Dynare++ DSGE solver

solves higher order approximation to a decision rule of a
Dynamic Stochastic General Equilibrium model about
deterministic and stochastic fix point

1. Utilities.

2. Exception. Start of `kord_exception.h` file.
This is a simple code defining an exception and two convenience macros.

```
#ifndef KORD_EXCEPTION_H
#define KORD_EXCEPTION_H
#include <cstring>
#include <cstdio>
#define KORD_RAISE(mes) throw KordException(__FILE__, __LINE__, mes);
#define KORD_RAISE_IF(expr, mes)
  if (expr) throw KordException(__FILE__, __LINE__, mes);
#define KORD_RAISE_X(mes, c) throw KordException(__FILE__, __LINE__, mes, c);
#define KORD_RAISE_IF_X(expr, mes, c)
  if (expr) throw KordException(__FILE__, __LINE__, mes, c);
  ⟨ KordException class definition 3 ⟩;
  ⟨ KordException error code definitions 4 ⟩;
#endif
```

3.

```
⟨ KordException class definition 3 ⟩ ≡
  class KordException {
  protected:
    char fname[50];
    int lnum;
    char message[500];
    int cd;
  public:
    KordException(const char *f, int l, const char *mes, int c = 255)
    {
      strncpy(fname, f, 50); fname[49] = '\0';
      strncpy(message, mes, 500); message[499] = '\0';
      lnum = l;
      cd = c;
    }
    virtual ~KordException() {}
    virtual void print() const
    { printf("At␣%s:%d:(%d):%s\n", fname, lnum, cd, message); }
    virtual int code() const
    { return cd; }
    const char *get_message() const
    { return message; }
  };
```

This code is used in section 2.

4.

```
⟨ KordException error code definitions 4 ⟩ ≡
#define KORD_FP_NOT_CONV  254
#define KORD_FP_NOT_FINITE  253
#define KORD_MD_NOT_STABLE  252
```

This code is used in section 2.

5. End of `kord_exception.h` file.

6. Resource usage journal. Start of `journal.h` file.

```
#ifndef JOURNAL_H
#define JOURNAL_H
#include "int_sequence.h"
#include <sys/time.h>
#include <cstdio>
#include <iostream>
#include <fstream>
  ⟨ SystemResources class declaration 7 ⟩;
  ⟨ SystemResourcesFlash struct declaration 8 ⟩;
  ⟨ Journal class declaration 11 ⟩;
  ⟨ JournalRecord class declaration 9 ⟩;
  ⟨ JournalRecordPair class declaration 10 ⟩;
#endif
```

7.

```
⟨ SystemResources class declaration 7 ⟩ ≡
  class SystemResources {
    timeval start;
  public:
    SystemResources();
    static long int pageSize();
    static long int physicalPages();
    static long int onlineProcessors();
    static long int availableMemory();
    void getRUS(double &load_avg, long int &pg_avail, double &utime, double &stime, double
        &elapsed, long int &idrss, long int &majflt);
  };
```

This code is used in section 6.

8.

```
⟨ SystemResourcesFlash struct declaration 8 ⟩ ≡
  struct SystemResourcesFlash {
    double load_avg;
    long int pg_avail;
    double utime;
    double stime;
    double elapsed;
    long int idrss;
    long int majflt;
    SystemResourcesFlash();
    void diff(const SystemResourcesFlash &pre);
  };
```

This code is used in section 6.

9.

```
#define MAXLEN 1000
⟨ JournalRecord class declaration 9 ⟩ ≡
  class JournalRecord;
  JournalRecord &endrec(JournalRecord &);
  class JournalRecord {
  protected:
    char recChar;
    int ord;
  public:
    Journal &journal;
    char prefix[MAXLEN];
    char mes[MAXLEN];
    SystemResourcesFlash flash;
    typedef JournalRecord &(*_Tfunc)(JournalRecord &);
    JournalRecord(Journal &jr, char rc = 'M')
    : recChar(rc), ord(jr.getOrd()), journal(jr) { prefix[0] = '\0';
      mes[0] = '\0';
      writePrefix(flash); }
    virtual ~JournalRecord() { }
    JournalRecord &operator≪(const IntSequence &s);
    JournalRecord &operator≪(_Tfunc f)
    { (*f)(*this);
      return *this; }
    JournalRecord &operator≪(const char *s)
    { strcat(mes, s);
      return *this; }
    JournalRecord &operator≪(int i)
    { sprintf(mes + strlen(mes), "%d", i);
      return *this; }
    JournalRecord &operator≪(double d)
    { sprintf(mes + strlen(mes), "%f", d);
      return *this; }
  protected:
    void writePrefix(const SystemResourcesFlash &f);
  };
```

This code is used in section 6.

10.

```
⟨ JournalRecordPair class declaration 10 ⟩ ≡
  class JournalRecordPair : public JournalRecord {
    char prefix_end[MAXLEN];
  public:
    JournalRecordPair(Journal &jr)
    : JournalRecord(jr, 'S') { prefix_end[0] = '\0';
      journal.incrementDepth(); }
    ~JournalRecordPair();
  private:
    void writePrefixForEnd(const SystemResourcesFlash &f);
  };
```

This code is used in section 6.

11.

```
⟨ Journal class declaration 11 ⟩ ≡
  class Journal : public ofstream {
    int ord;
    int depth;
  public:
    Journal(const char *fname)
    : ofstream(fname), ord(0), depth(0) { printHeader(); }
    ~Journal()
    { flush(); }
    void printHeader();
    void incrementOrd()
    { ord++; }
    int getOrd() const
    { return ord; }
    void incrementDepth()
    { depth++; }
    void decrementDepth()
    { depth--; }
    int getDepth() const
    {
      return depth;
    }
  };
```

This code is used in section 6.

12. End of `journal.h` file.

13. Start of `journal.cpp` file.

```
#include "journal.h"
#include "kord_exception.h"
#if ¬defined (__MINGW32__)
#include <sys/resource.h>
#include <sys/utsname.h>
#endif
#include <cstdlib>
#include <unistd.h>
#include <ctime>
  SystemResources _sysres;
#if defined (__MINGW32__)
  ⟨ sysconf Win32 implementation 28 ⟩;
#endif
#if defined (__APPLE__)
#define _SC_PHYS_PAGES  2
#define _SC_AVPHYS_PAGES  3
#endif
  ⟨ SystemResources constructor code 14 ⟩;
  ⟨ SystemResources::pageSize code 15 ⟩;
  ⟨ SystemResources::physicalPages code 16 ⟩;
  ⟨ SystemResources::onlineProcessors code 17 ⟩;
  ⟨ SystemResources::availableMemory code 18 ⟩;
  ⟨ SystemResources::getRUS code 19 ⟩;
  ⟨ SystemResourcesFlash constructor code 20 ⟩;
  ⟨ SystemResourcesFlash::diff code 21 ⟩;
  ⟨ JournalRecord::operator≪ symmetry code 22 ⟩;
  ⟨ JournalRecord::writePrefix code 23 ⟩;
  ⟨ JournalRecord::writePrefixForEnd code 24 ⟩;
  ⟨ JournalRecordPair destructor code 25 ⟩;
  ⟨ endrec code 26 ⟩;
  ⟨ Journal::printHeader code 27 ⟩;
```

14.

```
⟨ SystemResources constructor code 14 ⟩ ≡
  SystemResources::SystemResources( )
  {
    gettimeofday(&start, Λ);
  }
```

This code is used in section 13.

15.

```
⟨ SystemResources::pageSize code 15 ⟩ ≡
  long int SystemResources::pageSize( )
  {
    return sysconf(_SC_PAGESIZE);
  }
```

This code is used in section 13.

16.

⟨ **SystemResources** :: *physicalPages* code 16 ⟩ ≡

```
long int SystemResources::physicalPages()
{
  return sysconf(_SC_PHYS_PAGES);
}
```

This code is used in section 13.

17.

