Contents

Chapter 1

Introduction

This note presents the documentation of version 2.0 of the NMODEL statistical package. The work is supported by The research Council of Norway through the research programme "Toolkits in Industrial mathematics", where the Norwegian Computing Center (NR) is one of three participants, the other two being SINTEF and the University of Oslo.

NMODEL contains methods for regression and state space modelling. The routines are implemented in C++. In the manual, private members, protected members, and member functions considered as implementation details, are in general not included in the documentations.

The regression methods implemented are especially suited for problems with few and noisy data. There are implemented methods for linear regression with univariate response, including ordinary least squares and several biased regression methods. There are also methods for multivariate response; reduced rank regression (linear) and projection pursuit regression (non linear). The class Reg, section 2.1.1 gives information about the structure and the contents of the regression module.

The state space model module consists of linear Kalman filtering and a non linear extension. The class KalmanFilter, section 3.1.1, is the main class for linear Kalman filtering. Similar the class KalmanFilterMixed, section 3.2.1, contains the non linear extension.

Differences between versions 1.0 and 2.0.

In version 2.0, the version 2.0 of the classes in the matrix library of NUTILITY are included. The regression module are restructured and new models and methods are implemented. The Kalman filter are extended to allow missing data and estimation of the parameters. A non linear version of the Kalman filter is implemented.

In addition, minor changes are done to several classes, and errors are corrected.

Chapter 2

Regression

2.1 Some superior abstract base classes

2.1.1 Reg

NAME

Reg - an abstract base class for a hierarchy of regression methods

INCLUDE

```
include "Reg.h"
```

SYNTAX

```
class Reg
{
 private:
  void notimpl_on_this_level() const;

 protected:
  BooLean fitted;  // dpTRUE if the model parameters are fitted or set

  int nosegcv;     // no. of segments in cross validation
  int itmax;       // maximum no. of iterations
  double itstop;   // value for stopping iterations
 public:
  Reg() :UndefMatrix(1,1,-9.9), UndefVector(1,-9.9)
    {fitted = dpFALSE; nosegcv=0; itmax=100; itstop=0.005;}

  virtual ~Reg() { }


  virtual void est(const RegData& rd) {estFixedMetaPar(rd);}
  virtual void estFixedMetaPar(const RegData& rd) = 0;


  virtual nMatrix pred(const nMatrix& Xtest) const = 0;


  virtual nVector getMetaPar() const          {nVector vec; return vec;}
  virtual void setMetaPar(const nVector& nv)  {notimpl_on_this_level();}


  virtual nVector CrossVal(const RegData& rd);
  virtual nVector CrossValFixedMetaPar(const RegData& rd);


  void setNosegcv(int i)         {nosegcv=i;}
  int getNosegcv()         const {return nosegcv;}
  void setItmax(int i)           {itmax=i;}
  int getItmax()           const {return itmax;}
  void setItstop(double d)       {itstop=d;}
  double getItstop()          const {return itstop;}

  BooLean modelFitted() const {return fitted;}
  virtual char* typeId() const {return "Reg";}
};

inline
void
```

```
Reg::notimpl_on_this_level() const
{
  fatalerrorFP("Reg::...",\
               "This function is not implemented on this level of Reg or inherited classes!");
}
```

KEYWORDS

regression, linear regression, non-linear regression, cross validation, prediction

DESCRIPTION

This is an abstract base class for a hierarchy of regression methods. The aim of a regression analysis is to explain the variation of a set of q response variables (the y-s) by a set of p explanatory variables (the x-es). Typical operations are to estimate the model parameters from n observations of the variables (the nxp data matrix X and the nxq data matrix Y), and then predict new y-s by a matrix Xnew. Two important concepts for the methods used here are *meta parameters* and *cross validation*. Most of the methods use one or two meta parameters (or tuning parameters), which may be the number of components in principal components regression or the ridge parameter in ridge regression. Cross validation is used to estimate the meta parameters, and to estimate the uncertainty. Most of the available methods are treated in Aldrin (1995).

The methods are either linear methods or variants of Projection Pursuit Regression (PPR) (non-linear methods). The hierarchy is shown below:

The PPR methods implemented are moderate PPR and ordinary PPR. The hierarchy structure for the PPR methods are as follows, where the classes of the two upper levels are abstract:

```
          Reg

         PPReg

ModPPReg        OrdPPReg
```

Some of the linear regression methods work by defining a set of new explanatory variables. The hierarchy for these methods are as follows, where the classes of the three upper levels are abstract:

```
                              Reg

                             LinReg

                            NewvarReg

         RedRankReg         VSSReg PLSReg       PCReg

RedRankPCReg RedRankARReg                JSPCReg JSPCReg2
```

The hierarchy of the other linear methods are as follows, where the classes of the two upper levels are abstract:

```
                              Reg

                             LinReg

       RidgeReg          PAOlsReg PAOlsReg2 JSReg OlsReg

JSRidgeReg JSRidgeReg2
```

REFERENCES

Aldrin, M. (1995), Regression with few and noisy data. Ph.D. thesis, University of Oslo, Department of mathematics.

MEMBER FUNCTIONS

The most imortant functions are listed below. See also the `SYNTAX` section.

`est` - Estimates the model parameters and the meta parameters. The meta parameters are typically estimated by cross validation.

`estFixedMetaPar` - Estimates the model parameters for fixed meta parameters.

`pred` - Predict y values for given x-values.

`getMetaPar` - Returns the meta parameters.

`setMetaPar` - Allows the user to set the meta parameters.

`CrossVal` - Calculates cross validated variance (mean squared errors) including the estimation of the meta parameters. If the meta parameters are estimated by cross validation, this function then performs a double (nested) cross validation.

`CrossValFixedMetaPar` - Calculates cross validated variance (mean squared errors) for fixed meta parameters.

`setNosegcv` - Allows the user to set the number of segments used in cross validations. If the value of the `nosegcv` variable is 0 (the default), the number of segments will be equal number of observations (leave-one-out cross validation), else the no. of segments will be equal the value of `nosegcv`.

FILES

Reg.C

EXAMPLE

```
/*****************************************************************************/
/*                                                                           */
/* Regresjon ved hjelp av tre ulike regresjonsmetoder i NMODEL.              */
/*                                                                           */
/* Modellen for alle metodene er                                             */
/* y = b0 + b'x + e                                                          */
/* men de estimerer b paa ulike maater.                                      */
/*                                                                           */
/* Anta vi har n observasjoner, p x-variable og en y-variabel. La X vaere en */
/* nxp datamatrise for forklaringsvariablene, sentert (og skalert), og       */
/* Y en sentert n-dimensjonal vektor med responsvariable.                    */
/* Da er estimatene for b gitt ved:                                          */
/*                                                                           */
/*                                                                           */
/*  OLS (vanlig) regresjon:                                                  */
/*             -1                                                            */
/*  bhatt=(X'X)   X'Y                                                        */
/*                                                                           */
/*                                                                           */
/*                                                                           */
/*  Ridge regresjon:                                                         */
/*                -1                                                         */
/*  bhatt=(X'X+kI)   X'Y                                                     */
```

```
/*                                                                          */
/*                                                                          */
/*                                                                          */
/*  James-Stein-Ridge regresjon:                                            */
/*                  -1                                                      */
/*  bhatt = c (X'X+kI)   X'Y = c bhatt_ridge                                */
/*                                                                          */
/*                                                                          */
/*  For alle metodene er                                                    */
/*                                                                          */
/*  b0hatt = ymean - bhatt' xmean                                           */
/*                                                                          */
/****************************************************************************/


#include <iostream.h>
#include <fstream.h>
#include <math.h>


#include <errors.h>
#include <matrix.h>
#include <RegDataMore.h>


#include <OlsReg.h>
#include <RidgeReg.h>
#include <JSRidgeReg.h>

main()
{

  ofstream resfil("example.res",ios::out); // resultatfil

// reads data
  int n=29;
  int q=1;
  int qq=13;
  int p=9;
  nMatrix Data("/local/stat/viim/data/papirNS.dat",n,24);

  nMatrix Y=Data.submatrix(1,n,1,q);      // extracts Y from Data
  nMatrix X=Data.submatrix(1,n,qq+1,qq+p);  // extracts X from Data

  RegData rd(X,Y);                        // regression data set


  nMatrix Xtest=X.submatrix(1,2,1,p);


//  OLS-regresjon:
//-----------------

  OlsReg olsr;
  olsr.est(rd);                          // estimering

  resfil<<"\n\nB0:\n";
  olsr.getB0().print(resfil);            // skriver ut konstantledd
  resfil<<"\n\nB:\n";
  olsr.getB().print(resfil);             // skriver ut regresjonskoeffisienter

  nMatrix Ypredols=olsr.pred(Xtest);     // prediksjon i nye x-punkter
  resfil<<"\n\nYpred:\n";
  Ypredols.print(resfil);                // skriver ut

  nVector cvmseols=olsr.CrossVal(rd);    // beregner maal for usikkerhet
  resfil<<"\n\ncvmse:\n";
```

```
  cvmseols.print(resfil);                // skriver ut


//  Ridge-regresjon:
//-----------------

  RidgeReg rr;

  rr.est(rd);                           // estimering, inkludert estimering
                                        // av meta-parameteren k

  resfil<<"\n\nmetapar:\n";
  rr.getMetaPar().print(resfil);        // skriver ut k

  resfil<<"\n\nB0:\n";
  rr.getB0().print(resfil);             // skriver ut konstantledd
  resfil<<"\n\nB:\n";
  rr.getB().print(resfil);              // skriver ut regresjonskoeffisienter

  resfil<<"\n\nYpred:\n";
  nMatrix Ypredrr=rr.pred(Xtest);       // prediksjon i nye x-punkter
  Ypredrr.print(resfil);                // skriver ut

  nVector cvmserr=rr.CrossVal(rd);      // beregner maal for usikkerhet,
                                        // inkludert usikkerhet knytta til
                                        // estimering av k
  resfil<<"\n\ncvmse:\n";
  cvmserr.print(resfil);                // skriver ut


//  James-Stein-Ridge-regresjon:
//-----------------

  JSRidgeReg jsrr;

  jsrr.est(rd);                          // estimering, inkludert estimering
                                         // av meta-parameterne k og c.

  resfil<<"\n\nmetapar:\n";
  jsrr.getMetaPar().print(resfil);       // skriver ut k og c

  resfil<<"\n\nB0:\n";
  jsrr.getB0().print(resfil);            // skriver ut konstantledd
  resfil<<"\n\nB:\n";
  jsrr.getB().print(resfil);             // skriver ut regresjonskoeffisienter

  resfil<<"\n\nYpred:\n";
  nMatrix Ypredjsrr=jsrr.pred(Xtest);    // prediksjon i nye x-punkter
  Ypredjsrr.print(resfil);               // skriver ut

  nVector cvmsejsrr=jsrr.CrossVal(rd);   // beregner maal for usikkerhet,
                                         // inkludert usikkerhet knytta til
                                         // estimering av k og c
  resfil<<"\n\ncvmse:\n";
  cvmsejsrr.print(resfil);               // skriver ut


}
```

SEEALSO

ModPPReg, OrdPPReg, RedRankReg, VSSReg, PLSReg, PCReg, RedRankPCReg, JSPCReg, RedRankARReg, JSPCReg2, RidgeReg, PAOlsReg, PAOlsReg2, JSReg, OlsReg, JSRidgeReg, JSRidgeReg2

AUTHOR

Magne Aldrin, NR

2.1.2 LinReg

NAME

LinReg - abstract base class for linear regression

INCLUDE

```
include "LinReg.h"
```

SYNTAX

```
class LinReg : virtual public Reg
{
 protected:
  BooLean intercept;   // if dpTRUE, an intercept is included
  nVector B0;          // vector of intercepts
  nMatrix B;           // the pxq matrix of regression coefficients
 public:
  LinReg()
    : Reg(), B0(), B(), intercept(dpTRUE) {;}

  LinReg(BooLean intercept2)
    : Reg(), B0(), B(), intercept(intercept2) {;}

  nMatrix pred(const nMatrix& Xtest) const;


  BooLean getIntercept() const {return intercept;}
  nVector getB0() const {if(intercept) {return B0;} else{return UndefVector;}}
  nMatrix getB()         const {return B;}

  char* typeId() const {return "LinReg";}
};
```

KEYWORDS

linear regression

DESCRIPTION

This is an abstract base class for linear regression. The linear regression model is

```
y = B0 + B'x + e
```

where

```
y  : qx1 vector of response variables
x  : px1 vector of explanatory variables
B0 : qx1 vector of intercepts
B  : pxq vector of regression coefficients
```

FILES

LinReg.C

SEEALSO

Reg, OlsReg

AUTHOR

Magne Aldrin, NR

2.1.3 NewvarReg

NAME

NewvarReg - abstract base class for some linear regression methods

INCLUDE

```
include "NewvarReg.h"
```

SYNTAX

```
class NewvarReg : virtual public LinReg
{
 protected:
  int k;          // no. of new variables or components
 public:
  NewvarReg() : LinReg(dpTRUE) {k=-1;}
  NewvarReg(BooLean intercept) : LinReg(intercept) {k=-1;}


  void est (const RegData& rd);

  nVector getMetaPar() const {return nVector(1,double(k));}
  void setMetaPar(const nVector& nv)  {k=int(nv(1));}
  int getK() const {return k;}
  void setK(int k2) {k=k2;}

  char* typeId() const {return "NewvarReg";}
};
```

KEYWORDS

linear regression

DESCRIPTION

This is an abstract base class for a set of linear regression methods that perform an ordinary least squares regresion of a set of new explanatory variables. The new variables may be a subset of the original variables or linear combinations of the original variables.

FILES

NewvarReg.C

SEEALSO

Reg, LinReg, VSSReg, PAOlsReg

AUTHOR

Magne Aldrin, NR

2.2 Linear regression methods - primarily with univariate response

2.2.1 OlsReg

NAME

OlsReg - ordinary least squares regression

INCLUDE

```
include "OlsReg.h"
```

SYNTAX

```
class OlsReg : virtual public LinReg
{
 protected:
  nVector sigma;   // residual standard deviation

 public:
  OlsReg() : LinReg(dpTRUE) {;}
  OlsReg(BooLean intercept) : LinReg(intercept) {;}


  void estFixedMetaPar (const RegData& rd);

  nVector getSigma()     const {return sigma;}

  char* typeId() const {return "OlsReg";}
};
```

KEYWORDS

linear regression, ordinary least squares regression, biased regression, shrinked regression

DESCRIPTION

The class is inherited from `LinReg` and implements ordinary least squares regression (see for instance Weisberg 1985). The method is unbiased, as opposed to most other methods in this hiearchy of regression methods. The estimate of the matrix of regression coefficients is

```
       -1
B=(X'X)   X'Y
```

where

```
X      : nxp data matrix of explanatory variables
Y      : nx1 data matrix of response variables
```

REFERENCES

Weisberg, S. (1985), Applied Linear Regression, New York: John Wiley & Sons.

