadMatrixSolveFromStrGrd(mSolutionValues, StringGrid\_Solve2);

SetAdm;

ChartFX.OpenDataEx(Cod\_Values, Order, Pred(NumReturn));

for Serie := 1 to Order do

begin

ChartFX.SerLeg[Pred(Serie)] := 'X ' + IntToStr(Serie);

ChartFX.ThisSerie := Pred(Serie);

for Point := 0 to Pred(NumReturn) do

ChartFX.Value[Point] := mSolutionValues[Serie, Succ(Point)];

end;

ChartFX.CloseData(Cod\_Values);

end;

end;

mSolutionValues.Free;

end;

end;

procedure ComputeFromPage2;

begin

ComputeFromPage1;

end;

procedure ComputeFromPage3;

begin

case Form\_Main.RadioGroupChart.ItemIndex of

0 :ShowChart(1);

1 :ShowChart(2);

end;

end;

end.

**unit Matrix;**

interface

uses SysUtils;

type

Float = Extended;

EMatrixOperatingError = class (Exception);

const

NearlyZero = 1E-15;

type

TMatrix = class (TObject)

private

DataPtr: Pointer;

FCols, FRows: Word;

function GetCell (ACol, ARow: Word): Float;

procedure SetCell (ACol, ARow: Word; AValue: Float);

function GetItem (NumItem: LongInt): Float;

procedure SetItem (NumItem: LongInt; AValue: Float);

procedure SwitchRows (FirstRow, SecondRow: Word);

public

constructor Create (NCols, NRows: Word);

destructor Destroy; override;

procedure Assign (AMatrix: TMatrix);

procedure ReSize (NewCols, NewRows: Word);

procedure SetNull;

procedure SetSingle;

procedure SetNegative;

procedure AddConst (AConst: Float);

procedure AddMatrix (AMatrix: TMatrix);

procedure MultConst (MConst: Float);

procedure MultFromRight (MMatrix: TMatrix);

procedure MultFromLeft (MMatrix: TMatrix);

procedure NthPower (Power: Word);

procedure Transpose;

function Inverse: Boolean;

function Determinant: Float;

function Rang: Float;

property ColCount: Word read FCols;

property RowCount: Word read FRows;

property Cells [ACol, ARow: Word]: Float read GetCell write SetCell; default;

property Items [NumItem: LongInt]: Float read GetItem write SetItem;

end;

implementation

uses Windows;

function IncPtr (p: Pointer; i: LongInt): Pointer;

asm

push EBX

mov EBX,EAX

add EBX,EDX

mov EAX,EBX

pop EBX

end;

function TMatrix.GetCell (ACol, ARow: Word): Float;

var

CellPtr: ^Float;

begin

CellPtr := IncPtr(DataPtr, (FRows \* Pred(ACol) + Pred(ARow)) \* SizeOf(Float));

Result := CellPtr^;

end;

procedure TMatrix.SetCell (ACol, ARow: Word; AValue: Float);

var

CellPtr: ^Float;

begin

CellPtr := IncPtr(DataPtr, (FRows \* Pred(ACol) + Pred(ARow)) \* SizeOf(Float));

CellPtr^ := AValue;

end;

function TMatrix.GetItem (NumItem: LongInt): Float;

var

CellPtr: ^Float;

begin

CellPtr := IncPtr(DataPtr, Pred(NumItem) \* SizeOf(Float));

Result := CellPtr^;

end;

procedure TMatrix.SetItem (NumItem: LongInt; AValue: Float);

var

CellPtr: ^Float;

begin

CellPtr := IncPtr(DataPtr, Pred(NumItem) \* SizeOf(Float));

CellPtr^ := AValue;

end;

procedure TMatrix.SwitchRows (FirstRow, SecondRow: Word);

var

i: Word;

Buffer: Float;

begin

for i := 1 to FCols do

begin

Buffer := GetCell(i, FirstRow);

SetCell(i, FirstRow, GetCell(i, SecondRow));

SetCell(i, SecondRow, Buffer);

end;

end;

constructor TMatrix.Create (NCols, NRows: Word);

begin

inherited Create;

FCols := NCols;

FRows := NRows;