⟨ **SystemResources** :: *onlineProcessors* code 17 ⟩ ≡

```
long int SystemResources::onlineProcessors()
{
  return sysconf(_SC_NPROCESSORS_ONLN);
}
```

This code is used in section 13.

18.

⟨ **SystemResources** :: *availableMemory* code 18 ⟩ ≡

```
long int SystemResources::availableMemory()
{
  return pageSize() * sysconf(_SC_AVPHYS_PAGES);
}
```

This code is used in section 13.

19. Here we read the current values of resource usage. For MinGW, we implement only a number of available physical memory pages.

```
⟨ SystemResources::getRUS code 19 ⟩ ≡
  void SystemResources::getRUS(double &load_avg, long int &pg_avail, double &utime, double
        &stime, double &elapsed, long int &idrss, long int &majflt)
  {
    struct timeval now;
    gettimeofday(&now, Λ);
    elapsed = now.tv_sec − start.tv_sec + (now.tv_usec − start.tv_usec) ∗ 1.0 · 10^−6;
#if ¬defined (__MINGW32__)
    struct rusage rus;
    getrusage(RUSAGE_SELF, &rus);
    utime = rus.ru_utime.tv_sec + rus.ru_utime.tv_usec ∗ 1.0 · 10^−6;
    stime = rus.ru_stime.tv_sec + rus.ru_stime.tv_usec ∗ 1.0 · 10^−6;
    idrss = rus.ru_idrss;
    majflt = rus.ru_majflt;
#else
    utime = −1.0;
    stime = −1.0;
    idrss = −1;
    majflt = −1;
#endif
#define MINGCYGTMP (¬defined (__MINGW32__) ∧ ¬defined (__CYGWIN32__) ∧ ¬defined (__CYGWIN__))
#define MINGCYG (MINGCYGTMP ∧ ¬defined (__MINGW64__) ∧ ¬defined (__CYGWIN64__))
#if MINGCYG
    getloadavg(&load_avg, 1);
#else
    load_avg = −1.0;
#endif
    pg_avail = sysconf(_SC_AVPHYS_PAGES);
  }
```

This code is used in section 13.

20.

```
⟨ SystemResourcesFlash constructor code 20 ⟩ ≡
  SystemResourcesFlash::SystemResourcesFlash( )
  {
    _sysres.getRUS(load_avg, pg_avail, utime, stime, elapsed, idrss, majflt);
  }
```

This code is used in section 13.

21.

```
⟨ SystemResourcesFlash::diff code 21 ⟩ ≡
  void SystemResourcesFlash::diff(const SystemResourcesFlash &pre)
  {
    utime -= pre.utime;
    stime -= pre.stime;
    elapsed -= pre.elapsed;
    idrss -= pre.idrss;
    majflt -= pre.majflt;
  }
```

This code is used in section 13.

22.

```
⟨ JournalRecord::operator≪ symmetry code 22 ⟩ ≡
  JournalRecord &JournalRecord::operator≪(const IntSequence &s)
  {
    operator≪("[");
    for (int i = 0; i < s.size(); i++) {
      operator≪(s[i]);
      if (i < s.size() - 1) operator≪(",");
    }
    operator≪("]");
    return *this;
  }
```

This code is used in section 13.

23.

```
⟨ JournalRecord::writePrefix code 23 ⟩ ≡
  void JournalRecord::writePrefix(const SystemResourcesFlash &f)
  {
    for (int i = 0; i < MAXLEN; i++) prefix[i] = ' ';
    double mb = 1024 * 1024;
    sprintf(prefix, "%07.6g", f.elapsed);
    sprintf(prefix + 7, ":%c%05d", recChar, ord);
    sprintf(prefix + 14, ":%1.1f", f.load_avg);
    sprintf(prefix + 18, ":%05.4g", f.pg_avail * _sysres.pageSize()/mb);
    sprintf(prefix + 24, "%s", ":     :");
    for (int i = 0; i < 2 * journal.getDepth(); i++) prefix[i + 33] = ' ';
    prefix[2 * journal.getDepth() + 33] = '\0';
  }
```

This code is used in section 13.

24.

```
⟨ JournalRecord::writePrefixForEnd code 24 ⟩ ≡
  void JournalRecordPair::writePrefixForEnd(const SystemResourcesFlash &f)
  {
    for (int i = 0; i < MAXLEN; i++) prefix_end[i] = '␣';
    double mb = 1024 * 1024;
    SystemResourcesFlash difnow;
    difnow.diff(f);
    sprintf(prefix_end, "%07.6g", f.elapsed + difnow.elapsed);
    sprintf(prefix_end + 7, ":E%05d", ord);
    sprintf(prefix_end + 14, ":%1.1f", difnow.load_avg);
    sprintf(prefix_end + 18, ":%05.4g", difnow.pg_avail * _sysres.pageSize()/mb);
    sprintf(prefix_end + 24, ":%06.5g", difnow.majflt * _sysres.pageSize()/mb);
    sprintf(prefix_end + 31, "%s", ":␣");
    for (int i = 0; i < 2 * journal.getDepth(); i++) prefix_end[i + 33] = '␣';
    prefix_end[2 * journal.getDepth() + 33] = '\0';
  }
```

This code is used in section 13.

25.

```
⟨ JournalRecordPair destructor code 25 ⟩ ≡
  JournalRecordPair::~JournalRecordPair()
  {
    journal.decrementDepth();
    writePrefixForEnd(flash);
    journal ≪ prefix_end;
    journal ≪ mes;
    journal ≪ endl;
    journal.flush();
  }
```

This code is used in section 13.

26.

```
⟨ endrec code 26 ⟩ ≡
  JournalRecord &endrec(JournalRecord &rec)
  {
    rec.journal ≪ rec.prefix;
    rec.journal ≪ rec.mes;
    rec.journal ≪ endl;
    rec.journal.flush();
    rec.journal.incrementOrd();
    return rec;
  }
```

This code is used in section 13.

27.

```
⟨ Journal::printHeader code 27 ⟩ ≡
  void Journal::printHeader()
  {
    (*this) ≪ "This␣is␣Dynare++,␣Copyright␣(C)␣2004-2011,␣Ondra␣Kamenik\n" ≪
        "Dynare++␣comes␣with␣ABSOLUTELY␣NO␣WARRANTY␣and␣is␣distributed␣under\n" ≪
        "GPL:␣modules␣integ,␣tl,␣kord,␣sylv,␣src,␣extern␣and␣documentation\n" ≪
        "LGPL:␣modules␣parser,␣utils\n" ≪ "␣for␣GPL␣␣see␣http://www.gnu.org/licens\
        es/gpl.html\n" ≪ "␣for␣LGPL␣see␣http://www.gnu.org/licenses/lgpl.html\n" ≪ "\n\n";
#if ¬defined (__MINGW32__)
    utsname info;
    uname(&info);
    (*this) ≪ "System␣info:␣";
    (*this) ≪ info.sysname ≪ "␣" ≪ info.release ≪ "␣" ≪ info.version ≪ "␣";
    (*this) ≪ info.machine ≪ ",␣processors␣online:␣" ≪ _sysres.onlineProcessors();
    (*this) ≪ "\n\nStart␣time:␣";
    char ts[100];
    time_t curtime = time(Λ);
    tm loctime;
    localtime_r(&curtime, &loctime);
    asctime_r(&loctime, ts);
    (*this) ≪ ts ≪ "\n";
#else
    (*this) ≪ "System␣info:␣(not␣implemented␣for␣MINGW)\n";
    (*this) ≪ "Start␣time:␣␣(not␣implemented␣for␣MINGW)\n\n";
#endif
    (*this) ≪ "␣␣------␣elapsed␣time␣(seconds)␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣\n";
    (*this) ≪ "␣␣|␣␣␣␣␣␣␣------␣record␣unique␣identifier␣␣␣␣␣␣␣␣␣␣␣\n";
    (*this) ≪ "␣␣|␣␣␣␣␣␣␣|␣␣␣␣␣␣------␣load␣average␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣\n";
    (*this) ≪ "␣␣|␣␣␣␣␣␣␣|␣␣␣␣␣␣|␣␣␣␣␣------␣available␣memory␣(MB)␣␣␣\n";
    (*this) ≪ "␣␣|␣␣␣␣␣␣␣|␣␣␣␣␣␣|␣␣␣␣␣|␣␣␣␣␣␣------␣␣major␣faults␣(MB)\n";
    (*this) ≪ "␣␣|␣␣␣␣␣␣␣|␣␣␣␣␣␣|␣␣␣␣␣|␣␣␣␣␣␣|␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣\n";
    (*this) ≪ "␣␣V␣␣␣␣␣␣␣V␣␣␣␣␣␣V␣␣␣␣␣V␣␣␣␣␣␣V␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣\n";
    (*this) ≪ "\n";
  }
```

This code is used in section 13.