FILES

OlsReg.C

EXAMPLE

See class `Reg`

SEEALSO

Reg, LinReg, PAOlsReg

AUTHOR

Magne Aldrin, NR

2.2.2 VSSReg

NAME

VSSReg - variable subset selection regression

INCLUDE

```
include "VSSReg.h"
```

SYNTAX

```
class VSSReg : public NewvarReg
{
 public:
  VSSReg() : LinReg(dpTRUE), NewvarReg() {;}
  VSSReg(BooLean intercept) : LinReg(intercept), NewvarReg(intercept) {;}


  void estFixedMetaPar (const RegData& rd);

  char* typeId() const {return "VSSReg";}
};
```

KEYWORDS

linear regression, variable subset selection regression, biased regression, shrinked regression

DESCRIPTION

This class is inherited from `NewvarReg` and implements variable subset selection regression as it is described in Breiman and Spector (1992). The method performs ordinary least squares regression (see class `OlsReg`) on a subset of the explanatory variables. For a given number of variables (the meta parameter `k`), the variables in the subset are found by forward selection. If the `est` function is used, the number of variables (`k`) is found by cross validation.

REFERENCES

Breiman, L. and Spector. P. (1992), Submodel Selection and Evaluation in Regression. The X-random Case, International Statistical Review, 60, 291-319.

FILES

VSSReg.C

EXAMPLE

```
#include <iostream.h>
#include <fstream.h>
#include <math.h>


#include <errors.h>
#include <matrix.h>
#include <RegData.h>


#include <VSSReg.h>


main()
{

  ofstream resfil("example.res",ios::out); // file with results

// reads data
  int n=29;
  int q=13;
  int p=9;
  nMatrix Data("/local/stat/viim/data/papirNS.dat",n,24);

  nMatrix Y=Data.submatrix(1,n,1,q);      // extracts Y from Data
  nMatrix X=Data.submatrix(1,n,q+1,q+p);  // extracts X from Data


  RegData rd(X,Y);             // regression data set


  VSSReg vssr;
  vssr.est(rd);                // estimation

  resfil<<"\n B0:\n";
  vssr.getB0().print(resfil);  // prints out the intercept terms
  resfil<<"\n B:\n";
  vssr.getB().print(resfil);   // prints out the regression coeffisients
                               // coefficients for variables not in
                               // subset gets the value 0

  resfil<<"\n k:";
  resfil<<"\n"<<vssr.getK();     // prints no. of variables in the subset
}
```

SEEALSO

Reg, LinReg, NewvarReg, PAOlsReg

AUTHOR

Magne Aldrin, NR

2.2.3 PLSReg

NAME

PLSReg - partial least squares (PLS) regression

INCLUDE

```
include "PLSReg.h"
```

SYNTAX

```
class PLSReg : public NewvarReg
{
 public:
  PLSReg() : LinReg(dpTRUE), NewvarReg() {;}
  PLSReg(BooLean intercept) : LinReg(intercept), NewvarReg(intercept) {;}


  void estFixedMetaPar (const RegData& rd);

  char* typeId() const {return "PLSReg";}
};
```

KEYWORDS

linear regression, PLS regression, biased regression, shrinked regression

DESCRIPTION

This class inherits `NewvarReg` and implements the PLS regression (Martens and Naes 1989, Frank and Friedman 1993). PLS finds linear combinations of the explanatory variables, and treats these as new explanatory variables. The components is found by the so called PLS1 algorithm if the response variable is univariate, and by the PLS2 algorithm if it is multivariate. The number of components is a meta parameter called k, which is found by cross validation if the `est` function is used.

REFERENCES

Frank, I. E. and Friedman, J. H. (1993), A statistical View of some Chemometric Regression Tools (with discussion), Technometrics, 35, 109-147.

Martens, H. and Naes, T. (1989), Multivariate Calibration, Chichester: John Wiley & Sons.

FILES

PLSReg.C

EXAMPLE

See class `Reg` and class `VSSReg`.

SEEALSO

Reg, LinReg, NewvarReg, PAOlsReg

AUTHOR

Magne Aldrin, NR

2.2.4 PCReg

NAME

PCReg - principal components regression

INCLUDE

```
include "PCReg.h"
```

SYNTAX

```
class PCReg : public NewvarReg
{
/*<PCReg:*/
 protected:
  nMatrix V;
  SymMatrix ZtZmin1;

  nVector est_testpred (const RegData& rdtrain, const RegData& rdtest,
                        const nMatrix& MetaPar);

/*<PCReg:*/
 public:
  PCReg() : LinReg(dpTRUE), NewvarReg() {;}
  PCReg(BooLean intercept) : LinReg(intercept), NewvarReg(intercept) {;}


  void estFixedMetaPar (const RegData& rd);

  char* typeId() const {return "PCReg";}
};
```

KEYWORDS

linear regression, principal components regression, biased regression, shrinked regression

DESCRIPTION

This class inherits `NewvarReg` and implements principal components regression (PCR for short) (Massy 1965, Frank and Friedman 1993). PCR finds linear combinations of the explanatory variables by principal component decomposition of the explanatory variables, and treats these as new explanatory variables. The number of components is a meta parameter called k, which is found by cross validation if the `est` function is used.

REFERENCES

Frank, I. E. and Friedman, J. H. (1993), A statistical View of some Chemometric Regression Tools (with discussion), Technometrics, 35, 109-147.

Massy, W. F. (1965), Principal Components Regression in Explanatory Statistical Research, Journal of American Statistical Assiciation, 60, 234-246.

FILES

PCReg.C

EXAMPLE

See class `Reg` and class `VSSReg`.

SEEALSO

Reg, LinReg, NewvarReg, PAOlsReg

AUTHOR

Magne Aldrin, NR

2.2.5 JSPCReg

NAME

JSPCReg - James-Stein principal components regression (JSPCR1)

INCLUDE

```
include "JSPCReg.h"
```

SYNTAX

```
class JSPCReg : public PCReg
{
 protected:
  double c;        // the c parameter (James-Stein parameter)
 public:
  JSPCReg() : LinReg(dpTRUE), PCReg() {c=1; kfitted=dpFALSE;}
  JSPCReg(BooLean intercept)
    : LinReg(intercept), PCReg(intercept) {c=1; kfitted=dpFALSE;}


  void estFixedMetaPar (const RegData& rd);

  nVector getMetaPar() const
    {nVector nv(2); nv(1)=double(k); nv(2)=c; return nv;}
  void setMetaPar(const nVector& nv)  {k=int(nv(1)); c=nv(2);}

  void setK(int k2) {k=k2; kfitted=dpTRUE;}
  double getC() const {return c;}
  void setC(double c2) {c=c2;}

  char* typeId() const {return "JSPCReg";}
};
```

KEYWORDS

linear regression, James-Stein principal components regression, biased regression, shrinked regression

DESCRIPTION

This class implements a version of James-Stein principal components regression (JSPCR1) (Aldrin 1995). The estimate of the regression matrix is

```
B = c * Bpc
```

where

```
c  : a scalar called the James-Stein parameter
Bc : an estimate of the regression matrix based on principal
     components regression (see class "PCReg")
```

If the `est` function is used, the estimation of c is done by cross validation (as opposed to JSPCR2).

REFERENCES

Aldrin, M. (1995), James-Stein versions of ridge regression and other biased regression methods, Technical Report STAT/05/95, Oslo: Norwegian Computing Center.

FILES

JSPCReg.C

EXAMPLE

See class `Reg`.

SEEALSO

Reg, LinReg, NewvarReg, PCReg, JSPCReg2, PAOlsReg

AUTHOR

Magne Aldrin, NR

2.2.6 JSPCReg2

NAME

JSPCReg2 - James-Stein principal components regression (JSPCR2)

INCLUDE

```
include "JSPCReg2.h"
```

SYNTAX

```
class JSPCReg2 : public PCReg
{
 protected:
  nVector c;                // vector of c (James-Stein) parameters

  nMatrix Bridge;           // preliminary estimate of B,
                            // used to calculate c

  nVector sigma2ols;        // vector of variance estimates,
                            // used to calculate c
 public:
  JSPCReg2() : LinReg(dpTRUE), PCReg()
    {kfitted=dpFALSE; Bridgefitted=dpFALSE; sigma2olsfitted=dpFALSE;}
  JSPCReg2(BooLean intercept) : LinReg(intercept), PCReg(intercept)
    {kfitted=dpFALSE; Bridgefitted=dpFALSE; sigma2olsfitted=dpFALSE;}


  void est(const RegData& rd);
  void estFixedMetaPar (const RegData& rd);

  void setK(int k2) {k=k2; kfitted=dpTRUE;}

  nMatrix getBridge() const {return Bridge;}
  void setBridge(const nMatrix& B2) {Bridge=B2; Bridgefitted=dpTRUE;}

  nVector getSigma2ols() const {return sigma2ols;}
  void setSigma2ols(const nVector& s2) {sigma2ols=s2; sigma2olsfitted=dpTRUE;}

   nVector getMetaPar() const
     {nVector nv(1); nv(1)=double(k); nv=nv.concat(c); return nv;}
  void setMetaPar(const nVector& nv)
    {k=int(nv(1)); c=nv.subvector(2,nv.getLen());}
  nVector getC() const {return c;}
  void setC(const nVector c2) {c=c2;}


  char* typeId() const {return "JSPCReg2";}
};
```

KEYWORDS

linear regression, James-Stein principal components regression, biased regression, shrinked regression

DESCRIPTION

This class implements a version of James-Stein principal components regression (JSPCR2) (Aldrin, 1995). The estimate of the regression matrix is

```
B = Diag(c) * Bpc
```

where

```
c        : a vector of so called James-Stein parameters
Diag(c)  : The diagonal matrix with c on the diagonal
Bpc      : an estimate of the regression matrix based on
           principal components regression (see class "PCReg")
```

The estimation of c is done by an analytical expression which involves preliminary estimates of B and of the variances. For this; the matrix `Bridge` and the vector `sigma2ols` are used, see the `SYNTAX` section.

REFERENCES

Aldrin, M. (1995), James-Stein versions of ridge regression and other biased regression methods, Technical Report STAT/05/95, Oslo: Norwegian Computing Center.

FILES

JSPCReg2.C

EXAMPLE

See class `Reg`.

SEEALSO

Reg, LinReg, NewvarReg, PCReg, JSPCReg, PAOlsReg

AUTHOR

Magne Aldrin, NR

2.2.7 JSReg

NAME

JSReg - James-Stein regression

INCLUDE

```
include "JSReg.h"
```

SYNTAX

```
class JSReg : virtual public LinReg
{
 protected:
  double c;   // the c parameter (James-Stein parameter)
 public:
  JSReg() : LinReg(dpTRUE) {c=1;}
  JSReg(BooLean intercept) : LinReg(intercept) {c=1;}


  void est (const RegData& rd);
  void estFixedMetaPar (const RegData& rd);


  nVector getMetaPar() const {return nVector(1,c);}
  void setMetaPar(const nVector& nv)  {c=nv(1);}
  double getC() const {return c;}
  void setC(double c2) {c=c2;}


  char* typeId() const {return "JSReg";}
};
```

KEYWORDS

linear regression, James-Stein regression, biased regression, shrinked regression

DESCRIPTION

The class is inherited from `LinReg` and implements James-Stein regression (James and Stein 1961, Aldrin 1995), which is a biased linear regression method. The estimate of the matrix of regression coefficients is

```
B = c Bols
```

where

```
c    : a scalar called the James-Stein parameter
Bols : an estimate of the regression matrix based on
       ordinary least squares (see class "OlsReg").
```

REFERENCES

James, W. and Stein, C. (1961), Estimation with Quadratic Loss, in Proceedings of the Fourth Berkeley Symposium (Vol 1), ed. J. Neuman, Berkeley: University of California.

Aldrin, M. (1995), Prediction adjusted least squares. To be found in Regression with few and noisy data. Ph.D. thesis, University of Oslo, Department of mathematics.

FILES

JSReg.C

EXAMPLE

See class `Reg`

SEEALSO

Reg, LinReg, PAOlsReg

AUTHOR

Magne Aldrin, NR

2.2.8 RidgeReg

NAME

RidgeReg - ridge regression

INCLUDE

```
include "RidgeReg.h"
```

SYNTAX

```
class RidgeReg : virtual public LinReg
{
 protected:
  double kridge;    // the ridge parameter
 public:
  RidgeReg() : LinReg(dpTRUE) {kridge=0;}
  RidgeReg(BooLean intercept) : LinReg(intercept) {kridge=0;}


  void est (const RegData& rd);
  void estFixedMetaPar (const RegData& rd);


  nVector getMetaPar() const {return nVector(1,kridge);}
  void setMetaPar(const nVector& nv)  {kridge=nv(1);}
  double getKridge() const {return kridge;}
  void setKridge(double kr) {kridge=kr;}


  char* typeId() const {return "RidgeReg";}
};
```

KEYWORDS

linear regression, ridge regression, biased regression, shrinked regression

DESCRIPTION

The class is inherited from `LinReg` and implements ridge regression, which is a biased linear regression method. The estimate of the regression matrix is

$$B=(X'X+ \text{kridge}*I)^{-1} X'Y$$

where

```
b      : px1 vector of regression coefficients
kridge : a (scalar) meta parameter called the ridge parameter
I      : the pxp identity matrix
X      : nxp data matrix of explanatory variables
Y      : nx1 data matrix of response variables
```

The original reference is Hoerl and Kennard (1970), whereas Frank and Friedman (1993) is a more recent reference.

REFERENCES

Hoerl, A. E. and Kennard, R. W. (1970), Ridge Regression: Biased Estimation for Non-orthogonal Problems, Technometrics, 8, 27-51.

Frank, I. E. and Friedman, J. H. (1993), A statistical View of some Chemometric Regression Tools (with discussion), Technometrics, 35, 109-147.

FILES

RidgeReg.C

EXAMPLE

See class `Reg`.