DataPtr := AllocMem(FCols \* FRows \* SizeOf(Float));

end;

destructor TMatrix.Destroy;

begin

FreeMem(DataPtr);

inherited Destroy;

end;

procedure TMatrix.Assign (AMatrix: TMatrix);

var

NewMatrixSize: LongInt;

begin

NewMatrixSize := AMatrix.ColCount \* AMatrix.RowCount \* SizeOf(Float);

ReAllocMem(DataPtr, NewMatrixSize);

CopyMemory(DataPtr, AMatrix.DataPtr, NewMatrixSize);

FCols := AMatrix.ColCount;

FRows := AMatrix.RowCount

end;

procedure TMatrix.ReSize (NewCols, NewRows: Word);

var

NewMatrixSize: LongInt;

begin

NewMatrixSize := NewCols \* NewRows \* SizeOf(Float);

ReAllocMem(DataPtr, NewMatrixSize);

FCols := NewCols;

FRows := NewRows;

end;

procedure TMatrix.SetNull;

begin

ZeroMemory (DataPtr, FCols \* FRows \* SizeOf(Float));

end;

procedure TMatrix.SetSingle;

var

i: Word;

begin

if FCols <> FRows then

Raise EMatrixOperatingError.Create ('Единичная матрица должна быть '+

'квадратной')

else

begin

SetNull;

for i := 1 to FCols do SetCell (i, i, 1);

end;

end;

procedure TMatrix.SetNegative;

var

i: LongInt;

begin

for i := 1 to FCols \* FRows do SetItem(i, - GetItem(i));

end;

procedure TMatrix.AddConst (AConst: Float);

var

i: LongInt;

begin

for i := 1 to FCols \* FRows do SetItem (i, GetItem(i) + AConst);

end;

procedure TMatrix.AddMatrix (AMatrix: TMatrix);

var

i: LongInt;

begin

for i := 1 to FCols \* FRows do SetItem (i, GetItem(i) + AMatrix.Items [i]);

end;

procedure TMatrix.MultConst (MConst: Float);

var

i: LongInt;

begin

for i := 1 to FCols \* FRows do SetItem (i, GetItem(i) \* MConst);

end;

procedure TMatrix.MultFromRight (MMatrix: TMatrix);

var

j, i, k: Word;

DummyRes: Float;

DummyMatrix: TMatrix;

begin

DummyMatrix := TMatrix.Create (MMatrix.ColCount, FRows);

if FCols <> MMatrix.RowCount then

Raise EMatrixOperatingError.Create ('Перемножаемые матрицы должны быть '+

'соответствующей размерности')

else

for i := 1 to FRows do

for j := 1 to MMatrix.ColCount do

begin

DummyRes := 0;

for k := 1 to FCols do

DummyRes := DummyRes + Cells[k, i] \* MMatrix[j, k];

DummyMatrix[j, i] := DummyRes;

end;

Assign(DummyMatrix);

DummyMatrix.Free;

end;

procedure TMatrix.MultFromLeft (MMatrix: TMatrix);

var

j, i, k: Word;

DummyRes: Float;

DummyMatrix: TMatrix;

begin

DummyMatrix := TMatrix.Create (FCols, MMatrix.RowCount);

if MMatrix.ColCount <> FRows then

Raise EMatrixOperatingError.Create ('Перемножаемые матрицы должны быть '+

'соответствующей размерности')

else

for i := 1 to MMatrix.ColCount do

for j := 1 to FCols do

begin

DummyRes := 0;

for k := 1 to MMatrix.ColCount do

DummyRes := DummyRes + MMatrix[k, i] \* Cells[j, k];

DummyMatrix[j, i] := DummyRes;

end;

Assign(DummyMatrix);

DummyMatrix.Free;

end;

procedure TMatrix.NthPower (Power: Word);

var

i: Word;

DummyMatrix: TMatrix;

begin

DummyMatrix := TMatrix.Create (FCols, FRows);

DummyMatrix.Assign (Self);

if FCols <> FRows then

Raise EMatrixOperatingError.Create ('Возводимая в степень матрица должна '+

'быть квадратной')

else

case Power of

0 : SetSingle;

1 : begin end;

else

for i := 2 to Power do MultFromRight (DummyMatrix);

end;

DummyMatrix.Free;

end;

procedure TMatrix.Transpose;

var

i, j: Word;