28. Here we implement *sysconf* for MinGW. We implement only page size, number of physial pages, and a number of available physical pages. The pagesize is set to 1024 bytes, real pagesize can differ but it is not important. We can do this since Windows kernel32 *GlobalMemoryStatus* call returns number of bytes.

Number of online processors is not implemented and returns -1, since Windows kernel32 *GetSystemInfo* call is too complicated.

⟨ *sysconf* Win32 implementation 28 ⟩ ≡

```
#ifndef NOMINMAX
#define NOMINMAX      /* Do not define "min" and "max" macros */
#endif
#include <windows.h>
#define _SC_PAGESIZE  1
#define _SC_PHYS_PAGES  2
#define _SC_AVPHYS_PAGES  3
#define _SC_NPROCESSORS_ONLN  4
  long sysconf(int name)
  {
    switch (name) {
    case _SC_PAGESIZE:
      return 1024;
    case _SC_PHYS_PAGES:
      {
        MEMORYSTATUS memstat;
        GlobalMemoryStatus(&memstat);
        return memstat.dwTotalPhys/1024;
      }
    case _SC_AVPHYS_PAGES:
      {
        MEMORYSTATUS memstat;
        GlobalMemoryStatus(&memstat);
        return memstat.dwAvailPhys/1024;
      }
    case _SC_NPROCESSORS_ONLN:
      return -1;
    default:
      KORD_RAISE("Not implemented in Win32 sysconf.");
      return -1;
    }
  }
```

This code is used in section 13.

29. End of `journal.cpp` file.

30. Conjugate family for normal distribution. Start of `normal_conjugate.h` file.

The main purpose here is to implement a class representing conjugate distributions for mean and variance of the normal distribution. The class has two main methods: the first one is to update itself with respect to one observation, the second one is to update itself with respect to anothe object of the class. In the both methods, the previous state of the class corresponds to the prior distribution, and the final state corresponds to the posterior distribution.

The algrebra can be found in Gelman, Carlin, Stern, Rubin (p.87). It goes as follows: Prior conjugate distribution takes the following form:

$$\begin{aligned}\Sigma \sim &\text{InvWishart}_{\nu_0}(\Lambda_0^{-1})\\ \mu|\Sigma \sim &N(\mu_0, \Sigma/\kappa_0)\end{aligned}$$

If the observations are $y_1 \dots y_n$, then the posterior distribution has the same form with the following parameters:

$$\begin{aligned}\mu_n &= \frac{\kappa_0}{\kappa_0+n}\mu_0 + \frac{n}{\kappa_0+n}\bar{y}\\ \kappa_n &= \kappa_0 + n\\ \nu_n &= \nu_0 + n\\ \Lambda_n &= \Lambda_0 + S + \frac{\kappa_0 n}{\kappa_0+n}(\bar{y}-\mu_0)(\bar{y}-\mu_0)^T,\end{aligned}$$

where

$$\begin{aligned}\bar{y} &= \frac{1}{n}\sum_{i=1}^{n} y_i\\ S &= \sum_{i=1}^{n}(y_i-\bar{y})(y_i-\bar{y})^T\end{aligned}$$

```
#ifndef NORMAL_CONJUGATE_H
#define NORMAL_CONJUGATE_H
#include "twod_matrix.h"
  ⟨ NormalConj class declaration 31 ⟩;
#endif
```

31. The class is described by the four parameters: μ, κ, ν and Λ.

```
⟨ NormalConj class declaration 31 ⟩ ≡
  class NormalConj {
  protected:
    Vector mu;
    int kappa;
    int nu;
    TwoDMatrix lambda;
  public:
    ⟨ NormalConj constructors 32 ⟩;
    virtual ~NormalConj( ) { }
    void update(const ConstVector &y);
    void update(const ConstTwoDMatrix &ydata);
    void update(const NormalConj &nc);
    int getDim( ) const
    { return mu.length( ); }
    const Vector &getMean( ) const
    { return mu; }
    void getVariance(TwoDMatrix &v) const;
  };
```

This code is used in section 30.

32. We provide the following constructors: The first constructs diffuse (Jeffrey's) prior. It sets κ, and Λ to zeros, nu to -1 and also the mean μ to zero (it should not be referenced). The second constructs the posterior using the diffuse prior and the observed data (columnwise). The third is a copy constructor.

```
⟨ NormalConj constructors 32 ⟩ ≡
  NormalConj(int d);
  NormalConj(const ConstTwoDMatrix &ydata);
  NormalConj(const NormalConj &nc);
```

This code is used in section 31.

33. End of `normal_conjugate.h` file.

34. Start of `normal_conjugate.cpp` file.

```
#include "normal_conjugate.h"
#include "kord_exception.h"
  ⟨ NormalConj diffuse prior constructor 35 ⟩;
  ⟨ NormalConj data update constructor 36 ⟩;
  ⟨ NormalConj copy constructor 37 ⟩;
  ⟨ NormalConj::update one observation code 38 ⟩;
  ⟨ NormalConj::update multiple observations code 39 ⟩;
  ⟨ NormalConj::update with NormalConj code 40 ⟩;
  ⟨ NormalConj::getVariance code 41 ⟩;
```

35.

```
⟨ NormalConj diffuse prior constructor 35 ⟩ ≡
  NormalConj::NormalConj(int d)
  : mu(d), kappa(0), nu(-1), lambda(d,d) {
    mu.zeros();
    lambda.zeros();
  }
```

This code is used in section 34.

36.

```
⟨ NormalConj data update constructor 36 ⟩ ≡
  NormalConj::NormalConj(const ConstTwoDMatrix &ydata)
  : mu(ydata.numRows()), kappa(ydata.numCols()), nu(ydata.numCols()-1),
        lambda(ydata.numRows(),ydata.numRows()) {
    mu.zeros();
    for (int i=0; i<ydata.numCols(); i++) mu.add(1.0/ydata.numCols(),ConstVector(ydata,i));
    lambda.zeros();
    for (int i=0; i<ydata.numCols(); i++) {
      Vector diff(ConstVector(ydata,i));
      diff.add(-1,mu);
      lambda.addOuter(diff);
    }
  }
```

This code is used in section 34.

37.

```
⟨ NormalConj copy constructor 37 ⟩ ≡
  NormalConj::NormalConj(const NormalConj &nc)
  : mu(nc.mu), kappa(nc.kappa), nu(nc.nu), lambda(nc.lambda) { }
```

This code is used in section 34.

38. The method performs the following:

$$\begin{aligned}\mu_1 &= \frac{\kappa_0}{\kappa_0+1}\mu_0 + \frac{1}{\kappa_0+1}y\\ \kappa_1 &= \kappa_0 + 1\\ \nu_1 &= \nu_0 + 1\\ \Lambda_1 &= \Lambda_0 + \frac{\kappa_0}{\kappa_0+1}(y-\mu_0)(y-\mu_0)^T,\end{aligned}$$

⟨ **NormalConj** :: *update* one observation code 38 ⟩ ≡

```
void NormalConj::update(const ConstVector &y)
{
  KORD_RAISE_IF(y.length() ≠ mu.length(), "Wrong length of a vector in NormalConj::update");
  mu.mult(kappa/(1.0 + kappa));
  mu.add(1.0/(1.0 + kappa), y);
  Vector diff(y);
  diff.add(−1, mu);
  lambda.addOuter(diff, kappa/(1.0 + kappa));
  kappa++;
  nu++;
}
```

This code is used in section 34.