SEEALSO

Reg, LinReg, JSRidgeReg, PAOlsReg

AUTHOR

Magne Aldrin, NR

2.2.9 JSRidgeReg

NAME

JSRidgeReg - James-Stein ridge regression (JSRR1)

INCLUDE

```
include "JSRidgeReg.h"
```

SYNTAX

```
class JSRidgeReg : public RidgeReg
{
 protected:
  double c;        // the c parameter (James-Stein parameter)
 public:
  JSRidgeReg() : LinReg(dpTRUE), RidgeReg() {c=1; kridgefitted=dpFALSE;}
  JSRidgeReg(BooLean intercept)
    : LinReg(intercept), RidgeReg(intercept) {c=1; kridgefitted=dpFALSE;}


  void estFixedMetaPar (const RegData& rd);

  nVector getMetaPar() const {nVector nv(2); nv(1)=kridge; nv(2)=c; return nv;}
  void setMetaPar(const nVector& nv)  {kridge=nv(1); c=nv(2);}

  void setKridge(double kr) {kridge=kr; kridgefitted=dpTRUE;}
  double getC() const {return c;}
  void setC(double c2) {c=c2;}


  char* typeId() const {return "JSRidgeReg";}
};
```

KEYWORDS

linear regression, James-Stein ridge regression, biased regression, shrinked regression

DESCRIPTION

This class implements a version of James-Stein ridge regression (JSRR1) (Aldrin 1995). The estimate of the regression matrix is

```
B = c * Br
```

where

```
c  : a scalar called the James-Stein parameter
Br : an estimate of the regression matrix based on ridge
     regression (see class "RidgeReg")
```

If the `est` function is used, the estimation of c is done by cross validation (as opposed to JSRR2).

REFERENCES

Aldrin, M. (1995), James-Stein versions of ridge regression and other biased regression methods, Technical Report STAT/05/95, Oslo: Norwegian Computing Center.

FILES

JSRidgeReg.C

EXAMPLE

See class `Reg`.

SEEALSO

Reg, LinReg, RidgeReg, JSRRReg2, PAOlsReg

AUTHOR

Magne Aldrin, NR

2.2.10 JSRidgeReg2

NAME

JSRidgeReg2 - James-Stein ridge regression (JSRR2)

INCLUDE

```
include "JSRidgeReg2.h"
```

SYNTAX

```
class JSRidgeReg2 : public RidgeReg
{
 protected:
  nVector c;                // vector of c (James-Stein) parameters

  nMatrix Bridge;           // preliminary estimate of B,
                            // used to calculate c

  nVector sigma2ols;        // vector of variance estimates,
                            // used to calculate c
 public:
  JSRidgeReg2() : LinReg(dpTRUE), RidgeReg()
    {kridgefitted=dpFALSE; Bridgefitted=dpFALSE; sigma2olsfitted=dpFALSE;}
  JSRidgeReg2(BooLean intercept) : LinReg(intercept), RidgeReg(intercept)
    {kridgefitted=dpFALSE; Bridgefitted=dpFALSE; sigma2olsfitted=dpFALSE;}



  void est (const RegData& rd);
  void estFixedMetaPar (const RegData& rd);

  nVector getMetaPar() const
     {nVector nv(1); nv(1)=kridge; nv=nv.concat(c); return nv;}
  void setMetaPar(const nVector& nv)
    {kridge=nv(1); c=nv.subvector(2,nv.getLen());}

  void setKridge(double kr) {kridge=kr; kridgefitted=dpTRUE;}
  nVector getC() const {return c;}
  void setC(const nVector c2) {c=c2;}

  nMatrix getBridge() const {return Bridge;}
  void setBridge(const nMatrix& B2) {Bridge=B2; Bridgefitted=dpTRUE;}

  nVector getSigma2ols() const {return sigma2ols;}
  void setSigma2ols(const nVector& s2) {sigma2ols=s2; sigma2olsfitted=dpTRUE;}


  char* typeId() const {return "JSRidgeReg2";}
};
```

KEYWORDS

linear regression, James-Stein ridge regression, biased regression, shrinked regression

DESCRIPTION

This class implements a version of James-Stein ridge regression (JSRR2) (Aldrin, 1995). The estimate of the regression matrix is

```
B = Diag(c) * Br
```

where

```
c       : a vector of so called James-Stein parameters
Diag(c) : The diagonal matrix with c on the diagonal
Br      : an estimate of the regression matrix based on ridge
          regression (see class "RidgeReg")
```

The estimation of c is done by an analytical expression which estimates preliminary estimates of B and of the variances. For this; the matrix `Bridge` and the vector `sigma2ols` are used, see the `SYNTAX` section.

REFERENCES

Aldrin, M. (1995), James-Stein versions of ridge regression and other biased regression methods, Technical Report STAT/05/95, Oslo: Norwegian Computing Center.

FILES

JSRidgeReg2.C

EXAMPLE

See class `Reg`.

SEEALSO

Reg, LinReg, RidgeReg, JSRidgeReg, PAOlsReg

AUTHOR

Magne Aldrin, NR

2.2.11 PAOlsReg

NAME

PAOlsReg - prediction adjusted ordinary least squares regression (PAOLS1)

INCLUDE

```
include "PAOlsReg.h"
```

SYNTAX

```
class PAOlsReg : virtual public LinReg, public JSReg
{
 protected:
  double kridge;   // the ridge parameter (a meta parameter)
  double c;        // the c parameter     (a meta parameter)
  nMatrix M;       // the M matrix
 public:
  PAOlsReg() : LinReg(dpTRUE) {c=1; kridgefitted=dpFALSE;}
  PAOlsReg(BooLean intercept)
    : LinReg(intercept) {c=1; kridgefitted=dpFALSE;}


  void est (const RegData& rd);
  void estFixedMetaPar (const RegData& rd);


  nVector getMetaPar() const {nVector nv(2); nv(1)=kridge; nv(2)=c; return nv;}
  void setMetaPar(const nVector& nv)  {kridge=nv(1); c=nv(2);}
  double getKridge() const {return kridge;}
  void setKridge(double kr) {kridge=kr; kridgefitted=dpTRUE;}
  double getC() const {return c;}
  void setC(double c2) {c=c2;}
  nMatrix getM() const {return M;}
  void setM(const nMatrix& M2) {M=M2;}


  char* typeId() const {return "PAOlsReg";}
};
```

KEYWORDS

linear regression, prediction adjusted ordinary least squares regression, biased regression, shrinked regression

DESCRIPTION

The class is inherited from `LinReg` and implements a version of prediction adjusted least squares regression (PAOLS1), which is a biased linear regression method. The method is described in Aldrin (1995). The method may only be used for univariate y-es. The estimate of the regression vector is

```
B = c * M * Bols
```

where

```
B    : px1 vector
c    : scalar
Bols : px1 vector with ordinary least squares estimate of the
       regression coefficients
M    : pxp matrix
```

where

```
M = Br' Br C
```

```
Br : an estimate of the regression matrix based on ridge
     regression (see class RidgeReg)
C  : The correlation matrix of X (X is assumed scaled)
```

The scalars c and kridge (underlying Br) are regarded as meta parameters. They are typically estimated by cross validation by the `est` function.

REFERENCES

Aldrin, M. (1995), Prediction adjusted least squares. To be found in Regression with few and noisy data. Ph.D. thesis, University of Oslo, Department of mathematics.

FILES

PAOlsReg.C

EXAMPLE

See class `Reg`.

SEEALSO

Reg, LinReg, RidgeReg, PAOlsReg2

AUTHOR

Magne Aldrin, NR

2.2.12 PAOlsReg2

NAME

PAOlsReg2 - prediction adjusted ordinary least squares regression (PAOLS2)

INCLUDE

```
include "PAOlsReg2.h"
```

SYNTAX

```
class PAOlsReg2 : virtual public LinReg
{
 private:
  nMatrix A;                // adjusting matrix
  nMatrix Bridge;           // preliminary estimate of B,
                            // used to calculate c

  nVector sigma2ols;        // vector of variance estimates,
                            // used to calculate c
 public:
  PAOlsReg2() : LinReg(dpTRUE)
    {Bridgefitted=dpFALSE; sigma2olsfitted=dpFALSE;}
  PAOlsReg2(BooLean intercept) : LinReg(intercept)
    {Bridgefitted=dpFALSE; sigma2olsfitted=dpFALSE;}


  void estFixedMetaPar (const RegData& rd);

  nMatrix getA() const {return A;}

  nMatrix getBridge() const {return Bridge;}
  void setBridge(const nMatrix& B2) {Bridge=B2; Bridgefitted=dpTRUE;}

  nVector getSigma2ols() const {return sigma2ols;}
  void setSigma2ols(const nVector& s2) {sigma2ols=s2; sigma2olsfitted=dpTRUE;}

  char* typeId() const {return "PAOlsReg2";}
};
```

KEYWORDS

linear regression, prediction adjusted ordinary least squares regression, biased regression, shrinked regression

DESCRIPTION

The class is inherited from `LinReg` and implements a version of prediction adjusted least squares regression (PAOLS2), which is a biased linear regression method. The method is described in Aldrin (1995). The method may only be used for univariate y-es. The estimate of the regression vector is

```
B = A * Bols
```

where

```
B    : px1 vector
Bols : px1 vector with ordinary least squares estimate of the
       regression coefficients
A    : pxp matrix
```

where

```
A = (Br' Br C)/(s + Br'CBr)

C  : The correlation matrix of X (X is assumed scaled)
Br : an estimate of the regression matrix based on ridge
     regression (see class RidgeReg) ("Bridge" of the "SYNTAX" section)
s2 : an estimate of the variance ("sigma2ols" of the "SYNTAX" section)
```

REFERENCES

Aldrin, M (1995) Prediction adjusted least squares. To be found in Regression with few and noisy data. Ph.D. thesis, University of Oslo, Department of mathematics.

FILES

PAOlsReg2.C

EXAMPLE

See class `Reg`.

SEEALSO

Reg, LinReg, RidgeReg, PAOlsReg

AUTHOR

Magne Aldrin, NR

2.3 Reduced rank regression - linear with multivariate response

2.3.1 RedRankReg

NAME

RedRankReg - reduced rank regression

INCLUDE

```
include "RedRankReg.h"
```

SYNTAX

```
class RedRankReg : virtual public LinReg
{
 protected:
  int k;               // no. of components

  nMatrix Alpha;       // y-loadings
  nMatrix Beta;        // x-loadings

  BooLean one;         // if dpTRUE, the X-matrix is expanded by a
                       // column of 1-s

  nVector xmean;
  nVector ymean;

  RRCrossvalRes cvres;// results from cross validation

  WeightMat* wmat;    // optionally weights for observations
                      // and response variables

 public:
  RedRankReg()
    : LinReg(), cvres(), one(dpFALSE), Alpha(), Beta(), xmean(), ymean()
      {k=0; wmat = NULL;}

  RedRankReg(BooLean intercept)
    : LinReg(intercept), cvres(), one(dpFALSE), Alpha(), Beta(),
    xmean(), ymean()
      {k=0; wmat = NULL;}


  ~RedRankReg() {;}

  void est(const RegData& rd);
  void estFixedMetaPar(const RegData& rd);

  RRDiag diagnostics(const RegData& rd) const;

  BooLean getOne() const {return one;}
  void setOne(BooLean one2) {one=one2;}

  WeightMat* getWeightMat() const{return wmat;}
  void setWeightMat(WeightMat* wmat2) {wmat=wmat2;}

  nVector getMetaPar() const {return nVector(1,double(k));}
```

```
  void setMetaPar(const nVector& nv)  {k=int(nv(1));}
  int getK() const {return k;}
  void setK(int k2) {k=k2;}

  nMatrix getAlpha() const {if (k>0) {return Alpha;} else{return UndefMatrix;}}
  nMatrix getBeta()  const {if (k>0) {return Beta;} else{return UndefMatrix;}}

  RRCrossvalRes getCvres() const {return cvres;}

  char* typeId() const {return "RedRankReg";}
};
```

KEYWORDS

reduced rank regression, linear regression, multivariate regression

DESCRIPTION

The class is inherited from `NewvarReg`,which is inherited from `LinReg`, and implements reduced rank regression (Davies and Tso 1982, Aldrin 1995). The reduced rank regression model is

```
y = B0 +  B'x
```

where B can be written

```
B = Beta * Alpha'
```

Here

```
y     : q dimensional vector of responses
x     : p dimensional vector of explanatory variables
B0    : qx1 vector of intercepts
B     : pxq vector of regression coefficients
Alpha : qxk matrix of y-loadings
Beta  : pxk matrix of x-loadings
```

The matrix of response variables may contain missing observations. Then the actual `RegData` object should be of type class `RegDataM`, which is derived from class `RegData`.

By default, an intercept is included, which is computed separately from the decomposition of B. However, there is an option to include a column of 1-s in the X matrix. Then B will be a (p+1)xk matrix , and the decomposition of B will be different.

Another option is to weigth the response variables or the observations.

REFERENCES

Aldrin, M. (1995), Reduced rank regression for multivariate response with missing observations, Technical Report STAT/07/95, Oslo: Norwegian Computing Center.

Davies, P. T. and Tso, M. K.-S., Procedures for reduced-rank regression, Applied Statistics, 31, 244-255.

MEMBER FUNCTIONS

`diagnostics` - function computing some diagnostics for the estimated model.

FILES

RedRankReg.C

EXAMPLE

```
#include <iostream.h>
#include <fstream.h>
#include <math.h>


#include <errors.h>
#include <matrix.h>
#include <RegData.h>


#include <RedRankReg.h>


main()
{

  ofstream resfil("example.res",ios::out); // file with results

// reads data
  int n=29;
  int q=13;
  int p=9;
  nMatrix Data("/local/stat/viim/data/papirNS.dat",n,24);

  nMatrix Y=Data.submatrix(1,n,1,q);      // extracts Y from Data
  nMatrix X=Data.submatrix(1,n,q+1,q+p);  // extracts X from Data

  nMatrix M(n,q,1.0); // 1-s mean observed values and 0-s mean missing values

  M(3,1)=0;           // element (3,1) of Y is missing

  RegDataM rdm(X,Y,M);  // regression data with missing observations


  RedRankReg rrr;
  rrr.est(rdm);

  resfil<<"\n\nBeta:\n";
  rrr.getBeta().print(resfil);  // prints x-loadings
  resfil<<"\n\nAlpha:\n";
  rrr.getAlpha().print(resfil); // prints y-loadings
  resfil<<"\n\nB0:\n";
  rrr.getB0().print(resfil);    // prints out the intercept terms
  resfil<<"\n\nB:\n";
  rrr.getB().print(resfil);     // prints out the regression coeffisients

  resfil<<"\n"<< rrr.getK();    // prints no. of components
  rrr.getCvres().print(resfil); // prints cross validation results

  RRDiag rrd=rrr.diagnostics(rdm);
  rrd.print(resfil);         // prints diagnostics
```

```
  nMatrix Xnew=X.submatrix(1,2,1,9);
  nMatrix Ynewpred=rrr.pred(Xnew);  // prediction
}
```

SEEALSO

Reg, LinReg, NewvarReg, RedRankPCReg, RedRankARReg

AUTHOR

Magne Aldrin and Turid Follestad, NR

2.3.2 RedRankPCReg

NAME

RedRankPCReg - reduced rank regression combined with principal components

INCLUDE

```
include "RedRankPCReg.h"
```

SYNTAX

```
class RedRankPCReg : virtual public RedRankReg
{
 protected:
  int npc;      // no. principal components

  int pcmin;    // minimum  no. principal components
  int pcmax;    // maximum  no. principal components
  int pcnostep; // no. of steps from pcmin to pcmax

 public:
  RedRankPCReg()
    : LinReg(), RedRankReg() {npc=0; pcmin=1; pcmax=0; pcnostep=1;}

  RedRankPCReg(BooLean intercept)
    : LinReg(intercept), RedRankReg(intercept)
      {npc=0; pcmin=1; pcmax=0; pcnostep=1;}

  void est(const RegData& rd);
  void estFixedMetaPar(const RegData& rd);

  nVector getMetaPar() const {nVector nv(2); nv(1)=k; nv(2)=npc; return nv;}
  void setMetaPar(const nVector& nv)  {k=int(nv(1)); npc=int(nv(2));}
  int getK() const {return k;}
  void setK(int k2) {k=k2;}
  int getNpc() {return npc;}
  void setNpc(int npc2) {npc=npc2;}

  int getPcmin() const {return pcmin;}
  void setPcmin(int pcmin2) {pcmin=pcmin2;}
  int getPcmax() const {return pcmax;}
  void setPcmax(int pcmax2) {pcmax=pcmax2;}
  int getPcnostep() const {return pcnostep;}
  void setPcnostep(int pcnostep2) {pcnostep=pcnostep2;}

  char* typeId() const {return "RedRankPCReg";}
};
```

KEYWORDS

reduced rank regression, principal components regression, linear regression, multivariate regression

DESCRIPTION

This class is inherited from `RedRankReg`, and implements reduced rank regression (Davies

and Tso 1982, Aldrin 1995) combined with principal component analysis of the explanatory variables X. The first `npc` principal components of X are used as new variables, and a reduced rank regression is performed on these. The number of reduced rank components is denoted by `k`. Missing observations in the response variables are allowed.