Dummy: Float;

begin

if FCols <> FRows then

Raise EMatrixOperatingError.Create ('Транспонируемая матрица должна быть '+

'квадратной')

else

for i := 1 to FCols do

for j := 1 to FRows do

if j > i then

begin

Dummy := GetCell(j, i);

SetCell(j, i, GetCell(i, j));

SetCell(i, j, Dummy);

end

end;

function TMatrix.Inverse: Boolean;

var

DummyMatrix: TMatrix;

Divisor, Multiplier: Float;

Row, RefRow, NewRow, Term: Word;

Singular: Boolean;

begin

Singular := False;

DummyMatrix := TMatrix.Create (FCols, FRows);

if (FCols <> FRows) or (FCols = 0) then

Raise EMatrixOperatingError.Create ('Инвертируемая матрица должна быть '+

'квадратной и ненулевого размера');

if FCols = 1 then

if ABS(GetItem(1)) < NearlyZero then Singular := True

else DummyMatrix.Items[1] := 1 / GetItem(1);

if FCols > 1 then

begin

DummyMatrix.SetSingle;

RefRow := 0;

repeat

Inc(RefRow);

if ABS(Cells[RefRow, RefRow]) < NearlyZero then

begin

Singular := TRUE;

NewRow := RefRow;

repeat

Inc(NewRow);

if ABS(Cells[RefRow, NewRow]) > NearlyZero then

begin

SwitchRows(NewRow, RefRow);

DummyMatrix.SwitchRows(NewRow, RefRow);

Singular := False;

end;

until (not Singular) or (NewRow >= FCols);

end;

if not Singular then

begin

Divisor := Cells[RefRow, RefRow];

for Term := 1 to FCols do

begin

SetCell(Term, RefRow, GetCell(Term, RefRow)/Divisor);

DummyMatrix[Term, RefRow] := DummyMatrix[Term, RefRow]/Divisor;

end;

for Row := 1 to FCols do

if (Row <> RefRow) and (ABS(Cells[RefRow, Row]) > NearlyZero) then

begin

Multiplier := - Cells[RefRow, Row] / Cells[RefRow, RefRow];

for Term := 1 to FCols do

begin

SetCell(Term, Row, GetCell(Term, Row) +

Multiplier \* GetCell(Term, RefRow));

DummyMatrix[Term, Row] := DummyMatrix[Term, Row] +

Multiplier \* DummyMatrix[Term, RefRow];

end

end;

end;

until Singular or (RefRow >= FCols);

end;

Assign(DummyMatrix);

DummyMatrix.Free;

if not Singular then Result := True

else Result := False;

end;

function TMatrix.Determinant: Float;

begin

Result := 0;

end;

function TMatrix.Rang: Float;

begin

Result := 0;

end;

end.

**unit Operates;**

interface

uses Matrix, Grids, SysUtils;

const

MaxArraySize = 30;

type

Float = Extended;

TOrder = 1..MaxArraySize;

ESingularMatrix = class (Exception);

type

TComplex = record

Re, Im : Float;

end;

TComplexVector = record

Data : array [1..MaxArraySize] of TComplex;

Dim : TOrder;

end;

function SymmetricalFunction (Roots: TComplexVector; K: byte): Float;

procedure DiffSystemSolve (matrixA,

matrixB: TMatrix;

LowerLimit,

UpperLimit: Float;

InitialValues: TMatrix;

NumReturn,

NumIntervals: Word;

SolutionValues: TMatrix);

implementation

function SymmetricalFunction (Roots: TComplexVector; K: byte): Float;

var

Z: TComplex;

function SummComplex (FirstNC, SecondNC: TComplex): TComplex;

begin

Result.Re := FirstNC.Re + SecondNC.Re;

Result.Im := FirstNC.Im + SecondNC.Im;

end;

function MultComplex (FirstNC, SecondNC: TComplex): TComplex;

begin

Result.Re := FirstNC.Re \* SecondNC.Re - FirstNC.Im \* SecondNC.Im;

Result.Im := FirstNC.Re \* SecondNC.Im + FirstNC.Im \* SecondNC.Re;

end;

function DivComplex (FirstNC, SecondNC: TComplex): TComplex;

var

Z: Float;

begin

Z := Sqr(SecondNC.Re) + Sqr(SecondNC.Im);

Result.Re := (FirstNC.Re \* SecondNC.Re + FirstNC.Im \* SecondNC.Im) / Z;