39. The method evaluates the formula in the header file.

⟨ **NormalConj** :: *update* multiple observations code 39 ⟩ ≡

```
void NormalConj::update(const ConstTwoDMatrix &ydata)
{
  NormalConj nc(ydata);
  update(nc);
}
```

This code is used in section 34.

40.

⟨ **NormalConj** :: *update* with **NormalConj** code 40 ⟩ ≡

```
void NormalConj::update(const NormalConj &nc)
{
  double wold = ((double) kappa)/(kappa + nc.kappa);
  double wnew = 1 − wold;
  mu.mult(wold);
  mu.add(wnew, nc.mu);
  Vector diff(nc.mu);
  diff.add(−1, mu);
  lambda.add(1.0, nc.lambda);
  lambda.addOuter(diff);
  kappa = kappa + nc.kappa;
  nu = nu + nc.kappa;
}
```

This code is used in section 34.

41. This returns $\frac{1}{\nu-d-1}\Lambda$, which is the mean of the variance in the posterior distribution. If the number of degrees of freedom is less than d, then NaNs are returned.

```
⟨ NormalConj::getVariance code 41 ⟩ ≡
  void NormalConj::getVariance(TwoDMatrix &v) const
  {
    if (nu > getDim() + 1) {
      v = (const TwoDMatrix &) lambda;
      v.mult(1.0/(nu - getDim() - 1));
    }
    else v.nans();
  }
```

This code is used in section 34.

42. End of `normal_conjugate.cpp` file.

43. Random number generation. Start of `random.h` file.

```
#ifndef RANDOM_H
#define RANDOM_H
  ⟨ RandomGenerator class declaration 44 ⟩;
  ⟨ SystemRandomGenerator class declaration 45 ⟩;
  extern SystemRandomGenerator system_random_generator;
#endif
```

44. This is a general interface to an object able to generate random numbers. Subclass needs to implement *uniform* method, other is, by default, implemented here.

```
⟨ RandomGenerator class declaration 44 ⟩ ≡
  class RandomGenerator {
  public:
    virtual double uniform() = 0;
    int int_uniform();
    double normal();
  };
```

This code is used in section 43.

45. This implements **RandomGenerator** interface with system *drand* or *rand*. It is not thread aware.

```
⟨ SystemRandomGenerator class declaration 45 ⟩ ≡
  class SystemRandomGenerator : public RandomGenerator {
  public:
    double uniform();
    void initSeed(int seed);
  };
```

This code is used in section 43.

46. End of `random.h` file.

47. Start of `random.cpp` file.

```
#include "random.h"
#include <cstdlib>
#include <limits>
#include <cmath>
  ⟨ RandomGenerator::int_uniform code 48 ⟩;
  ⟨ RandomGenerator::normal code 49 ⟩;
  SystemRandomGenerator system_random_generator;
  ⟨ SystemRandomGenerator::uniform code 50 ⟩;
  ⟨ SystemRandomGenerator::initSeed code 51 ⟩;
```

48.

```
⟨ RandomGenerator::int_uniform code 48 ⟩ ≡
  int RandomGenerator::int_uniform(){ double s = std::numeric_limits < int > ::max()*uniform();
      return (int) s; }
```

This code is used in section 47.

49. This implements Marsaglia Polar Method.

```
⟨ RandomGenerator::normal code 49 ⟩ ≡
  double RandomGenerator::normal()
  {
    double x1, x2;
    double w;
    do {
      x1 = 2 * uniform() − 1;
      x2 = 2 * uniform() − 1;
      w = x1 * x1 + x2 * x2;
    } while (w ≥ 1.0 ∨ w < 1.0 · 10^−30);
    return x1 * std::sqrt((−2.0 * std::log(w))/w);
  }
```

This code is used in section 47.

50.

```
⟨ SystemRandomGenerator::uniform code 50 ⟩ ≡
  double SystemRandomGenerator::uniform()
  {
#if ¬defined (__MINGW32__)
    return drand48();
#else
    return ((double) rand())/RAND_MAX;
#endif
  }
```

This code is used in section 47.

51.

```
⟨ SystemRandomGenerator::initSeed code 51 ⟩ ≡
  void SystemRandomGenerator::initSeed(int seed)
  {
#if ¬defined (__MINGW32__)
    srand48(seed);
#else
    srand(seed);
#endif
  }
```

This code is used in section 47.

52. End of `random.cpp` file.

53. Mersenne Twister PRNG. Start of `mersenne_twister.h` file.

This file provides a class for generating random numbers with encapsulated state. It is based on the work of Makoto Matsumoto and Takuji Nishimura, implementation inspired by code of Richard Wagner and Geoff Kuenning.

```
#ifndef MERSENNE_TWISTER_H
#define MERSENNE_TWISTER_H
#include "random.h"
#include <cstring>
  ⟨ MersenneTwister class declaration 54 ⟩;
  ⟨ MersenneTwister inline method definitions 56 ⟩;
#endif
```

54.

```
⟨ MersenneTwister class declaration 54 ⟩ ≡
  class MersenneTwister : public RandomGenerator {
  protected:
    typedef unsigned int uint32;
    enum {
      STATE_SIZE = 624
    };
    enum {
      RECUR_OFFSET = 397
    };
    uint32 statevec[STATE_SIZE];
    int stateptr;
  public:
    MersenneTwister(uint32 iseed);
    MersenneTwister(const MersenneTwister &mt);
    virtual ~MersenneTwister( )
    { }
    uint32 lrand( );
    double drand( );
    double uniform( )
    { return drand( ); }
  protected:
    void seed(uint32 iseed);
    void refresh( );
  private:
    ⟨ MersenneTwister static inline methods 55 ⟩;
  };
```

This code is used in section 53.

55.

⟨ **MersenneTwister** static inline methods 55 ⟩ ≡
```
static uint32 hibit(uint32 u)
{
  return u & #80000000UL;
}
static uint32 lobit(uint32 u)
{
  return u & #00000001UL;
}
static uint32 lobits(uint32 u)
{
  return u & #7fffffffUL;
}
static uint32 mixbits(uint32 u, uint32 v)
{
  return hibit(u) | lobits(v);
}
static uint32 twist(uint32 m, uint32 s0, uint32 s1)
{
  return m ⊕ (mixbits(s0, s1) ≫ 1) ⊕ (−lobit(s1) & #9908b0dfUL);
}
```
This code is used in section 54.

56.

⟨ **MersenneTwister** inline method definitions 56 ⟩ ≡
```
⟨ MersenneTwister constructor code 57 ⟩;
⟨ MersenneTwister copy constructor code 58 ⟩;
⟨ MersenneTwister::lrand code 59 ⟩;
⟨ MersenneTwister::drand code 60 ⟩;
⟨ MersenneTwister::seed code 61 ⟩;
⟨ MersenneTwister::refresh code 62 ⟩;
```
This code is used in section 53.

57.

⟨ **MersenneTwister** constructor code 57 ⟩ ≡
```
inline MersenneTwister::MersenneTwister(uint32 iseed)
{
  seed(iseed);
}
```
This code is used in section 56.

58.

⟨ **MersenneTwister** copy constructor code 58 ⟩ ≡
```
inline MersenneTwister::MersenneTwister(const MersenneTwister &mt)
: stateptr(mt.stateptr) {
  memcpy(statevec, mt.statevec, sizeof(uint32) * STATE_SIZE);
}
```
This code is used in section 56.

59.

⟨ **MersenneTwister** :: *lrand* code 59 ⟩ ≡
```
inline MersenneTwister::uint32 MersenneTwister::lrand()
{
  if (stateptr ≥ STATE_SIZE) refresh();
  register uint32 v = statevec[stateptr++];
  v ⊕= v ≫ 11;
  v ⊕= (v ≪ 7) & #9d2c5680;
  v ⊕= (v ≪ 15) & #efc60000;
  return (v ⊕ (v ≫ 18));
}
```
This code is used in section 56.