REFERENCES

Aldrin, M. (1995), Reduced rank regression for multivariate response with missing observations, Technical Report STAT/07/95, Oslo: Norwegian Computing Center.

Davies, P. T. and Tso, M. K.-S., Procedures for reduced-rank regression, Applied Statistics, 31, 244-255.

FILES

RedRankPCReg.C

EXAMPLE

See class `RedRankReg`

SEEALSO

Reg, LinReg, NewvarReg, RedRankReg

AUTHOR

Magne Aldrin and Turid Follestad, NR

2.3.3 RedRankARReg

NAME

RedRankARReg - reduced rank regression with serially correlated errors

INCLUDE

```
include "RedRankARReg.h"
```

SYNTAX

```
class RedRankARReg : public virtual RedRankReg
{
 protected:
  RRARModel* armod;                   // AR-model

 public:
  RedRankARReg()
    : LinReg(), RedRankReg() {armod = NULL;}

  RedRankARReg(BooLean intercept)
    : LinReg(intercept), RedRankReg(intercept) {armod = NULL;}

  RedRankARReg(RRARModel& arm)
    : LinReg(), RedRankReg() {armod = &arm;}

  RedRankARReg(BooLean intercept, RRARModel& arm)
    : LinReg(intercept), RedRankReg(intercept) {armod = &arm;}

  ~RedRankARReg() { }


  nVector getARpar() const {return armod->getArPar();}

  RRDiag diagnostics(const RegData& rd) const;

  char* typeId() const {return "RedRankARReg";}
};
```

KEYWORDS

reduced rank regression, time series, AR-model, serial correlated errors

DESCRIPTION

This class is inherited from `RedRankReg`, and implements reduced rank regression (Davies and Tso 1982, Aldrin 1995) with an autoregressive model for the errors. Missing observations in the response variables are allowed.

The actual AR model for the residuals must be specified by the user. The user must define a class derived from `RRARModel`, and let an object of this class be input to a constructor. The parameters are estimated by an iterative procedure, iterating between the AR-model and the regression coefficient estimations.

REFERENCES

Aldrin, M. (1995), Reduced rank regression for multivariate response with missing observations, Technical Report STAT/07/95, Oslo: Norwegian Computing Center.

Davies, P. T. and Tso, M. K.-S., Procedures for reduced-rank regression, Applied Statistics, 31, 244-255.

CONSTRUCTORS AND INITIALIZATION

The constructors with the `RRARModel` objects define the AR-model for the errors.

FILES

CRCrossval.C

EXAMPLE

```
#include <iostream.h>
#include <fstream.h>
#include <math.h>


#include <errors.h>
#include <matrix.h>
#include <RegData.h>


#include <RedRankARReg.h>


#include "MyARModel.h"  // class derived from ARModel

main()
{

  ofstream resfil("example.res",ios::out); // file with results

// reads data
  int n=29;
  int q=13;
  int p=9;
  nMatrix Data("/local/stat/viim/data/papirNS.dat",n,24);

  nMatrix Y=Data.submatrix(1,n,1,q);      // extracts Y from Data
  nMatrix X=Data.submatrix(1,n,q+1,q+p);  // extracts X from Data


  RegData rd(X,Y);             // regression data set


  // AR-model object:

  int narp = ...;    // number of AR-parameters
  MyARModel armod(narp);

  RedRankARReg rrr(armod);
  rrr.est(rd);
```

```
  rrarrr.getARpar().print(resfil); // prints AR-parameters
  rrr.getBeta().print(resfil);  // prints x-loadings
  rrr.getAlpha().print(resfil); // prints y-loadings
  rrr.getB0().print(resfil);    // prints out the intercept terms
  rrr.getB().print(resfil);     // prints out the regression coeffisients

  resfil<<"\n"<< rrr.getK();    // prints no. of components
  rrr.getCvres().print(resfil); // prints cross validation results
}
```

SEEALSO

RedRankReg, RRARModel

AUTHOR

Magne Aldrin and Turid Follestad, NR

2.4 Projection pursuit regression - non-linear with multivariate response

2.4.1 PPReg

NAME

PPReg - abstract base class for projection pursuit regression

INCLUDE

```
include "PPReg.h"
```

SYNTAX

```
class PPReg : virtual public Reg
{
 protected:
  int k;               // no. of components

  nMatrix Alpha;       // y-loadings
  nMatrix Beta;        // x-loadings

  nMatrix Phiarg;      // arguments of the Phi-functions, Beta'X
  nMatrix Phi;         // the Phi-functions

  int kmin;             // minimum no. of components
  int kmax;             // maximum no. of components
  int knostep;          // no. of steps from kmin to kmax
  RRCrossvalRes cvres;  // various results from cross validation

  // smoothing parameters:
  double    span;
  double    bass;
 public:
  PPReg()
    : Reg(), Alpha(), Beta(), Phiarg(), Phi(), xmean(), ymean(), cvres()
      {k=0; span=0.0; bass=0.0; kmin=0; kmax=0; knostep=1;}


  ~PPReg() {;}

  void est(const RegData& rd);
  void estFixedMetaPar(const RegData& rd);

  nMatrix pred(const nMatrix& Xtest) const;

  nMatrix predTrain() const;               // prediction in the training sets


  nVector getMetaPar() const {return nVector(1,double(k));}
  void setMetaPar(const nVector& nv)  {k=int(nv(1));}
  int getK() const {return k;}             // returns no. of components
  void setK(int k2) {k=k2;}                // sets no. of components


  nMatrix getAlpha() const {if(k>0) {return Alpha;} else{return UndefMatrix;}}
  nMatrix getBeta()  const {if(k>0) {return Beta;} else{return UndefMatrix;}}
  nMatrix getPhiarg() const {if(k>0) {return Phiarg;}else{return UndefMatrix;}}
```

```
  nMatrix getPhi()    const {if(k>0) {return Phi;} else{return UndefMatrix;}}

  RRCrossvalRes getCvres() const {return cvres;}

  double getSpan() const {return span;}
  void setSpan(double span2) {span=span2;}
  double getBass() const {return bass;}
  void setBass(double bass2) {bass=bass2;}

  int getKmin() const {return kmin;}
  void setKmin(int kmin2) {kmin=kmin2;}
  int getKmax() const {return kmax;}
  void setKmax(int kmax2) {kmax=kmax2;}
  int getKnostep() const {return knostep;}
  void setKnostep(int knostep2) {knostep=knostep2;}

  char* typeId() const {return "PPReg";}
};
```

KEYWORDS

projection pursuit regression, non-linear regression

DESCRIPTION

Abstract base class for methods for projection pursuit regression.

The regression model is

```
y= ymean +  sum  Alpha  * Phi (Beta' (x-xmean) )
             j        j      j     j
```

where

```
y     : q dimensional vector of responses
x     : p dimensional vector of explanatory variables
ymean : q dimensional vector with means of the y-es
xmean : p dimensional vector with means of the x-es
Alpha : qxk matrix of y-loadings
Beta  : pxk matrix of x-loadings
Phi   : k non-parametric functions
```

With n observations (the data Y and X) , the Phi functions are represented as a nxk matrix `Phi` corresponding to the nxk matrix of arguments `Phiarg` = Beta'(X-xmean) (see the `SYNTAX` section). More information about projection pursuit regression are found in Friedman and Stuetzle (1981), Friedman 1985 and Aldrin (1995).

REFERENCES

Aldrin, M. (1995), Moderate projection pursuit regression for multivariate response data. To appear in Computational Statistics and Data Analysis.

Friedman, J. H. and Stuetzle, W. (1981), Projection Pursuit Regression, Journal of American Statistical Association, 76, 817-823.

Friedman, J. H. (1985),Classification and Multiple Regression through Projection Pursuit, Stanford University, Department of statistics, Report LCS, 12.

FILES

PPReg.C

EXAMPLE

```
#include <iostream.h>
#include <fstream.h>
#include <math.h>


#include <errors.h>
#include <matrix.h>
#include <RegDataMore.h>


#include <OrdPPReg.h>
#include <ModPPReg.h>

main()
{

  ofstream resfil("example.res",ios::out); // resultatfil

// reads data
  int n=29;
  int q=5;
  int qq=13;
  int p=9;
  nMatrix Data("/local/stat/viim/data/papirNS.dat",n,24);

  nMatrix Y=Data.submatrix(1,n,1,q);      // extracts Y from Data
  nMatrix X=Data.submatrix(1,n,qq+1,qq+p);  // extracts X from Data

  RegData rd(X,Y);                        // regression data set

  nMatrix Xnew=X;

// Ordinary PPR:
  OrdPPReg oppr;
  oppr.est(rd);                          // estimation
  nMatrix Ypredtrain=oppr.predTrain(); // prediction in training set
  nMatrix resid=Y-Ypredtrain;          // computing residuals
  nMatrix Yprednew=oppr.pred(Xnew);    // prediction for new x-es
  int k=oppr.getK();                   // optimal no. of components


// Moderate PPR:
  ModPPReg mppr;
  mppr.setK(3);                   // sets no. of components
  mppr.estFixedMetaPar(rd);       // estimation with fixed no. of components
  // prints the model to file:
  mppr.getAlpha().print(resfil);
  mppr.getBeta().print(resfil);
  mppr.getPhi().print(resfil);
  mppr.getPhiarg().print(resfil);

}
```

SEEALSO

Reg, OrdPPReg, ModPPReg

AUTHOR

Magne Aldrin, NR

2.4.2 OrdPPReg

NAME

OrdPPReg - ordinary projection pursuit regression

INCLUDE

```
include "OrdPPReg.h"
```

SYNTAX

```
class OrdPPReg : virtual public PPReg
{
 public:
  OrdPPReg() : PPReg() {print_to_file = dpFALSE;}


  void printToFile(BooLean b) { print_to_file = b;}

  char* typeId() const {return "OrdPPReg";}
};
```

KEYWORDS

ordinary projection pursuit regression, non-linear regression

DESCRIPTION

This class is derived from `PPReg` and implements ordinary projection pursuit regression (ordinary PPR, see Friedman and Stuetzle, 1981 and Friedman, 1985). If the data are few or noisy, moderate PPR may be a good alternative (see class `ModPPReg`).

REFERENCES

Friedman, J. H. and Stuetzle, W. (1981), Projection Pursuit Regression, Journal of American Statistical Association, 76, 817-823.

Friedman, J. H. (1985),Classification and Multiple Regression through Projection Pursuit, Stanford University, Department of statistics, Report LCS, 12.

MEMBER FUNCTIONS

printToFile - If the argument is dpTRUE, the results are written to the file `smart.res`.

For other functions, see the `SYNTAX` section and class `PPReg`.

FILES

OrdPPReg.C

EXAMPLE

See class `PPReg`.

SEEALSO

Reg, PPReg, ModPPReg

AUTHOR

Magne Aldrin, NR

2.4.3 ModPPReg

NAME

ModPPReg - moderate projection pursuit regression

INCLUDE

```
include "ModPPReg.h"
```

SYNTAX

```
class ModPPReg : virtual public PPReg
 public:
  ModPPReg() : PPReg()
    {diffminj=9999999; convcode=-9;}


  char* typeId() const {return "ModPPReg";}
};
```

KEYWORDS

moderate projection pursuit regression, non-linear regression

DESCRIPTION

This class is derived from `PPReg` and implements moderate projection pursuit regression (moderate PPR, see Aldrin, 1995). Compared to the ordinary PPR (see class `OrdPPReg`), moderate PPR may be more suitable if the data are few or noisy.

REFERENCES

Aldrin, M. (1995), Moderate projection pursuit regression for multivariate response data. To appear in Computational Statistics and Data Analysis.

FILES

ModPPReg.C

EXAMPLE

See class `PPReg`.

SEEALSO

Reg, PPReg, OrdPPReg

AUTHOR

Magne Aldrin, NR

2.5 Regression data

2.5.1 RegData

NAME

RegData - contains data matrices for regression

INCLUDE

```
include "RegData.h"
```

SYNTAX

```
class RegData : public DataSet
{
protected:
  int n;
  int p;
  int q;

  nMatrix X;
  nMatrix Y;

  BooLean intercept_added;   // dpTRUE if intercept is added to X-nMatrix
                             // by addIntercept()
public:
  RegData(int nn, int pp, int qq)
    :X(nn,pp),Y(nn,qq)
      { n = nn; p = pp; q = qq; intercept_added = dpFALSE;}
  RegData(const RegData& rd)
    :X(rd.X),Y(rd.Y)
      { n = rd.n; p = rd.p; q = rd.q; intercept_added = rd.intercept_added;}
  RegData(const nMatrix& X, const nMatrix& Y);

  // ** intercept:

  void addIntercept();         // appends a column of ones in X-nMatrix
  BooLean removeIntercept();   // removes appended column of ones in X-nMatrix
  BooLean interceptAdded() const { return intercept_added; }

  void setX(const nMatrix& m);  // with index check
  void setY(const nMatrix& m);  // with index check

  int getNobs() const {return n;}
  int getP() const {return X.getCdim();}
  int getQ() const {return Y.getCdim();}

  nMatrix getX()   const {return X;}
  nMatrix getY()   const {return Y;}

  virtual BooLean ok ();       // returns dpFALSE if n == 0
  virtual char* typeId() const { return "RegData";}

  // ** edition of the data:

  void cleanUp () { ;}         // inherited from class DataSet

  void extract(const nIntVector& ind, DataSet& ds) const;
  void extract(int from, int to, DataSet& ds) const;
```

```
  void remove (const nIntVector& ind);
  void remove (int from, int to);
  void insert (const DataSet&, int from);

  void scan (Is in);    // reads  X  and  Y  from input stream
  void print (Os out) const;  // writes  X  and  Y  to output stream
};
```

KEYWORDS

regression, data

DESCRIPTION

The class contains the matrices of predictor (`X`) and response (`Y`) variables of regression, and has several member functions for editing the data. The class is derived from the class `DataSet`, an abstract base class for sample data sets.