Result.Im := (FirstNC.Im \* SecondNC.Re - FirstNC.Re \* SecondNC.Im) / Z;

end;

function CombinationSumm (LowLimit, HighLimit, K: byte): TComplex;

var i: byte;

begin

Result.Re := 0;

Result.Im := 0;

if LowLimit = HighLimit then Result := Roots.Data[LowLimit]

else

for i := LowLimit to HighLimit - K + 1 do

if K = 1 then Result := SummComplex(Result, Roots.Data [i])

else Result := SummComplex(Result,

MultComplex(Roots.Data [i],

CombinationSumm(i + 1,

HighLimit, K-1)));

end;

begin

Z := CombinationSumm(1, Roots.Dim, K);

Result := Z.Re;

end;

procedure DiffSystemSolve (matrixA, matrixB: TMatrix;

LowerLimit, UpperLimit: Float;

InitialValues: TMatrix;

NumReturn, NumIntervals: Word;

SolutionValues: TMatrix);

type

Ptr = ^Data;

Data = record

Values: TMatrix;

Next: Ptr;

end;

var

ValuesStack: Ptr;

Spacing, HalfSpacing: Float;

Index, Term: Word;

F1, F2, F3, F4,

CurrentValues,

TempValues: TMatrix;

NumEquations, NumTimeCol: Word;

function TargetALL (matrixA, mayrixB: TMatrix; Values: TMatrix; KRow: Word): Float;

var

j: Word;

begin

try

Result := matrixB.Items[KRow];

for j := 1 to NumEquations do

Result := Result + matrixA[j, KRow] \* Values.Items[j];

except

on EO: EOverflow do EO.Message := 'Не буду считать !!!'#10 +

'С уменьшите разброс коэффициентов в матрице А'#10 +

'либо измените опции (уменьшите их pls.)';

end;

end;

procedure Push (var ValuesStack: Ptr;

CurrentValues: TMatrix);

var

NewNode : Ptr;

begin

New(NewNode);

NewNode^.Values := TMatrix.Create(NumTimeCol, 1);

NewNode^.Values.Assign(CurrentValues);

NewNode^.Next := ValuesStack;

ValuesStack := NewNode;

end; { procedure Push }

procedure Pop (var ValuesStack: Ptr;

CurrentValues: TMatrix);

var

OldNode : Ptr;

begin

OldNode := ValuesStack;

ValuesStack := OldNode^.Next;

CurrentValues.Assign(OldNode^.Values);

OldNode^.Values.Free;

Dispose(OldNode);

end; { procedure Pop }

procedure GetValues(NumReturn, NumIntervals: Word; var ValuesStack: Ptr;

SolutionValues: TMatrix);

var

Index, Term: Integer;

j: Word;

CurrValues: TMatrix;

begin

SolutionValues.ReSize(NumTimeCol, Succ(NumReturn));

CurrValues := TMatrix.Create(NumTimeCol, 1);

Term := NumIntervals;

for Index := NumReturn downto 0 do

begin

Pop(ValuesStack, CurrValues);

Dec(Term);

while (Term / NumIntervals >= Index / NumReturn) and (Term >= 0) do

begin

Pop(ValuesStack, CurrValues);

Dec(Term);

end;

for j := 1 to NumTimeCol do

SolutionValues[j, Succ(Index)] := CurrValues.Items[j];

end;

CurrValues.Free;

end; { procedure GetValues }

procedure Step(Spacing: Float; CurrentValues: TMatrix; F: TMatrix);

var

i : byte;

begin

for i := 1 to NumEquations do

F.Items[i] := Spacing \* TargetALL (matrixA, matrixB, CurrentValues, i);

end; { procedure Step }

begin

NumEquations := matrixA.RowCount;

NumTimeCol := Succ(NumEquations);

ValuesStack := nil;

Spacing := (UpperLimit - LowerLimit) / NumIntervals;

CurrentValues := TMatrix.Create(1, 1);

CurrentValues.Assign(InitialValues);

CurrentValues.ReSize(NumTimeCol, 1);

CurrentValues.Items[NumTimeCol] := LowerLimit;

TempValues := TMatrix.Create(NumTimeCol, 1);

F1 := TMatrix.Create(NumTimeCol, 1);

F2 := TMatrix.Create(NumTimeCol, 1);