60.

⟨ **MersenneTwister** :: *drand* code 60 ⟩ ≡
```
inline double MersenneTwister::drand()
{
  uint32 a = lrand() ≫ 5;
  uint32 b = lrand() ≫ 6;
  return (a * 67108864.0 + b) * (1.0/9007199254740992.0);
}
```
This code is used in section 56.

61. PRNG of D. Knuth

⟨ **MersenneTwister** :: *seed* code 61 ⟩ ≡
```
inline void MersenneTwister::seed(uint32 iseed)
{
  statevec[0] = iseed & #ffffffff_UL;
  for (int i = 1; i < STATE_SIZE; i++) {
    register uint32 val = statevec[i − 1] ≫ 30;
    val ⊕= statevec[i − 1];
    val *= 1812433253_UL;
    val += i;
    statevec[i] = val & #ffffffff_UL;
  }
  refresh();
}
```
This code is used in section 56.

62.

⟨ **MersenneTwister** :: *refresh* code 62 ⟩ ≡
```
inline void MersenneTwister::refresh()
{
  register uint32 *p = statevec;
  for (int i = STATE_SIZE − RECUR_OFFSET; i−−; ++p) *p = twist(p[RECUR_OFFSET], p[0], p[1]);
  for (int i = RECUR_OFFSET; −−i; ++p) *p = twist(p[RECUR_OFFSET − STATE_SIZE], p[0], p[1]);
  *p = twist(p[RECUR_OFFSET − STATE_SIZE], p[0], statevec[0]);
  stateptr = 0;
}
```
This code is used in section 56.

63. End of `mersenne_twister.h` file.

64. Faa Di Bruno evaluator. Start of `faa_di_bruno.h` file.
This defines a class which implements Faa Di Bruno Formula

$$[B_{s^k}]_{\alpha_1\dots\alpha_l} = [f_{z^l}]_{\beta_1\dots\beta_l} \sum_{c\in M_{l,k}} \prod_{m=1}^{l} \left[z_{s^k(c_m)}\right]_{c_m(\alpha)}^{\beta_m}$$

where s^k is a general symmetry of dimension k and z is a stack of functions.

```
#ifndef FAA_DI_BRUNO_H
#define FAA_DI_BRUNO_H
#include "journal.h"
#include "stack_container.h"
#include "t_container.h"
#include "sparse_tensor.h"
#include "gs_tensor.h"
  ⟨ FaaDiBruno class declaration 65 ⟩;
#endif
```

65. Nothing special here. See ⟨ **FaaDiBruno**::*calculate* folded sparse code 68 ⟩ for reason of having *magic_mult*.

```
⟨ FaaDiBruno class declaration 65 ⟩ ≡
  class FaaDiBruno {
    Journal &journal;
  public:
    FaaDiBruno(Journal &jr)
    : journal(jr) { }
    void calculate(const StackContainer⟨FGSTensor⟩ &cont,
        const TensorContainer⟨FSSparseTensor⟩ &f, FGSTensor &out);
    void calculate(const FoldedStackContainer &cont, const FGSContainer &g, FGSTensor
        &out);
    void calculate(const StackContainer⟨UGSTensor⟩ &cont,
        const TensorContainer⟨FSSparseTensor⟩ &f, UGSTensor &out);
    void calculate(const UnfoldedStackContainer &cont, const UGSContainer &g, UGSTensor
        &out);
  protected:
    int estimRefinment(const TensorDimens &tdims, int nr, int l, int &avmem_mb, int &tmpmem_mb);
    static double magic_mult;
  };
```

This code is used in section 64.

66. End of `faa_di_bruno.h` file.

67. Start of `faa_di_bruno.cpp` file.

```
#include "faa_di_bruno.h"
#include "fine_container.h"
#include <cmath>
  double FaaDiBruno::magic_mult = 1.5;
  ⟨ FaaDiBruno::calculate folded sparse code 68 ⟩;
  ⟨ FaaDiBruno::calculate folded dense code 69 ⟩;
  ⟨ FaaDiBruno::calculate unfolded sparse code 70 ⟩;
  ⟨ FaaDiBruno::calculate unfolded dense code 71 ⟩;
  ⟨ FaaDiBruno::estimRefinment code 72 ⟩;
```

68. We take an opportunity to refine the stack container to avoid allocation of more memory than available.

```
⟨ FaaDiBruno::calculate folded sparse code 68 ⟩ ≡
  void FaaDiBruno::calculate(const StackContainer⟨FGSTensor⟩ &cont, const
          TensorContainer⟨FSSparseTensor⟩ &f, FGSTensor &out)
  {
    out.zeros();
    for (int l = 1; l ≤ out.dimen(); l++) {
      int mem_mb, p_size_mb;
      int max = estimRefinment(out.getDims(), out.nrows(), l, mem_mb, p_size_mb);
      FoldedFineContainer fine_cont(cont, max);
      fine_cont.multAndAdd(l, f, out);
      JournalRecord recc(journal);
      recc ≪ "dim=" ≪ l ≪ "␣avmem=" ≪ mem_mb ≪ "␣tmpmem=" ≪ p_size_mb ≪ "␣max=" ≪ max ≪
          "␣stacks=" ≪ cont.numStacks() ≪ "->" ≪ fine_cont.numStacks() ≪ endrec;
    }
  }
```

This code is cited in sections 65 and 70.

This code is used in section 67.

69. Here we just simply evaluate *multAndAdd* for the dense container. There is no opportunity for tuning.

```
⟨ FaaDiBruno::calculate folded dense code 69 ⟩ ≡
  void FaaDiBruno::calculate(const FoldedStackContainer &cont, const FGSContainer
          &g, FGSTensor &out)
  {
    out.zeros();
    for (int l = 1; l ≤ out.dimen(); l++) {
      long int mem = SystemResources::availableMemory();
      cont.multAndAdd(l, g, out);
      JournalRecord rec(journal);
      int mem_mb = mem/1024/1024;
      rec ≪ "dim=" ≪ l ≪ "␣avmem=" ≪ mem_mb ≪ endrec;
    }
  }
```

This code is used in section 67.

70. This is the same as ⟨ **FaaDiBruno** :: *calculate* folded sparse code 68 ⟩. The only difference is that we construct unfolded fine container.

```
⟨ FaaDiBruno :: calculate unfolded sparse code 70 ⟩ ≡
  void FaaDiBruno::calculate(const StackContainer⟨UGSTensor⟩ &cont, const
          TensorContainer⟨FSSparseTensor⟩ &f, UGSTensor &out)
  {
    out.zeros();
    for (int l = 1; l ≤ out.dimen(); l++) {
      int mem_mb, p_size_mb;
      int max = estimRefinment(out.getDims(), out.nrows(), l, mem_mb, p_size_mb);
      UnfoldedFineContainer fine_cont(cont, max);
      fine_cont.multAndAdd(l, f, out);
      JournalRecord recc(journal);
      recc ≪ "dim=" ≪ l ≪ "␣avmem=" ≪ mem_mb ≪ "␣tmpmem=" ≪ p_size_mb ≪ "␣max=" ≪ max ≪
          "␣stacks=" ≪ cont.numStacks() ≪ "->" ≪ fine_cont.numStacks() ≪ endrec;
    }
  }
```

This code is used in section 67.

71. Again, no tuning opportunity here.

```
⟨ FaaDiBruno :: calculate unfolded dense code 71 ⟩ ≡
  void FaaDiBruno::calculate(const UnfoldedStackContainer &cont, const UGSContainer
          &g, UGSTensor &out)
  {
    out.zeros();
    for (int l = 1; l ≤ out.dimen(); l++) {
      long int mem = SystemResources::availableMemory();
      cont.multAndAdd(l, g, out);
      JournalRecord rec(journal);
      int mem_mb = mem/1024/1024;
      rec ≪ "dim=" ≪ l ≪ "␣avmem=" ≪ mem_mb ≪ endrec;
    }
  }
```

This code is used in section 67.