CONSTRUCTORS AND INITIALIZATION

The class has a copy constructor, a constructor with two nMatrix arguments holding the predictor (`X`) and response (`Y`) variables, and a constructor taking three integer arguments. These are the number of observations (`n`), the number of predictor variables (`p`) and the number of response varibles (`q`).

MEMBER FUNCTIONS

`ok` - returns dpTRUE if the number of observations `n` > 0.

In the following functions for editing the data, the actual `DataSet` reference arguments are expected to be of type class `RegData`.

`extract` - extracts observations indexed by `ind`, or from `from` to `to` (observations number `from` and `to` included), returning a new `DataSet`, holding the extracted values.

`remove` - removes observations indexed by `ind`, or from `from` to `to` (observations number `from` and `to` included).

`insert` - inserts data set after observation number `from-1`.

`scan` - reads coordinates and response values from input stream. The first `p` columns are expected to contain the predictor variables (`X`), and the following `q` columns the corresponding response variables (`Y`). If the total of number of response and predictor variables do not correspond to the values used to initialize the object, an error message is given. If the number of points on the input stream is greater than the initialized value, a warning will be issued, and the member matrices redimensioned to correspond to the number of lines on the stream.

`print` - prints the predictor and response variables in the same order as explained above for the `scan` function.

EXAMPLE

See class LinCanRegModel

SEEALSO

class RegDataM, class WeightMat

AUTHOR

Magne Aldrin, NR

2.5.2 RegDataM

NAME

RegDataM - contains data matrices for regression with missing observations.

INCLUDE

```
include "RegData.h"
```

SYNTAX

```
class RegDataM : public RegData
{
private:
  nMatrix M;

public:
  RegDataM(int n, int p, int q)
    : RegData(n,p,q), M(n,q) { }
  RegDataM(const RegDataM& rdm)
    : RegData(rdm), M(rdm.M) { }
  RegDataM(const nMatrix& X, const nMatrix& Y, const nMatrix& M);

  void setM(const nMatrix& m);  // with index check

  nMatrix getM()   const {return M;}

  char* typeId() const { return "RegDataM";}

  // ** functions for editing the data:

  void extract(const nIntVector& ind, DataSet& ds) const;
  void extract(int from, int to, DataSet& ds) const;
  void remove (const nIntVector& ind);
  void remove (int from, int to);
  void insert (const DataSet&, int from);
  void scan (Is in)  { RegData::scan(in);}  // missing values set by
                                            // setM(...)
  void print (Os out) const { RegData::print(out);}
};
```

KEYWORDS

regression, data, missing observations

DESCRIPTION

The class, derived from class `RegData`, is a representation of a set of regression data with missing observations. In addition to the member matrices `X` and `Y` inherited from class `RegData`, the class has a member nMatrix `M` of the same dimensions as `Y`, indicating nonmissing (1) and missing (0) observations of the response variables.

CONSTRUCTORS AND INITIALIZATION

The class has three constructors, including a copy constructor. One constructor takes three nMatrix arguments holding the predictor (`X`) and response (`Y`) variables, and the nMatrix `M` of ones and zeros indicating nonmissing and missing observations respectively. The last constructor takes three integer arguments. These are the number of observations (`n`), the number of predictor variables (`p`) and the number of response varibles (`q`).

MEMBER FUNCTIONS

In the following functions for editing the data, the actual `DataSet` reference arguments are expected to be of type class `RegData`.

`extract` - extracts observations indexed by `ind`, or from `from` to `to` (observations number `from` and `to` included), returning a new `DataSet`.

`remove` - removes observations indexed by `ind`, or from `from` to `to` (observations number `from` and `to` included).

`insert` - inserts data set after observation number `from-1`.

`scan` - calls `RegData::scan`. The values of the nMatrix M, can be initialized by the member function `setM`.

`print` - calls `RegData::print`.

EXAMPLE

See class LinCanRegModel

SEEALSO

class RegData, class WeightMat

AUTHOR

Magne Aldrin, NR

2.5.3 WeightMat

NAME

WeightMat - contains weights for regression data.

INCLUDE

```
include "RegData.h"
```

SYNTAX

```
class WeightMat
{
private:
  int n;
  int q;

  nVector v;
  SymMatrix W;

public:
  WeightMat(int n, int q);
  WeightMat(const WeightMat& wm);
  WeightMat(const nVector& v, const SymMatrix& W);

  // ** input:
  void setV(const nVector& v)  { this->v.setEqual(v) ;}
  void setW(const SymMatrix& m)  { this->W.setEqual(m) ;}

  // ** output:
  int getN() const {return n;}
  int getQ() const {return q;}

  nVector getV()   const {return v;}
  SymMatrix getW()   const {return W;}
};
```

KEYWORDS

regression, data, weights

DESCRIPTION

The class contains weights for case weighting and response variable weigting in regression.

CONSTRUCTORS AND INITIALIZATION

The class has three constructors, including a copy constructor. One constructor takes two integer arguments, the number of observations (`n`) and the number of response variables (`q`). The arguments fo the second constructor are: a nVector holding the `n` weights for case weighting, and a `q` by `q` symmetric matrix.holding the response variable weights.

EXAMPLE

See class LinCanRegModel

SEEALSO

class RegData, class RegDataM, class CanRegModel

AUTHOR

Magne Aldrin, NR

2.5.4 RegDataManip

NAME

RegDataManip - base class for manipulated regression data

INCLUDE

```
include "RegDataMore.h"
```

SYNTAX

```
class RegDataManip : public RegData
{
private:
  void notimpl() const;  // used for functions not implemented

public:
  RegDataManip(int n, int p, int q)
    : RegData(n,p,q) {;}
  RegDataManip(const RegDataManip& rdc)
    : RegData(rdc) {;}
  RegDataManip(const nMatrix& X, const nMatrix& Y)
    : RegData(X,Y) {;}
  RegDataManip(const RegData& rd)
    : RegData(rd) {;}



  virtual nVector getXmean() const {notimpl();}
  virtual nVector getYmean() const {notimpl();}
  virtual nVector getXstd()  const {notimpl();}
  virtual nVector getYstd()  const {notimpl();}

  // ** some functions defined by class RegData are not valid functions here:
  void addIntercept() {notimpl();}
  BooLean removeIntercept() {notimpl();}
  void remove (const nIntVector& ind) {notimpl();}
  void remove (int from, int to) {notimpl();}
  void insert (const DataSet&, int from) {notimpl();}
  void scan (istream& in)  {notimpl();}

  void print (ostream& out) const { RegData::print(out);}

  char* typeId() const { return "RegDataManip";}
};
```

KEYWORDS

regression data

DESCRIPTION

A base class for regression data where the X and/or the Y matrices are manipulated in some sense (e.g. centered or standardized). The class is derived from class RegData, but some functions defined there are redefined as unimplemented functions here.

FILES

RegDataMore.C

EXAMPLE

```
nMatrix X(nobs,ixdim);
nMatrix Y(nobs,iydim);

// put then values into X and Y

RegData rd1(X,Y);         // regression data
RegDataCent rd2(X,Y);     // regression data with centered X and Y
RegDataCXstand rd3(X,Y);  // regression data with centered X and Y,
                         // and standardized X
RegDataXstand rd4(X,Y);   // regression data with standardized X
RegDataYstand rd5(X,Y);   // regression data with standardized Y
```

SEEALSO

RegData, RegDataCent, RegDataCXstand, RegDataXstand, RegDataYstand

AUTHOR

Magne Aldrin, NR

2.5.5 RegDataCent

NAME

RegDataCent - regression data with centered X and Y

INCLUDE

```
include "RegDataMore.h"
```

SYNTAX

```
class RegDataCent : public RegDataManip
{
private:

protected:
  nVector xmean;
  nVector ymean;

  void center();

public:
  RegDataCent(int n, int p, int q)
    : RegDataManip(n,p,q), xmean(p), ymean(q) {;}
  RegDataCent(const RegDataCent& rdc)
    : RegDataManip(rdc), xmean(rdc.xmean), ymean(rdc.ymean) {;}
  RegDataCent(const nMatrix& X2, const nMatrix& Y2)
    : RegDataManip(X2,Y2), xmean() , ymean() {center();}
  RegDataCent(const RegData& rd)
    : RegDataManip(rd), xmean() , ymean() {center();}


  nVector getXmean()   const {return xmean;}
  nVector getYmean()   const {return ymean;}

  void print (ostream& out) const
    { RegDataManip::print(out); xmean.print(out); ymean.print(out);}

  char* typeId() const { return "RegDataCent";}
};
```

KEYWORDS

regression data, centering

DESCRIPTION

A class for regression data where X and Y are centered.

CONSTRUCTORS AND INITIALIZATION

The constructors centers X and Y when the object is initialized.

FILES

RegDataManip.C

EXAMPLE

```
nMatrix X(nobs,ixdim);
nMatrix Y(nobs,iydim);

// put then values into X and Y

RegData rd1(X,Y);         // regression data
RegDataCent rd2(X,Y);     // regression data with centered X and Y
RegDataCXstand rd3(X,Y);  // regression data with centered X and Y,
                         // and standardized X
RegDataXstand rd4(X,Y);   // regression data with standardized X
RegDataYstand rd5(X,Y);   // regression data with standardized Y
```

SEEALSO

RegData, RegDataManip, RegDataCXstand, RegDataXstand, RegDataYstand

AUTHOR

Magne Aldrin, NR

2.5.6 RegDataCXStand

NAME

RegDataCXStand - regression data with standardized X and centered Y

INCLUDE

```
include "RegDataMore.h"
```

SYNTAX

```
class RegDataCXStand : public RegDataCent
{
private:

protected:
  nVector xstd;

  void standX();

public:
  RegDataCXStand(int n, int p, int q)
    : RegDataCent(n,p,q), xstd(p) { }
  RegDataCXStand(const RegDataCXStand& rdcxs)
    : RegDataCent(rdcxs), xstd(rdcxs.xstd) { }
  RegDataCXStand(const nMatrix& X2, const nMatrix& Y2)
    : RegDataCent(X2,Y2), xstd() {standX();}
  RegDataCXStand(const RegData& rd)
    : RegDataCent(rd), xstd() {standX();}


  nVector getXstd()    const {return xstd;}

  void print (ostream& out) const
    { RegDataCent::print(out); xstd.print(out);}

  char* typeId() const { return "RegDataCXStand";}
};
```

KEYWORDS

regression data, centering, standardization

DESCRIPTION

A class for regression data where X and Y are centered, and X is standardized to standard deviation 1. The standard deviation is defined with n (number of observations) in the denominator, not n-1!

CONSTRUCTORS AND INITIALIZATION

The centering and standardization are done in the initialization.

FILES

RegDataMore.C

EXAMPLE

```
nMatrix X(nobs,ixdim);
nMatrix Y(nobs,iydim);

// put then values into X and Y

RegData rd1(X,Y);         // regression data
RegDataCent rd2(X,Y);     // regression data with centered X and Y
RegDataCXstand rd3(X,Y);  // regression data with centered X and Y,
                         // and standardized X
RegDataXstand rd4(X,Y);   // regression data with standardized X
RegDataYstand rd5(X,Y);   // regression data with standardized Y
```

SEEALSO

RegData, RegDataManip, RegDataCent, RegDataXStand, RegDataYStand

AUTHOR

Magne Aldrin, NR

2.5.7 RegDataXStand

NAME

RegDataXStand

INCLUDE

```
include "RegDataMore.h"
```

SYNTAX

```
class RegDataXStand : public RegDataManip
{
private:

protected:
  nVector xstd;

  void standX();

public:
  RegDataXStand(int n, int p, int q)
    : RegDataManip(n,p,q), xstd(p) { }
  RegDataXStand(const RegDataXStand& rdxs)
    : RegDataManip(rdxs), xstd(rdxs.xstd) { }
  RegDataXStand(const nMatrix& X2, const nMatrix& Y2)
    : RegDataManip(X2,Y2), xstd() {standX();}
  RegDataXStand(const RegData& rd)
    : RegDataManip(rd), xstd() {standX();}


  nVector getXstd()    const {return xstd;}

  char* typeId() const { return "RegDataXStand";}

  // ** functions for editing the data:

  void print (ostream& out) const
    { RegDataManip::print(out); xstd.print(out);}
};
```

KEYWORDS

regression data, standardization

DESCRIPTION

A class for regression data where X is standardized to standard deviation 1. The standard deviation is defined with n (number of observations) in the denominator, not n-1!

CONSTRUCTORS AND INITIALIZATION

The standardization is done in the initialization.

FILES

RegDataMore.C

EXAMPLE

```
nMatrix X(nobs,ixdim);
nMatrix Y(nobs,iydim);

// put then values into X and Y

RegData rd1(X,Y);         // regression data
RegDataCent rd2(X,Y);     // regression data with centered X and Y
RegDataCXstand rd3(X,Y);  // regression data with centered X and Y,
                         // and standardized X
RegDataXstand rd4(X,Y);   // regression data with standardized X
RegDataYstand rd5(X,Y);   // regression data with standardized Y
```

SEEALSO

RegData, RegDataManip, RegDataCent, RegDataCXStand, RegDataYStand

AUTHOR

Magne Aldrin, NR

2.5.8 RegDataYStand

NAME

RegDataYStand

INCLUDE

```
include "RegDataMore.h"
```

SYNTAX

```
class RegDataYStand : public RegDataManip
{
private:

protected:
  nVector ystd;

  void standY();

public:
  RegDataYStand(int n, int p, int q)
    : RegDataManip(n,p,q), ystd(p) { }
  RegDataYStand(const RegDataYStand& rdxs)
    : RegDataManip(rdxs), ystd(rdxs.ystd) { }
  RegDataYStand(const nMatrix& X2, const nMatrix& Y2)
    : RegDataManip(X2,Y2), ystd() {standY();}
  RegDataYStand(const RegData& rd)
    : RegDataManip(rd), ystd() {standY();}


  nVector getYstd()    const {return ystd;}

  void print (ostream& out) const
    { RegDataManip::print(out); ystd.print(out);}

  char* typeId() const { return "RegDataYStand";}
};
```

KEYWORDS

regression data, standardization

DESCRIPTION

A class for regression data where Y is standardized to standard deviation 1. The standard deviation is defined with n (number of observations) in the denominator, not n-1!