F3 := TMatrix.Create(NumTimeCol, 1);

F4 := TMatrix.Create(NumTimeCol, 1);

Push(ValuesStack, CurrentValues);

HalfSpacing := Spacing / 2;

for Index := 1 to NumIntervals do

begin

{ First step - calculate F1 }

Step(Spacing, CurrentValues, F1);

TempValues.Items[NumTimeCol] := CurrentValues.Items[NumTimeCol] + HalfSpacing;

for Term := 1 to NumEquations do

TempValues.Items[Term] := CurrentValues.Items[Term] + 0.5 \* F1.Items[Term];

{ 2nd step - calculate F2 }

Step(Spacing, TempValues, F2);

for Term := 1 to NumEquations do

TempValues.Items[Term] := CurrentValues.Items[Term] + 0.5 \* F2.Items[Term];

{ Third step - calculate F3 }

Step(Spacing, TempValues, F3);

TempValues.Items[NumTimeCol] := CurrentValues.Items[NumTimeCol] + Spacing;

for Term := 1 to NumEquations do

TempValues.Items[Term] := CurrentValues.Items[Term] + F3.Items[Term];

{ Fourth step - calculate F4[1]; first equation }

Step(Spacing, TempValues, F4);

{ Combine F1, F2, F3, and F4 to get }

{ the solution at this mesh point }

CurrentValues.Items[NumTimeCol] := CurrentValues.Items[NumTimeCol] + Spacing;

for Term := 1 to NumEquations do

CurrentValues.Items[Term] := CurrentValues.Items[Term] +

(F1.Items[Term] + 2 \* F2.Items[Term] +

2 \* F3.Items[Term] + F4.Items[Term]) /6;

Push(ValuesStack, CurrentValues);

end;

GetValues(NumReturn, NumIntervals, ValuesStack, SolutionValues);

F1.Free;

F2.Free;

F3.Free;

F4.Free;

CurrentValues.Free;

TempValues.Free;

end;

end.

**unit HelpUnit;**

interface

uses

Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,

StdCtrls, ExtCtrls, Buttons;

type

TForm\_Help = class(TForm)

BitBtn1: TBitBtn;

Bevel1: TBevel;

Label1: TLabel;

Label2: TLabel;

Label3: TLabel;

Label4: TLabel;

Label5: TLabel;

Label6: TLabel;

Label7: TLabel;

private

{ Private declarations }

public

{ Public declarations }

end;

var

Form\_Help: TForm\_Help;

implementation

{$R \*.DFM}

end.

**unit OptsUnit;**

interface

uses

Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,

StdCtrls, Spin, Buttons, ExtCtrls;

type

TForm\_Options = class(TForm)

CheckBox\_Link: TCheckBox;

SpinEdit1: TSpinEdit;

SpinEdit2: TSpinEdit;

SpinEdit3: TSpinEdit;

Label\_UpLimit: TLabel;

Label\_PointsNumber: TLabel;

Label\_Intervals: TLabel;

Label\_1: TLabel;

BitBtn\_Ok: TBitBtn;

BitBtn\_Cancel: TBitBtn;

SpinEdit0: TSpinEdit;

Label1: TLabel;

Bevel1: TBevel;

procedure SpinEdit0Change(Sender: TObject);

procedure SpinEdit2Change(Sender: TObject);

procedure CheckBox\_LinkClick(Sender: TObject);

private

{ Private declarations }

public

{ Public declarations }

end;

var

Form\_Options: TForm\_Options;

implementation

uses MainUnit, SubUnit;

{$R \*.DFM}

procedure TForm\_Options.SpinEdit0Change(Sender: TObject);

begin

SpinEdit1.MinValue := Succ(SpinEdit0.Value);

if SpinEdit1.Value < SpinEdit1.MinValue then SpinEdit1.Value:= SpinEdit1.MinValue;

end;

procedure TForm\_Options.SpinEdit2Change(Sender: TObject);

begin

SpinEdit3.MinValue := SpinEdit2.Value;

if SpinEdit3.Value < SpinEdit3.MinValue then SpinEdit3.Value:= SpinEdit3.MinValue;

end;

procedure TForm\_Options.CheckBox\_LinkClick(Sender: TObject);

begin

if CheckBox\_Link.State = cbChecked then Form\_Main.BindGrids

else Form\_Main.UnBindGrids

end;

end.