72. This function returns a number of maximum rows used for refinement of the stacked container. We want to set the maximum so that the expected memory consumption for the number of paralel threads would be less than available memory. On the other hand we do not want to be too pesimistic since a very fine refinement can be very slow.

Besides memory needed for a dense unfolded slice of a tensor from f, each thread needs $magic\_mult * per\_size$ bytes of memory. In the worst case, *magic_mult* will be equal to 2, this means memory *per_size* for target temporary (permuted symmetry) tensor plus one copy for intermediate result. However, this shows to be too pesimistic, so we set *magic_mult* to 1.5. The memory for permuted symmetry temporary tensor *per_size* is estimated as a weigthed average of unfolded memory of the *out* tensor and unfolded memory of a symetric tensor with the largest coordinate size. Some experiments showed that the best combination of the two is to take 100% if the latter, so we set *lambda* to zero.

The *max* number of rows in the refined *cont* must be such that each slice fits to remaining memory. Number of columns of the slice are never greater max^l. (This is not true, since stacks corresponing to unit/zero matrices cannot be further refined). We get en equation:

$$nthreads \cdot max^l \cdot 8 \cdot r = mem - magic\_mult \cdot nthreads \cdot per\_size \cdot 8 \cdot r,$$

where *mem* is available memory in bytes, *nthreads* is a number of threads, r is a number of rows, and 8 is **sizeof**(**double**).

If the right hand side is less than zero, we set *max* to 10, just to let it do something.

⟨ **FaaDiBruno**::*estimRefinment* code 72 ⟩ ≡

```
int FaaDiBruno::estimRefinment(const TensorDimens &tdims, int nr, int l, int &avmem_mb, int
        &tmpmem_mb)
{
  int nthreads = THREAD_GROUP::max_parallel_threads;
  long int per_size1 = tdims.calcUnfoldMaxOffset();
  long int per_size2 = (long int) pow((double) tdims.getNVS().getMax(), l);
  double lambda = 0.0;
  long int per_size = sizeof(double) * nr * (long int)(lambda * per_size1 + (1 - lambda) * per_size2);
  long int mem = SystemResources::availableMemory();
  int max = 0;
  double num_cols = ((double)(mem - magic_mult * nthreads * per_size))/nthreads/sizeof(double)/nr;
  if (num_cols > 0) {
    double maxd = pow(num_cols, ((double) 1)/l);
    max = (int) floor(maxd);
  }
  if (max == 0) {
    max = 10;
    JournalRecord rec(journal);
    rec << "dim=" << l << " run out of memory, imposing max=" << max;
    if (nthreads > 1) rec << " (decrease number of threads)";
    rec << endrec;
  }
  avmem_mb = mem/1024/1024;
  tmpmem_mb = (nthreads * per_size)/1024/1024;
  return max;
}
```

This code is used in section 67.

73. End of `faa_di_bruno.cpp` file.

74. Retrieving derivatives.

75. First order at deterministic steady. Start of `first_order.h` file.

```
#ifndef FIRST_ORDER_H
#define FIRST_ORDER_H
#include "korder.h"
  ⟨ FirstOrder class declaration 76 ⟩;
  ⟨ FirstOrderDerivs class declaration 77 ⟩;
#endif
```

76.

```
⟨ FirstOrder class declaration 76 ⟩ ≡
  template⟨int⟩ class FirstOrderDerivs;
  class FirstOrder {
    template⟨int⟩ friend class FirstOrderDerivs;
    PartitionY ypart;
    int nu;
    TwoDMatrix gy;
    TwoDMatrix gu;
    bool bk_cond;
    double b_error;
    int sdim;
    Vector alphar;
    Vector alphai;
    Vector beta;
    double qz_criterium;
    Journal &journal;
  public:
    FirstOrder(int num_stat, int num_pred, int num_both, int num_forw, int num_u, const
          FSSparseTensor &f, Journal &jr, double qz_crit)
    : ypart(num_stat, num_pred, num_both, num_forw), nu(num_u), gy(ypart.ny(), ypart.nys()),
          gu(ypart.ny(), nu), alphar(ypart.ny() + ypart.nboth), alphai(ypart.ny() + ypart.nboth),
          beta(ypart.ny() + ypart.nboth), qz_criterium(qz_crit), journal(jr) { solve(FFSTensor(f)); }
    bool isStable() const
    { return bk_cond; }
    const TwoDMatrix &getGy() const
    { return gy; }
    const TwoDMatrix &getGu() const
    { return gu; }
  protected:
    void solve(const TwoDMatrix &f);
    void journalEigs();
  };
```

This code is cited in section 182.

This code is used in section 75.

77. This class only converts the derivatives g_{y^*} and g_u to a folded or unfolded container.

```
⟨ FirstOrderDerivs class declaration 77 ⟩ ≡
  template⟨int t⟩ class FirstOrderDerivs : public ctraits⟨t⟩::Tg
  {
  public:
    FirstOrderDerivs(const FirstOrder &fo)
    : ctraits⟨t⟩::Tg(4) {
      IntSequence nvs(4);
      nvs[0] = fo.ypart.nys();
      nvs[1] = fo.nu;
      nvs[2] = fo.nu;
      nvs[3] = 1;
      _Ttensor *ten = new _Ttensor(fo.ypart.ny(), TensorDimens(Symmetry(1,0,0,0), nvs));
      ten→zeros();
      ten→add(1.0, fo.gy);
      this→insert(ten);
      ten = new _Ttensor(fo.ypart.ny(), TensorDimens(Symmetry(0,1,0,0), nvs));
      ten→zeros();
      ten→add(1.0, fo.gu);
      this→insert(ten);
    }
  }
  ;
```

This code is used in section 75.

78. End of `first_order.h` file.

79. Start of `first_order.cpp` file.

```
#include "kord_exception.h"
#include "first_order.h"
#include <dynlapack.h>
  double qz_criterium = 1.000001;
  ⟨ order_eigs function code 80 ⟩;
  ⟨ FirstOrder::solve code 81 ⟩;
  ⟨ FirstOrder::journalEigs code 93 ⟩;
```

80. This is a function which selects the eigenvalues pair used by *dgges*. See documentation to DGGES for details. Here we want to select (return true) the pairs for which $\alpha < \beta$.

```
⟨ order_eigs function code 80 ⟩ ≡
  lapack_int order_eigs(const double *alphar, const double *alphai, const double *beta)
  {
    return (*alphar * *alphar + *alphai * *alphai < *beta * *beta * qz_criterium);
  }
```

This code is used in section 79.

81. Here we solve the linear approximation. The result are the matrices g_{y^*} and g_u. The method solves the first derivatives of g so that the following equation would be true:

$$E_t[F(y^*_{t-1}, u_t, u_{t+1}, \sigma)] = E_t[f(g^{**}(g^*(y^*_{t-1}, u_t, \sigma), u_{t+1}, \sigma), g(y^*_{t-1}, u_t, \sigma), y^*_{t-1}, u_t)] = 0$$

where f is a given system of equations.

It is known that g_{y^*} is given by $F_{y^*} = 0$, g_u is given by $F_u = 0$, and g_σ is zero. The only input to the method are the derivatives fd of the system f, and partitioning of the vector y (from object).

```
⟨ FirstOrder::solve code 81 ⟩ ≡
  void FirstOrder::solve(const TwoDMatrix &fd)
  {
    JournalRecordPair pa(journal);
    pa ≪ "Recovering␣first␣order␣derivatives␣" ≪ endrec;
    ::qz_criterium = FirstOrder::qz_criterium;
    ⟨ solve derivatives gy 82 ⟩;
    ⟨ solve derivatives gu 92 ⟩;
    journalEigs();
    if (¬gy.isFinite() ∨ ¬gu.isFinite()) {
      throw KordException(__FILE__, __LINE__,
          "NaN␣or␣Inf␣asserted␣in␣first␣order␣derivatives␣in␣FirstOrder::solve");
    }
  }
```

This code is used in section 79.