CONSTRUCTORS AND INITIALIZATION

The standardization is done in the initialization.

FILES

RegDataMore.C

EXAMPLE

```
nMatrix X(nobs,ixdim);
nMatrix Y(nobs,iydim);

// put then values into X and Y

RegData rd1(X,Y);         // regression data
RegDataCent rd2(X,Y);     // regression data with centered X and Y
RegDataCXstand rd3(X,Y);  // regression data with centered X and Y,
                         // and standardized X
RegDataXstand rd4(X,Y);   // regression data with standardized X
RegDataYstand rd5(X,Y);   // regression data with standardized Y
```

SEEALSO

RegData, RegDataManip, RegDataCent, RegDataCXStand, RegDataYStand

AUTHOR

Magne Aldrin, NR

Chapter 3

State space modelling

3.1 Linear state space modelling - The Kalman filter

3.1.1 KalmanFilter

NAME

KalmanFilter - main class for all methods/models related to Kalman filtering

INCLUDE

```
include "KalmanFilter.h"
```

SYNTAX

```
class KalmanFilter
{
protected:
  KalmanStart* ks;             // starting values for filter, "estimated"
  ArrayGenSimple(nMatrix)* y;  // nMatrix for data time series
  ArrayGenSimple(nMatrix)* M;  // nMatrix indicating missing (1) and
                               // non missing (0) observations


  KalmanFRes* kfr;             // the result from the kalman filter

public:
  KalmanParam* kp;             // contains the Kalman parameters

  // ** constructors:
  KalmanFilter() { kfr = NULL; kp = NULL; ks = NULL; y = NULL; M = NULL;}
  KalmanFilter(KalmanParam& param,
               const ArrayGenSimple(nMatrix)& maty);
  KalmanFilter(KalmanParam& param,
               const ArrayGenSimple(nMatrix)& maty,
               const ArrayGenSimple(nMatrix)& M2);

  // ** destructor:
  ~KalmanFilter() { if (kp != NULL) delete kp; if (ks != NULL) delete ks;
                    if (y != NULL) delete y; if (M != NULL) delete M;
                    if (kfr != NULL) delete kfr;}

  // ** function related to the Kalman filter:
  void estParam();             // estimating parameters by maximum
                               // likelihood
  void filter();               // the Kalman filter
  KalmanSRes smooth();         // smoothing (simple algorithm)
  KalmanSRes smooth_dejong();  // alternative smoothing algorithm
  KalmanPRes predict(const KalmanParam& kp2); //prediction

  // ** input
  void setXest_start(const nMatrix& m){ks->setXest_start(m);}
  void setVar_xest_start(const nMatrix& m){ks->setVar_xest_start(m);}

  // ** output
  KalmanFRes getKfr() const {return *kfr;}

  friend class KalmanFilterMixed;
};
```

KEYWORDS

kalman filter, state space model, time series, filtering, smoothing, prediction

DESCRIPTION

This is the main class for all methods and models related to Kalman filtering.

The state space model considered here consists of

the measurement equation:

y(t) = H(t) x(t) + v(t)

and the system equation (transition equation):

x(t) = F(t) x(t) + w(t)

in which y(t) is an (iydim x 1) matrix of the time series y and x(t) is an (ixdim x 1) state matrix. It is assumed that v(t) ~ N(0 , s2 x V(t)) and w(t) ~ N(0, s2 x W(t)) where V(t) is a fixed (iydim x iydim) matrix, W(T) a fixed (ixdim x ixdim) matrix, and s2 a scalar (which serve as a scaling factor in every variance- covariance matrix). All variances are scaled with this common scaling factor. The user should in advance scale W and V such that the largest elements are about 1.

The parameters in the model are then F(t), W(t), H(t) and V(t).

This is a very broad model class, which for instance include the linear regression model, ARIMA-models, and the so called structural models. Another name of the model is Dynamic Linear Model.

Assume that the time series y is observed from time tstart to tend. Then the following actions are appropriate:

- `Parameter estimation.` Some or all of F, W, H and V is assumed to depend upon an unknown parametervector Theta which is estimated by maximizing the likelihood using an optimizing algorithm.

- `Kalman filtering.` This is a recursive alogrithm from t=tstart to t=tend, which estimates x(t) given y(tstart), ..., y(t-1) and x(t) given y(tstart), ... , y(t). The estimates are written x(t;t-1) and x(t;t) with scaled corresponding covariance matrix Var_x(t;t-1) and Var_x(t;t). The scaling means that Var(x(t;t-1)) = s2 Var_x(t;t-1), and so on.

The Kalman filter also calculates the innovations z(t) = y(t) - H(t) x(t), i.e. one step ahead prediction error, with corresponding scaled covariance matrix. The log likelihood is calculated and the scaling factor s2 estimated.

The Kalman filter needs prior starting values for x(tstart-1 ; tstart-1) with the corresponding scaled covariance. One choice is the so called diffuse prior. Here, this is implemented with a large constant number at the diagonal of the covariance matrix.

- `Smoothing.` With smoothing means to estimate x(t) given all observations, that is x(t;tend), with corresponding scaled covariances.

- `Signal extraction.` This deals with estimating the signal sign(t) = H(t)x(t) given all observations, that is sign(t;tend) = H(t) x(t;tend), with corresponding scaled covariances.

- `Prediction.` This is to predict the state and the time series, with corresponding scaled covariances. The notation for the l-step ahead predictions are x(tend+l ; l) and y(tend+l ; tend).

Notation used in the class and the accompanying classes in the following subsections, are close to the notation used in the description above. The connection is:

```
C++            : above              :explanation
```

```
iydim          : iydim              :the dimension of the time
                                     series y(t)

ixdim          : ixdim              :the dim. of the state
                                     space nVector x(t)

tstart         : tstart             :usually the start of the
                                     observed time series, but
                                     sometimes other time
                                     points

tend           : tend               :usually the end of the
                                     observed timeseries, but
                                     sometimes other time
                                     points

tstartlikelihood                    :time for starting
                                     calculate the likelihood

loglikelihood  : log likelihood     :the log likelihood
                                     calculated by the Kalman
                                     filter

sigmasquared   : s2                 :the scaling factor used
                                     for every variance

F(t)           : F(t)

W(t)           : W(t)

H(t)           : H(t)

V(t)           : V(t)

xest(t)        : x(t;t)             :state space estimates

Var_xest(t)    : Var_x(t;t)         :scaled variance of state
                                     space estimates

xpred(t)       : x(t;t-1)           :state space
                                     predictions one step
                                     ahead OR

 -"-           : x(tend+1;tend)     :state space
                                     predictions l step
                                     ahead

Var_xpred(t)   : Var_x(t;t-1)       :scaled variance of
                                     state space predictions
                                     one steap ahead OR
```

```
 -"-           : Var_x(tend+1;tend)  :scaled variance of
                                      state space predictions
                                      1 step ahead

z(t)           : z(t)                :innovation

Var_z(t)       : Var_z(t)            :scaled variance of
                                      innovation

Var_z_inv      : inv(Var_z(t))       :the inverse of this

K(t)                                 :the so called Kalman gain

xest_start     : x(tstart-1;tstart-1):start values of state
                                      space estimates

Var_xest_start : x(tstart-1;tstart-1):start values of the
                                      variance of the state
                                      space estimates

xsmooth(t)     : x(t;tend)           :smoothed state nVector

Var_xsmooth(t) : Var_x(t;tend)       :scaled variance of the
                                      smoothed state nVector

signsmooth(t)  : sign(t;tend)        :signal estimate

Var_signsmooth : Var_sign(t;tend)    :scaled variance of the
                                      signal estimate

ypred(t)       : y(tend+1;tend)      :1 step ahead prediction
                                      of y

Var_ypred(t)   : Var_y(tend+1;tend)  :scaled variance of the
                                      1 step ahead prediction
                                      of y
```

The result of all member functions are objects of certain classes. The necesarry dimensions are private datamembers of each classes. In addition, the classes have various double, `nMatrix` or `ArrayGenSimple(nMatrix)` objects as data members. The most important classes and their data members are

```
class "KalmanParam":
  "ArrayGenSimple(nMatrix)" "F";
  "ArrayGenSimple(nMatrix)" "W";
  "ArrayGenSimple(nMatrix)" "H";
  "ArrayGenSimple(nMatrix)" "V";

class "KalmanStart":
  "nMatrix" "xest_start";
  "nMatrix" "Var_xest_start";
```

```
class "KalmanFRes":
  double loglikelihood;
  double sigmasquared;
  "ArrayGenSimple(nMatrix)" "xest";
  "ArrayGenSimple(nMatrix)" "Var_xest";
  "ArrayGenSimple(nMatrix)" "xpred";
  "ArrayGenSimple(nMatrix)" "Var_xpred";
  "ArrayGenSimple(nMatrix)" "z";
  "ArrayGenSimple(nMatrix)" "Var_z";
  "ArrayGenSimple(nMatrix)" "Var_z_inv";
  "ArrayGenSimple(nMatrix)" "K";

class "KalmanSRes":
  "ArrayGenSimple(nMatrix)" "xsmooth";
  "ArrayGenSimple(nMatrix)" "Var_xsmooth";
  "ArrayGenSimple(nMatrix)" "signsmooth";
  "ArrayGenSimple(nMatrix)" "Var_signsmooth";

class "KalmanPRes":
  "ArrayGenSimple(nMatrix)" "xpred";
  "ArrayGenSimple(nMatrix)" "Var_xpred";
  "ArrayGenSimple(nMatrix)" "ypred";
  "ArrayGenSimple(nMatrix)" "Var_ypred";
```

Overviews of the topic may be found in Harrison and Stevens (1976) and Harvey (1984). De Jong (1989) treats smoothing and signal extraction, and these algorithms are adopted here.

REFERENCES

Harrison, P. J. and Stevens, C. F.; 1976: "Bayesian forecasting", J.R.S.S., series B, Vol. 38, p. 205-247.

Harvey, A. C.; 1984: "A Unified View of Statistical Forecasting Procedures", Journ. of Forecasting, Vol 3, p. 245-275.

De Jong, P.; 1989: "Smoothing and Interpolation With the State-Space Model", J.A.S.A., Vol. 84, No. 408, p. 1085-1088.

CONSTRUCTORS AND INITIALIZATION

The class has three constructors. One takes the parameter and the observed data as input. Another constructor is designed for the case where some of the data are missing. This constructor takes the M matrix, which contains information about where the missing observations are, in addition. There is also a default constructor.

MEMBER FUNCTIONS

`estPar` - This function perform parameter estimation.

It uses the data members: class `KalmanParam` (the model parameters), class `KalmanStart` (starting values), and the data time series y.

The result of the estimation is stored in the data member kp.

`filter` - This function perform the Kalman filter.

It uses the data members: class `KalmanParam` (the model parameters), class `KalmanStart` (starting values), and the data time series y.

The result of the filter is stored in the data member kfr.

`smooth` - This function performs the smoothing. There are two variants of the function, see smooth_dejong below, which gives exactly the same result, but uses different private member functions.

`smooth` uses the data members: class `KalmanParam` (the model parameters), class `KalmanFRes` (result from the Kalman filter).

The output of the function is an object of class `KalmanSRes`, containing the smoothed values.

`smooth_dejong` - This function performs smoothing also. This variant of `smooth` additionally uses the data time series y.

The output of the function is an object of class `KalmanSRes`, containing the smoothed values.

`predict` -This function perform the l-step ahead predictions, l=1, 2, ..., usually for a time span after the span with observed data.

It takes as input an object of class `KalmanParam` (the model parameters, but usually with another time span than in the Kalman filter).

It also uses the private data member class `KalmanStart` (the starting values is, usually, assigned to last `xest`(t) and `Var_xest`(t) after a Kalman filter run).

The output of the function is an object of class `KalmanPRes`, containing the predictions.

FILES

KalmanFilter.C

EXAMPLE

```
#include <KalmanFilter.h>
#include <KalmanParamConst.h>
#include <KalmanLinGrowth.h>
#include <matrix.h>
#include <ArrayGenSimple_nMatrix.h>

int main()
{

  int tstartlikelihood=3, tstart=1, tend=100, iydim=1, ixdim=2;

  // Specification of time constant model parameters:
  // Note that the largest element of W an V are 1!
  nMatrix F(ixdim,ixdim);
  F(1,1)=1;
  F(1,2)=1;
  F(2,1)=0;
  F(2,2)=1;

  nMatrix W(ixdim,ixdim);
  W(1,1)=1;
  W(2,2)=0.2;
```

```
  nMatrix H(iydim,ixdim);
  H(1,1)=1;
  H(1,2)=0;
  nMatrix V(iydim,iydim);
  V(1,1)=0.5;

  // Putting the model parameters into a object of class KalmanParam,
  // with time span (tstart, tend) for Kalman filtering and smoothing,
  // and one with time span (tend+1, tend+20) for predictions):
  KalmanParamConst kp(tstart,tend,F,W,H,V);
  KalmanParamConst kp2(tend+1,tend+20,F,W,H,V);

  ArrayGenSimple(nMatrix) y(tend-tstart+1);
  for( int i=tstart; i<=tend; i++ )
     y(i).redim(1,1);

  FILE *fp;

  fp = fopen("data.file","r");
  float help;

  // Reads data into y:
  for( int m = 1; m <= 100; m++){
    fscanf(fp,"%f", &help);
    fprintf(stderr, "%f ", help);
    y(m)(1,1) = help;
  }

  // The Kalman filter:
  KalmanFilter kalman(kp, y);

//*********************************************************************
// If observation 2,3, 8 and 76 where missing, kalman would be declared
// as follows:
//
//  ArrayGenSimple(nMatrix) M(tend-tstart+1);
//  for( i=tstart; i<=tend; i++ ){
//    M(i).redim(1,1);
//  }
//
// // indicating where the missing observations are
//  M(2)(1,1)=1;
//  M(3)(1,1)=1;
//  M(8)(1,1)=1;
//  M(76)(1,1)=1;
//
// // The Kalman filter with missing observations:
//  KalmanFilter kalman(kp, y, M);;
//*********************************************************************

  kalman.filter();

  // The last estimated state space vector with corresponding
  // scaled covariance are start values for the predictions:
  kalman.setXest_start(kalman.getKfr().xest(tend));
  kalman.setVar_xest_start(kalman.getKfr().Var_xest(tend));

  // The prediction 20 time steps ahead:
  KalmanPRes kpr=kalman.predict(kp2);

  // smoothing with the following two functions gives the same result!
  KalmanSRes ksr1=kalman.smooth();
  KalmanSRes ksr2=kalman.smooth_dejong();
```

```
// Assume now a linear growth model with unknown parameters

// Putting the model parameters into a object of class KalmanLinGrowth.
  KalmanLinGrowth kpLG(tstart,tend);

// The Kalman filter:
  KalmanFilter kalmanLG(kpLG, y);

// Estimating the parameters:
  kalmanLG.estParam();

// Printing the resulted estimate:
  kpLG.getTheta().print(cout);

// After the estimation the other functions is used on exactly the same
// way as above

  kalmanLG.filter();

  kalmanLG.setXest_start(kalmanLG.getKfr().xest(tend));
  kalmanLG.setVar_xest_start(kalmanLG.getKfr().Var_xest(tend));

  KalmanParam kp3(tend, tend+20, kalmanLG.kp->F(tend),
                                 kalmanLG.kp->W(tend),
                                 kalmanLG.kp->H(tend),
                                 kalmanLG.kp->V(tend));

  KalmanPRes kprLG=kalmanLG.predict(kp3);

  KalmanSRes ksrLG1=kalmanLG.smooth();
  KalmanSRes ksrLG2=kalmanLG.smooth_dejong();
}
```

SEEALSO

class KalmanParam, class KalmanStart, class KalmanFRes, class KalmanSRes, class KalmanPRes, class KalmanLinGrowth and class KalmanRegAR.

AUTHOR

Magne Aldrin and Tove Andersen, NR

3.1.2 KalmanParam

NAME

KalmanParam - a base class for parameters in the state space model (Kalman filter model)

INCLUDE

```
include "KalmanParam.h"
```

SYNTAX

```
class KalmanParam
{
protected:
  int tstart;
  int tend;
  int iydim;
  int ixdim;

public:
  ArrayGenSimple(nMatrix) F;
  ArrayGenSimple(nMatrix) W;
  ArrayGenSimple(nMatrix) H;
  ArrayGenSimple(nMatrix) V;

  // ** constructors:
  KalmanParam(int tstart2, int tend2);
  KalmanParam(int tstart2, int tend2, int iydim2, int ixdim2);
  KalmanParam(const KalmanParam& kp);
  KalmanParam(const ArrayGenSimple(nMatrix)& F2,
              const ArrayGenSimple(nMatrix)& W2,
              const ArrayGenSimple(nMatrix)& H2,
              const ArrayGenSimple(nMatrix)& V2);
  KalmanParam(int tstart2, int tend2, const nMatrix& F2,
                                      const nMatrix& W2,
                                      const nMatrix& H2,
                                      const nMatrix& V2);

  // ** output:
  int getTstart() const         {return tstart;}
  int getTend()   const         {return tend;}
  int getIydim()  const         {return iydim;}
  int getIxdim()  const         {return ixdim;}
  // ** friend classes:
  friend class KalmanFilter;
};
```

KEYWORDS

kalman filter, state space model, time series

DESCRIPTION

The class contains the parameters in the state space model (Kalman filter model). This includes two time dependent matrices in the system and the observation equations, and the two (scaled) covariance-matrices of the noise.

CONSTRUCTORS AND INITIALIZATION

The class has five constructors, including a copy constructor. One constructor takes the time span as parameters, and initialize all datamembers of type `ArrayGenSimple(nMatrix)` to 0. Another takes in addition the y and x dimensions as parameters. One constructor takes four `ArrayGenSimple(nMatrix)` objects as parameters, and the last one take the time span and four `nMatrix`. See documentation of class `KalmanFilter` for an explanation of the contructor parameters.

FILES

KalmanParam.C

EXAMPLE

See documentation of class `KalmanFilter`.

SEEALSO

class KalmanFilter, class KalmanParamConst, class KalmanParamEst.

AUTHOR

Magne Aldrin and Tove Andersen, NR

3.1.3 KalmanParamConst

NAME

KalmanParamConst - parameters in the state space model (Kalman filter model) when the parameters are known

INCLUDE

```
include "KalmanParamConst.h"
```

SYNTAX

```
class KalmanParamConst:public KalmanParam
{
public:
  // ** constructors:
  KalmanParamConst(int tstart2, int tend2);
  KalmanParamConst(int tstart2, int tend2, int iydim2, int ixdim2);
  KalmanParamConst(const KalmanParamConst& kp);
  KalmanParamConst(const ArrayGenSimple(nMatrix)& F2,
              const ArrayGenSimple(nMatrix)& W2,
              const ArrayGenSimple(nMatrix)& H2,
              const ArrayGenSimple(nMatrix)& V2);
  KalmanParamConst(int tstart2, int tend2, const nMatrix& F2,
                   const nMatrix& W2, const nMatrix& H2, const nMatrix& V2);
};
```

KEYWORDS

kalman filter, state space model, time series

DESCRIPTION

The class inherits class KalmanParam which contains the parameters in the state space model (Kalman filter).

The class contains the known parameters in the state space model (Kalman filter model).

FILES

KalmanParamConst.C

EXAMPLE

See documentation of class `KalmanFilter`.

SEEALSO

class KalmanParam and class KalmanFilter.

AUTHOR

Tove Andersen, NR

3.1.4 KalmanParamEst

NAME

KalmanParamEst - an abstract class containing the parameters in the state space model (Kalman filter model) when estimation is to be undertaken

INCLUDE

```
include "KalmanParamEst.h"
```

SYNTAX

```
class KalmanParamEst:public KalmanParam
{
protected:
  nVector Theta;         // the unknown parameter nVector
  nVector ThetaTrans;    // the transformed unknown parameter nVector
  nIntVector index;      // index vector needed for checking the scaling
  BooLean nBounds;       // if bounds on ThetaTrans is provided (dpTRUE)
                         // or not (dpFALSE=default)
  nVector UpperBounds;   // if nBounds == dpTRUE, containing the upper
                         // bounds of ThetaTrans
  nVector LowerBounds;   // if nBounds == dpTRUE, containing the lower
                         // bounds of ThetaTrans
  double optim_tol;      // the accurancy of the estimated ThetaTrans,
                         // optim_tol < 1. (Default 0.001)
  double linesearch_tol; // internal parameter in the optimizing,
                         // 0.0 <= linesearch_tol < 1.0 (Default 0.5)
  BooLean nPrint;        // if printout of the optimizing routine is
                         // required (dpTRUE) or not (dpFALSE=default)
public:
  // ** constructors:
  KalmanParamEst(int tstart2, int tend2);
  KalmanParamEst(int tstart2, int tend2, int iydim2, int ixdim2);
  KalmanParamEst(const KalmanParamEst& kp);

  virtual void setParam(const nVector& ThetaTrans2){;};

  void setOptim_tol(double opt);
  void setLinesearch_tol(double lin);

  nVector getTheta() const {return Theta;}
  nIntVector getIndex() const {return index;}
  nVector getThetaTrans() const {return ThetaTrans;}
  friend class KalmanFilter;
};
```

KEYWORDS

kalman filter, state space model, time series, estimation

DESCRIPTION

The class inherits class KalmanParam which contains the parameters in the state space model (Kalman filter).

KalmanParamEst contains data members needed when one estimates the parameters in the state space model. These data members are explained in the class description.

When the user wants to perform parameter estimation, he must declare his own class derived of this class. In the users derived class, a constructor where all data members are initialisized must be constructed. The user must also declare the function setParam. For examples see class KalmanLinGrowth and class KalmanRegAR.

MEMBER FUNCTIONS

The member functions `setOptim_tol` and `setLinesearch_tol` set internal parameters in the optimizing routine, used for maximizing the loglikelihood. For more information see function e04jbc in `Nag C Library Manual, Mark 3`, Volume 2, 1994.

The function `setParam` sets the value for Theta, ThetaTrans, H, F, V and W from a given value of the transformed theta, ThetaTrans2.

The rest of the member functions are self-explanatory. They all perform output of the private data members.

FILES

KalmanParamEst.C

EXAMPLE

See documentation of class `KalmanFilter`.

SEEALSO

class KalmanParam, class KalmanFilter, class KalmanLinGrowth, and class KalmanRegAR.

AUTHOR

Tove Andersen, NR

3.1.5 KalmanLinGrowth

NAME

KalmanLinGrowth - The Linear Growth model

INCLUDE

```
include "KalmanLinGrowth.h"
```

SYNTAX

```
class KalmanLinGrowth:public virtual KalmanParamEst
{
public:
  // ** constructors:
  KalmanLinGrowth(int tstart2, int tend2);

  // ** setting the parameters:
  void setParam(const nVector& ThetaTrans2);
};
```

KEYWORDS

kalman filter, state space model, time series, linear growth

DESCRIPTION

This class inherits KalmanParamEst and it contains the parameters in the linear growth model, i.e. the respons variable, y, is

$$y_t = x_t + v_t$$

where v is white noise with unknown variance and the state space vector:

$$x_{1,t} = x_{1,t-1} + x_{2,t-1} + w_{1,t}$$

$$x_{2,t} = x_{2,t-1} + w_{2,t}$$

Here the ws are white noise with unknown variances.

This class is used as input parameter to the constructors of class `KalmanFilter`. By using the member function `estParam` of class `KalmanFilter`, the variances in the model are estimated by maximum likelihood. The estimated values are found in the parameter vector `Theta`.

This class is also an example of the kind of classes the users need to define when he wants to perform maximimum likelihood estimation of the parameters in a specific model. See the source code for how this is done.

FILES

KalmanLinGrowth.C

EXAMPLE

```
#include <KalmanFilter.h>
#include <KalmanLinGrowth.h>
#include <matrix.h>
#include <ArrayGenSimple_nMatrix.h>

int main()
{
  int tstartlikelihood= ;
  int tstart=  ;
  int tend= ;
  int iydim=1, ixdim=2;

  ArrayGenSimple(nMatrix) y(tend-tstart+1);
  y.setBase(tstart);

  for( i=tstart; i<= tend; i++)
    y(i) = nMatrix(iydim,1);

  //read data
  ...
  ...

  KalmanLinGrowth kp(tstart, tend);
  KalmanFilter kalman(kp, y);

  //estimate theta
  kalman.estParam();

// Printing the resulted estimate:
  kp.getTheta().print(cout);
}
```

SEEALSO

class KalmanParamEst, class KalmanFilter.

AUTHOR

Tove Andersen, NR

3.1.6 KalmanRegAR

NAME

KalmanRegAR - Regression with autoregressive errors

INCLUDE

```
include "KalmanRegAR.h"
```

SYNTAX

```
class KalmanRegAR:public virtual KalmanParamEst
{
private:
  int p;      // no. of explanatory variables
  int par;    // no. of AR-parameters

public:
  // ** constructors:
  KalmanRegAR(int par, const nMatrix& X); // input are the AR-order
                                          // and the matrix of
                                          // explanatory variables

  // ** functions needed for estimating parameters:
  void setParam(const nVector& Theta2);
};
```

KEYWORDS

kalman filter, state space model, time series

DESCRIPTION

This class inherits KalmanParamEst and it contains the parameters for regression with autoregressive errors. The model is

```
       t
 y  = b u  + n
  t      t    t
```

where y is the respons variable (at time t), u a p-dimensional vector of explanatory variables (at time t) and b the vector of regression coefficients. The noise n follows an autoregressive model of order par:

```
 n  = phi  n    + phi  n    + ... + phi    n      + e
  t      1  t-1      2  t-2            par  t-par    t
```

where the phi-s are the AR parameters and e is white noise. This is implemented by the Kalman filter with:

```
b: the state vector x
u: the H matrix
```

This class is used as input parameter to the constructors of class `KalmanFilter`. By using the member function `estParam` of class `KalmanFilter`, the phi-s are estimated by maximum likelihood. The estimated values are found in the parameter vector `Theta`.

This class is also an example of the kind of classes the users need to define when he wants to perform maximimum likelihood estimation of the parameters in a specific model. See the source code for how this is done.

FILES

KalmanRegAR.C

EXAMPLE

```
#include <KalmanRegAR.h>
#include <KalmanFilter.h>
#include <matrix.h>
#include <ArrayGenSimple_nMatrix.h>

nMatrix scale2var1(const nMatrix& Y);

int main()
{

  int tstart = ;
  int tend   = ;

  int iydim  = ;
  int ixdim  = ;

  int nofObs = ;
  int par    = ;

  ArrayGenSimple(nMatrix) y(tend-tstart+1);
  y.setBase(tstart);

  nMatrix x(nofObs, ixdim);

// reads data
  ......

  KalmanRegAR kp(par,x);
  KalmanFilter kalman(kp, y);

  kalman.estParam();
}
```

SEEALSO

class KalmanParamEst, class KalmanParam, class KalmanFilter.

AUTHOR

Magne Aldrin, NR

3.2 Non linear state space modelling

3.2.1 KalmanFilterMixed

NAME

KalmanFilterMixed - the main class for mixed processes in dynamic linear models.

INCLUDE

```
include "KalmanFilterMixed.h"
```

SYNTAX

```
class KalmanFilterMixed:public virtual KalmanFilter
{
  KalmanParamMixed* mp;
  long int seed;

public:

  // ** constructors:
  KalmanFilterMixed(const KalmanParamMixed& mp2,
              const KalmanStart& start,
              const ArrayGenSimple(nMatrix)& maty);

  // ** destructors:
  ~KalmanFilterMixed() { if (mp != NULL) delete mp; }

  // ** functions:
  void setkp(const nIntVector& C);
  void setSeed(const long int seed2);

  void mixedFilter(const int ny, const int B,
                   const nIntVector* indexEst=NULL);

  KalmanParamMixed getmp() const { return *mp; }

};
```

KEYWORDS

dynamic linear models, kalman filter, state space model, time series, filtering.

DESCRIPTION

The class inherits class `KalmanFilter`, which is the main class for all models and methods related to Kalman filtering.

This class is the main class for mixed processes in state space models. The state space model is assumed to depend upon an underlying discrete stocastic process {C(t)}. The dependence of the underlying process is restricted to be in F and W, such that F(t) = F(C(t)) and W(t)=W(C(t)). The underlying process is assumed to be a Markov chain,

and H and V is constrained to be independent of t. The state space model considered may then be written:

the measurement equation:

$y(t) = H\ x(t) + v(t)$

and the system equation (transition equation):

$x(t) = F(C(t))\ x(t) + w(t)$

where v(t) is N(0, V) and w(t) is N(0, W(C(t))), and {C(t)} is an unobservable Markov chain with transition probability matrix P. The possible values of C(t) is {1,...,K}. The parameters of the model is contained in the class `KalmanParamMixed`.

A stochasting simulation method, based upon the Metropolis algorithm, is used to estimate the underlying Markov chain. If some of the parameters in the state space model are unknown, they can be estimated together with the estimation of the Markov chain.