82. The derivatives g_{y^*} are retrieved from the equation $F_{y^*} = 0$. The calculation proceeds as follows:

1) For each variable appearing at both $t-1$ and $t-1$ we add a dummy variable, so that the predetermined variables and forward looking would be disjoint. This is, the matrix of the first derivatives of the system written as:
$$\begin{bmatrix} f_{y_+^{**}} & f_{ys} & f_{yp} & f_{yb} & f_{yf} & f_{y_-^*} \end{bmatrix},$$
where f_{ys}, f_{yp}, f_{yb}, and f_{yf} are derivatives wrt static, predetermined, both, forward looking at time t, is rewritten to the matrix:
$$\begin{bmatrix} f_{y_+^{**}} & f_{ys} & f_{yp} & f_{yb} & 0 & f_{yf} & f_{y_-^*} \\ 0 & 0 & 0 & I & -I & 0 & 0 \end{bmatrix},$$
where the second line has number of rows equal to the number of both variables.
2) Next, provided that forward looking and predetermined are disjoint, the equation $F_{y^*} = 0$ is written as:
$$\left[f_+ y_+^{**}\right]\left[g_{y^*}^{**}\right]\left[g_{y^*}^{*}\right] + \left[f_{ys}\right]\left[g_{y^*}^{s}\right] + \left[f_{y^*}\right]\left[g_{y^*}^{*}\right] + \left[f_{y^{**}}\right]\left[g_{y^*}^{**}\right] + \left[f_{y_-^*}\right] = 0$$
This is rewritten as
$$\begin{bmatrix} f_{y^*} & 0 & f_{y_+^{**}} \end{bmatrix}\begin{bmatrix} I \\ g_{y^*}^{s} \\ g_{y^*}^{**} \end{bmatrix}\left[g_{y^*}^{*}\right] + \begin{bmatrix} f_{y_-^*} & f_{ys} & f_{y^{**}} \end{bmatrix}\begin{bmatrix} I \\ g_{y^*}^{s} \\ g_{y^*}^{**} \end{bmatrix} = 0$$
Now, in the above equation, there are the auxiliary variables standing for copies of both variables at time $t+1$. This equation is then rewritten as:
$$\begin{bmatrix} f_{yp} & f_{yb} & 0 & f_{y_+^{**}} \\ 0 & I & 0 & 0 \end{bmatrix}\begin{bmatrix} I \\ g_{y^*}^{s} \\ g_{y^*}^{**} \end{bmatrix}\left[g_{y^*}^{*}\right] + \begin{bmatrix} f_{y_-^*} & f_{ys} & 0 & f_{yf} \\ 0 & 0 & -I & 0 \end{bmatrix}\begin{bmatrix} I \\ g_{y^*}^{s} \\ g_{y^*}^{**} \end{bmatrix} = 0$$
The two matrices are denoted as D and $-E$, so the equation takes the form:
$$D\begin{bmatrix} I \\ g_{y^*}^{s} \\ g_{y^*}^{**} \end{bmatrix}\left[g_{y^*}^{*}\right] = E\begin{bmatrix} I \\ g_{y^*}^{s} \\ g_{y^*}^{**} \end{bmatrix}$$
3) Next we solve the equation by Generalized Schur decomposition:
$$\begin{bmatrix} T_{11} & T_{12} \\ 0 & T_{22} \end{bmatrix}\begin{bmatrix} Z_{11}^T & Z_{21}^T \\ Z_{12}^T & Z_{22}^T \end{bmatrix}\begin{bmatrix} I \\ X \end{bmatrix}\left[g_{y^*}^{*}\right] = \begin{bmatrix} S_{11} & S_{12} \\ 0 & S_{22} \end{bmatrix}\begin{bmatrix} Z_{11}^T & Z_{21}^T \\ Z_{12}^T & Z_{22}^T \end{bmatrix}\begin{bmatrix} I \\ X \end{bmatrix}$$
We reorder the eigenvalue pair so that S_{ii}/T_{ii} with modulus less than one would be in the left-upper part.
4) The Blanchard–Kahn stability argument implies that the pairs with modulus less that one will be in and only in S_{11}/T_{11}. The exploding paths will be then eliminated when
$$\begin{bmatrix} Z_{11}^T & Z_{21}^T \\ Z_{12}^T & Z_{22}^T \end{bmatrix}\begin{bmatrix} I \\ X \end{bmatrix} = \begin{bmatrix} Y \\ 0 \end{bmatrix}$$
From this we have, $Y = Z_{11}^{-1}$, and $X = Z_{21}Y$, or equivalently $X = -Z_{22}^{-T}Z_{12}^T$. From the equation, we get $\left[g_{y^*}^{*}\right] = Y^{-1}T_{11}^{-1}S_{11}Y$, which is $Z_{11}T_{11}^{-1}S_{11}Z_{11}^{-1}$.
5) We then copy the derivatives to storage gy. Note that the derivatives of both variables are in X and in $\left[g_{y^*}^{*}\right]$, so we check whether the two submatrices are the same. The difference is only numerical error.

⟨ solve derivatives gy 82 ⟩ ≡
⟨ setup submatrices of f 83 ⟩;
⟨ form matrix D 84 ⟩;
⟨ form matrix E 85 ⟩;
⟨ solve generalized Schur 86 ⟩;
⟨ make submatrices of right space 87 ⟩;
⟨ calculate derivatives of static and forward 88 ⟩;
⟨ calculate derivatives of predetermined 89 ⟩;
⟨ copy derivatives to gy 90 ⟩;
⟨ check difference for derivatives of both 91 ⟩;
This code is used in section 81.

83. Here we setup submatrices of the derivatives fd.

⟨ setup submatrices of f 83 ⟩ ≡
int $off = 0$;
ConstTwoDMatrix $fyplus(fd, off, ypart.nyss())$;
$off \mathrel{+}= ypart.nyss()$;
ConstTwoDMatrix $fyszero(fd, off, ypart.nstat)$;
$off \mathrel{+}= ypart.nstat$;
ConstTwoDMatrix $fypzero(fd, off, ypart.npred)$;
$off \mathrel{+}= ypart.npred$;
ConstTwoDMatrix $fybzero(fd, off, ypart.nboth)$;
$off \mathrel{+}= ypart.nboth$;
ConstTwoDMatrix $fyfzero(fd, off, ypart.nforw)$;
$off \mathrel{+}= ypart.nforw$;
ConstTwoDMatrix $fymins(fd, off, ypart.nys())$;
$off \mathrel{+}= ypart.nys()$;
ConstTwoDMatrix $fuzero(fd, off, nu)$;
$off \mathrel{+}= nu$;
This code is used in section 82.

84.

⟨ form matrix D 84 ⟩ ≡
$lapack\_int n = ypart.ny() + ypart.nboth$;
TwoDMatrix $matD(n, n)$;
$matD.zeros()$;
$matD.place(fypzero, 0, 0)$;
$matD.place(fybzero, 0, ypart.npred)$;
$matD.place(fyplus, 0, ypart.nys() + ypart.nstat)$;
for (**int** $i = 0$; $i < ypart.nboth$; $i{+}{+}$) $matD.get(ypart.ny() + i, ypart.npred + i) = 1.0$;
This code is used in section 82.

85.

⟨ form matrix E 85 ⟩ ≡
```
TwoDMatrix matE(n, n);
matE.zeros();
matE.place(fymins, 0, 0);
matE.place(fyszero, 0, ypart.nys());
matE.place(fyfzero, 0, ypart.nys() + ypart.nstat + ypart.nboth);
for (int i = 0; i < ypart.nboth; i++) matE.get(ypart.ny() + i, ypart.nys() + ypart.nstat + i) = -1.0;
matE.mult(-1.0);
```
This code is used in section 82.

86.