REFERENCES

Andre Teigland and Tove Andersen, Mixed processes in Dynamic Linear Models, NR-notat, STAT/17/95.

CONSTRUCTORS AND INITIALIZATION

There is one constructor which takes the parameters of the model, the start values used in the Kalman filter and the observed data as input.

MEMBER FUNCTIONS

`setkp` - for a given realization of the underlying process {C(t)} the function setkp sets F(C(t)) and W(C(t)) for all t.

`setSeed` - sets the start seed used by `filterMixed`. If the seed is not set, `filterMixed` uses the time as start seed.

`filterMixed` - estimates the underlying Markov chain and the result is stored in class `KalmanParamMixed`. The Metropolis algorithm with ny repetitions is used, and the function calculates B realizations of the Markov chain. After {C(t)} is estimated the Kalman filter for the process, with parameters corresponding to the estimated Markov chain, is run and the result is stored in the data member kfr. If some parameters in the model are unknown this is shown by a 1 in the vector IndexEst. The length of the vector should be 4 and the four elements correspond to, in this order, H, V, F and W. Start values for the unknown parameters must be given in the class `KalmanParamMixed` and these will be substituted with the estimated values.

After `filterMixed` is run or `setkp` is used to set F and W for a given Markov chain, the smoothing and prediction functions of class `KalmanFilter` can be used.

FILES

KalmanFilterMixed.C

EXAMPLE

```
#include <KalmanFilterMixed.h>
#include <matrix.h>
#include <ArrayGenSimple_nMatrix.h>

int main()
{

  int t, i, j, tstartlikelihood, tstart=1, tend=80, iydim=1, ixdim=1;

  // Specifications of time constant model parameters:
  nMatrix H(iydim, ixdim);
  H(1,1)=1;

  nMatrix V(iydim, iydim);
  V(1,1) = 0.01;

  ArrayGenSimple(nMatrix) y(tend-tstart+1);
  y.setBase(tstart);

  for( i=tstart; i<= tend; i++)
    y(i) = nMatrix(iydim,1);

  int m;
  float help;

  FILE *fp;
  fp = fopen("data","r");

  // Reads data into y:
  for( m = 1; m <= (tend-tstart+1); m++){
    fscanf(fp,"%f", &help);
    y(m)(1,1) = help;
  }

  //start values
  KalmanStart ks(ixdim);

  nMatrix help1(1,1);

  ks.setXest_start(help1);

  help1(1,1)=0.01;

  ks.setVar_xest_start(help1);

  // 2 possible states of the Markov chain
  int K = 2;

  // the transition probability matrix
  nMatrix P(2,2);

  P(1,1) = 0.95;
  P(1,2) = 0.05;
  P(2,1) = 0.05;
  P(2,2) = 0.95;

  // the parameters in the system equation. They both have two possible
  // values, one for state 1 of the Markov chain and one for state 2.
  // In this example F is independent of the Markov chain.
  ArrayGenSimple(nMatrix) FK(K);

  FK(1) = nMatrix(ixdim, ixdim);
  FK(2) = nMatrix(ixdim, ixdim);

  FK(1)(1,1)=1;
```

```
  FK(2)(1,1)=1;

  ArrayGenSimple(nMatrix) WK(K);

  WK(1) = nMatrix(ixdim, ixdim);
  WK(2) = nMatrix(ixdim, ixdim);

  WK(1)(1,1)=0.01;
  WK(2)(1,1)=1;

  // the kalman parameters
  KalmanParamMixed mp(ixdim, iydim, H, V, FK, WK, P);

  // the mixed Kalman filter
  KalmanFilterMixed kal(mp, ks, y);

  //set the start seed used in mixedFilter. This is optional. If the seed
  //is not set, the time is used as the start seed.

  long int seed = 86987762;

  kal.setSeed(seed);
  // running the estimation of the Markov process and the kalman filter.
  // 100 repetitions in the metropolis algorithm, 50 times.
  kal.mixedFilter(100, 50);

  //printing the estimated Markov chain
  (kal.getmp()).getMarkovChain().print("FILE=Mcres");

  //printing the estimated state space vector
  (kal.getKfr()).xest.print("FILE=xres");

//************************************************************************
// If W is to be estimated: 0.01 and 1 is now assumed to be the initial
// values for W.
//
// nIntVector index(4);
//
// index(4)=1;
//
// kal.mixedFilter(100, 50, &index);
//
// //printing the estimated parameter W
// (kal.getmp()).mixW.print("FILE=xres");
//************************************************************************
}
```

SEEALSO

class KaLmanParamMixed and class KalmanFilter.

AUTHOR

Tove Andersen, NR

3.2.2 KalmanParamMixed

NAME

KalmanParamMixed - parameters for mixed processes in dynamic linear models

INCLUDE

```
include "KalmanParamMixed.h"
```

SYNTAX

```
class KalmanParamMixed
{
  int mixdim;       // number of possible states in the Markov cahin
  int ixdim, iydim; // the dimensions in the state space model

  nMatrix P;        // the transition probability matrix of the Markov chain
  nIntVector C;     // the Markov chain (estimated)

  nMatrix H;        // parameter in the measurement equation
  nMatrix V;        // parameter in the measurement equation

public:
  ArrayGenSimple(nMatrix) mixF;  // parameter in the system equation,
                                 // given for all posible states of the
                                 // Markov chain.
  ArrayGenSimple(nMatrix) mixW;  // parameter in the system equation,
                                 // given for all posible states of the
                                 // Markov chain.
  // ** constructors:
  KalmanParamMixed(int ixdim2, int iydim2,
                   const nMatrix H2, const nMatrix V2,
                   const ArrayGenSimple(nMatrix)& mixF2,
                   const ArrayGenSimple(nMatrix)& mixW2,
                   const nMatrix& P2);
  KalmanParamMixed(const KalmanParamMixed& mp2);

  // ** output
  nIntVector getMarkovChain() const {return C;}
  nMatrix getH() const {return H;}
  nMatrix getV() const {return V;}

  friend class KalmanFilterMixed;
};
```

KEYWORDS

dynamic linear model, kalman filter, state space model, time series

DESCRIPTION

The class contains the parameters for mixed processes in state space models. The model are described in class `KalmanFilterMixed`.

CONSTRUCTORS AND INITIALIZATION

The constructor takes as input: the x- and y-dimension in the model, the parameters, H and V, in the measurement equation, the system parameters, F and W, as a functions of the K possible states of the Markov chain and finally the transition probability matrix for the Markov chain. There is also a copy constructor.

FILES

KalmanParamMixed.C

EXAMPLE

See documentation of class `KalmanFilterMixed`.

SEEALSO

class KalmanFilterMixed.

AUTHOR

Tove Andersen, NR

3.3 Result and data classes for linear and non linear state space modelling.

3.3.1 KalmanFRes

NAME

KalmanFRes - filtered state space vector and signal after Kalman filtering

INCLUDE

```
include "KalmanFRes.h"
```

SYNTAX

```
class KalmanFRes
{
private:
  int tstart;
  int tend;
  int iydim;
  int ixdim;

  double loglikelihood;
  double sigmasquared;

public:
  ArrayGenSimple(nMatrix) xest;
  ArrayGenSimple(nMatrix) Var_xest;
  ArrayGenSimple(nMatrix) xpred;
  ArrayGenSimple(nMatrix) Var_xpred;
  ArrayGenSimple(nMatrix) z;
  ArrayGenSimple(nMatrix) Var_z;
  ArrayGenSimple(nMatrix) Var_z_inv;
  ArrayGenSimple(nMatrix) K;

  // ** constructors:
  KalmanFRes(int tstart, int tend, int iydim, int ixdim);
  KalmanFRes(int tstart, int tend, int iydim, int ixdim,
             const ArrayGenSimple(nMatrix)& M2);
  KalmanFRes(const KalmanFRes& kfr);

  // ** output:
  int getTstart() const         {return tstart;}
  int getTend()   const         {return tend;}
  int getIydim()  const         {return iydim;}
  int getIxdim()  const         {return ixdim;}

  double getLoglikelihood() const  {return loglikelihood;}
  double getSigmasquared()  const  {return sigmasquared;}

  // ** friend classes:
  friend class KalmanFilter;
};
```

KEYWORDS

kalman filter, state space model, time series, filtering

DESCRIPTION

The class contains the result of one run through the Kalman filter. This includes predicted and estimated state space vector x (x(t;t-1) and x(t;t)) and the innovations z=y(t)-H(t)*x(t;t-1), all with corresponcing (scaled) covariance matrices. In additon, the Kalman gain, the loglikelihood, and the scaling factor sigmasquared are included in the class.

CONSTRUCTORS AND INITIALIZATION

There are three constructors. The first constructor takes the time span and the y and x dimensions as parameters, and initialize all data member of type `ArrayGenSimple(nMatrix)` to zero. The second constructor take in additon the matrix indicating missing observations. The third constructor is a copy-constructor.

FILES

KalmanFRes.C

EXAMPLE

See documentation of class `KalmanFilter`.

SEEALSO

class KalmanFilter

AUTHOR

Magne Aldrin, NR

3.3.2 KalmanPRes

NAME

KalmanPRes - predicted state space vector and ts/signal after Kalman predict.

INCLUDE

```
include "KalmanPRes.h"
```

SYNTAX

```
class KalmanPRes
{
private:
  int tstart;
  int tend;
  int iydim;
  int ixdim;

public:
  ArrayGenSimple(nMatrix) xpred;
  ArrayGenSimple(nMatrix) Var_xpred;
  ArrayGenSimple(nMatrix) ypred;
  ArrayGenSimple(nMatrix) Var_ypred;

  // ** constructors:
  KalmanPRes(int tstart, int tend, int iydim, int ixdim);
  KalmanPRes(const KalmanPRes& kpr);

  // ** output:
  int getTstart() const         {return tstart;}
  int getTend() const           {return tend;}
  int getIydim() const          {return iydim;}
  int getIxdim() const          {return ixdim;}

  // ** friend classes:
  friend class KalmanFilter;
};
```

KEYWORDS

kalman filter, state space model, time series, prediction

DESCRIPTION

The class contains predicted state space vector and time series (= predicted signal) with (scaled) covariance-matrices, after Kalman prediction. The class is the return value of the member function `predict` of class `KalmanFilter`. See documentation of class `KalmanFilter`.

FILES

KalmanPRes.C

EXAMPLE

See documentation of class `KalmanFilter`.

SEEALSO

class KalmanFilter.

AUTHOR

Magne Aldrin, NR

3.3.3 KalmanSRes

NAME

KalmanSRes - smoothed state space vector and signal after Kalman smoothing

INCLUDE

```
include "KalmanSRes.h"
```

SYNTAX

```
class KalmanSRes
{
private:
  int tstart;
  int tend;
  int iydim;
  int ixdim;

public:
  ArrayGenSimple(nMatrix) xsmooth;
  ArrayGenSimple(nMatrix) Var_xsmooth;
  ArrayGenSimple(nMatrix) signsmooth;
  ArrayGenSimple(nMatrix) Var_signsmooth;

  // ** constructors:
  KalmanSRes(int tstart, int tend, int iydim, int ixdim);
  KalmanSRes(const KalmanSRes& ksr);

  // ** output:
  int getTstart() const         {return tstart;}
  int getTend()   const         {return tend;}
  int getIydim()  const         {return iydim;}
  int getIxdim()  const         {return ixdim;}

  // ** friend classes:
  friend class KalmanFilter;
};
```

KEYWORDS

kalman filter, state space model, time series, smoothing

DESCRIPTION

The class contains smoothed state space vector and signal with (scaled) covariance-matrices, after Kalman smoothing according to the method of De Jong (1989). The class is the return value of the member function `smooth` of class `KalmanFilter`. See documentation of class `KalmanFilter`.

FILES

KalmanSRes.C

EXAMPLE

See documentation of class `KalmanFilter`.

SEEALSO

class KalmanFilter

AUTHOR

Magne Aldrin, NR

3.3.4 KalmanTRes

NAME

KalmanTRes - help matrices needed for Kalman smoothing

INCLUDE

```
include "KalmanTRes.h"
```

SYNTAX

```
class KalmanTRes
{
private:
  int tstart;
  int tend;
  int iydim;
  int ixdim;

  ArrayGenSimple(nMatrix) r;
  ArrayGenSimple(nMatrix) R;
  ArrayGenSimple(nMatrix) n;
  ArrayGenSimple(nMatrix) N;

public:
  // ** constructors:
  KalmanTRes(int tstart, int tend, int iydim, int ixdim);
  KalmanTRes(const KalmanTRes& ktr);

  // ** friend classes:
  friend class KalmanFilter;
};
```

KEYWORDS

kalman filter

DESCRIPTION

The class contains some help matrices needed for Kalman smoothing, according to the method of De Jong (1989). The class is the return value of the private member function `tmp` of class `KalmanFilter`. See documentation of class `KalmanFilter`.

FILES

KalmanTRes.C

SEEALSO

class KalmanFilter and class KalmanSRes

AUTHOR

Magne Aldrin, NR

3.3.5 KalmanStart

NAME

KalmanStart - starting values for the Kalman filter

INCLUDE

```
include "KalmanStart.h"
```

SYNTAX

```
class KalmanStart
{
private:
  int ixdim;
  int tstartlikelihood;  // time for starting calculate the likelihood

  nMatrix xest_start;
  nMatrix Var_xest_start;

public:
  // ** constructors:
  KalmanStart(int ixdim);
  KalmanStart(const KalmanStart& ks);

  // ** input:
  void setXest_start(const nMatrix& m)  { this->xest_start.setEqual(m) ;}
  void setVar_xest_start(const nMatrix& m) { this->Var_xest_start.setEqual(m);}
  void setTstartlikelihood(int ts=1)  { tstartlikelihood = ts;}

  // ** output:
  int getIxdim() const          {return ixdim;}
  nMatrix getXest_start() const       {return xest_start;}
  nMatrix getVar_xest_start() const   {return Var_xest_start;}
  int getTstartlikelihood() const {return tstartlikelihood;}

  // ** friend classes:
  friend class KalmanFilter;
};
```

KEYWORDS

kalman filter, state space model, time series

DESCRIPTION

The class contains starting values for the Kalman filter, which mean "estimated" state space vector and (scaled) covariance-nMatrix at time tstart-1, when the Kalman filter starts at tstart. Default initialization values are with "infinity variance" and tstartlikelihood equal to ixdim+1. If the user wants other values this should be set by member functions.

FILES

KalmanStart.C

EXAMPLE

See documentation of class `KalmanFilter`.

SEEALSO

class KalmanFilter, class KalmanParam, class KalmanFRes, class KalmanSRes, class KalmanPRes.

AUTHOR

Magne Aldrin, NR

Index