⟨ solve generalized Schur 86 ⟩ ≡
```
TwoDMatrix vsl(n, n);
TwoDMatrix vsr(n, n);
lapack_int lwork = 100 * n + 16;
Vector work(lwork);
lapack_int * bwork = new lapack_int[n];
lapack_int info;
lapack_int sdim2 = sdim;
dgges("N", "V", "S", order_eigs, &n, matE.getData().base(), &n, matD.getData().base(), &n, &sdim2,
      alphar.base(), alphai.base(), beta.base(), vsl.getData().base(), &n, vsr.getData().base(), &n,
      work.base(), &lwork, bwork, &info);
if (info) {
  throw KordException(__FILE__, __LINE__, "DGGES returns an error in FirstOrder::solve");
}
sdim = sdim2;
bk_cond = (sdim ≡ ypart.nys());
delete[] bwork;
```
This code is used in section 82.

87. Here we setup submatrices of the matrix Z.

⟨ make submatrices of right space 87 ⟩ ≡
```
ConstGeneralMatrix z11(vsr, 0, 0, ypart.nys(), ypart.nys());
ConstGeneralMatrix z12(vsr, 0, ypart.nys(), ypart.nys(), n - ypart.nys());
ConstGeneralMatrix z21(vsr, ypart.nys(), 0, n - ypart.nys(), ypart.nys());
ConstGeneralMatrix z22(vsr, ypart.nys(), ypart.nys(), n - ypart.nys(), n - ypart.nys());
```
This code is used in section 82.

88. Here we calculate $X = -Z_{22}^{-T} Z_{12}^T$, where X is *sfder* in the code.

⟨ calculate derivatives of static and forward 88 ⟩ ≡
```
GeneralMatrix sfder(z12, "transpose");
z22.multInvLeftTrans(sfder);
sfder.mult(-1);
```
This code is used in section 82.

89. Here we calculate $g^*_{y^*} = Z_{11}T_{11}^{-1}S_{11}Z_{11}^{-1} = Z_{11}T_{11}^{-1}(Z_{11}^{-T}S_{11}^T)^T$.

⟨ calculate derivatives of predetermined 89 ⟩ ≡
```
ConstGeneralMatrix s11(matE,0,0,ypart.nys(),ypart.nys());
ConstGeneralMatrix t11(matD,0,0,ypart.nys(),ypart.nys());
GeneralMatrix dumm(s11,"transpose");
z11.multInvLeftTrans(dumm);
GeneralMatrix preder(dumm,"transpose");
t11.multInvLeft(preder);
preder.multLeft(z11);
```
This code is used in section 82.

90.

⟨ copy derivatives to gy 90 ⟩ ≡
```
gy.place(preder,ypart.nstat,0);
GeneralMatrix sder(sfder,0,0,ypart.nstat,ypart.nys());
gy.place(sder,0,0);
GeneralMatrix fder(sfder,ypart.nstat+ypart.nboth,0,ypart.nforw,ypart.nys());
gy.place(fder,ypart.nstat+ypart.nys(),0);
```
This code is used in section 82.

91.

⟨ check difference for derivatives of both 91 ⟩ ≡
```
GeneralMatrix bder((const GeneralMatrix &) sfder,ypart.nstat,0,ypart.nboth,ypart.nys());
GeneralMatrix bder2(preder,ypart.npred,0,ypart.nboth,ypart.nys());
bder.add(-1,bder2);
b_error = bder.getData().getMax();
```
This code is used in section 82.

92. The equation $F_u = 0$ can be written as

$$\left[f_{y^{**}_+}\right]\left[g^{**}_{y^*}\right]\left[g^*_u\right] + \left[f_y\right]\left[g_u\right] + \left[f_u\right] = 0$$

and rewritten as

$$\left[f_y + \left[\begin{array}{ccc} 0 & f_{y^{**}_+}g^{**}_{y^*} & 0 \end{array}\right]\right] g_u = f_u$$

This is exactly done here. The matrix $\left[f_y + \left[\begin{array}{ccc} 0 & f_{y^{**}_+}g^{**}_{y^*} & 0 \end{array}\right]\right]$ is $matA$ in the code.

⟨ solve derivatives gu 92 ⟩ ≡
```
GeneralMatrix matA(ypart.ny(),ypart.ny());
matA.zeros();
ConstGeneralMatrix gss(gy,ypart.nstat+ypart.npred,0,ypart.nyss(),ypart.nys());
GeneralMatrix aux(fyplus,gss);
matA.place(aux,0,ypart.nstat);
ConstGeneralMatrix fyzero(fd,0,ypart.nyss(),ypart.ny(),ypart.ny());
matA.add(1.0,fyzero);
gu.zeros();
gu.add(-1.0,fuzero);
ConstGeneralMatrix(matA).multInvLeft(gu);
```
This code is used in section 81.

93.

```
⟨ FirstOrder::journalEigs code 93 ⟩ ≡
  void FirstOrder::journalEigs()
  {
    if (bk_cond) {
      JournalRecord jr(journal);
      jr ≪ "Blanchard-Kahn␣conditition␣satisfied,␣model␣stable" ≪ endrec;
    }
    else {
      JournalRecord jr(journal);
      jr ≪ "Blanchard-Kahn␣condition␣not␣satisfied,␣model␣not␣stable:␣sdim=" ≪ sdim ≪
          "␣" ≪ "npred=" ≪ ypart.nys() ≪ endrec;
    }
    if (¬bk_cond) {
      for (int i = 0; i < alphar.length(); i++) {
        if (i ≡ sdim ∨ i ≡ ypart.nys()) {
          JournalRecord jr(journal);
          jr ≪ "------------------------------------------------------␣";
          if (i ≡ sdim) jr ≪ "sdim";
          else jr ≪ "npred";
          jr ≪ endrec;
        }
        JournalRecord jr(journal);
        double mod = sqrt(alphar[i] * alphar[i] + alphai[i] * alphai[i]);
        mod = mod/round(100000 * std::abs(beta[i])) * 100000;
        jr ≪ i ≪ "\t(" ≪ alphar[i] ≪ "," ≪ alphai[i] ≪ ")␣/␣" ≪ beta[i] ≪ "␣␣\t" ≪ mod ≪ endrec;
      }
    }
  }
```

This code is used in section 79.

94. End of `first_order.cpp` file.

95. Higher order at deterministic steady. Start of `korder.h` file.

The main purpose of this file is to implement a perturbation method algorithm for an SDGE model for higher order approximations. The input of the algorithm are sparse tensors as derivatives of the dynamic system, then dimensions of vector variables, then the first order approximation to the decision rule and finally a covariance matrix of exogenous shocks. The output are higher order derivatives of decision rule $y_t = g(y^*_{t-1}, u_t, \sigma)$. The class provides also a method for checking a size of residuals of the solved equations.

The algorithm is implemented in **KOrder** class. The class contains both unfolded and folded containers to allow for switching (usually from unfold to fold) during the calculations. The algorithm is implemented in a few templated methods. To do this, we need some container type traits, which are in **ctraits** struct. Also, the **KOrder** class contains some information encapsulated in other classes, which are defined here. These include: **PartitionY**, **MatrixA**, **MatrixS** and **MatrixB**.

```
#ifndef KORDER_H
#define KORDER_H
#include "int_sequence.h"
#include "fs_tensor.h"
#include "gs_tensor.h"
#include "t_container.h"
#include "stack_container.h"
#include "normal_moments.h"
#include "t_polynomial.h"
#include "faa_di_bruno.h"
#include "journal.h"
#include "kord_exception.h"
#include "GeneralSylvester.h"
#include <dynlapack.h>
#include <cmath>
#define TYPENAME typename
  ⟨ ctraits type traits declaration 96 ⟩;
  ⟨ PartitionY struct declaration 97 ⟩;
  ⟨ PLUMatrix class declaration 98 ⟩;
  ⟨ MatrixA class declaration 99 ⟩;
  ⟨ MatrixS class declaration 100 ⟩;
  ⟨ MatrixB class declaration 101 ⟩;
  ⟨ KOrder class declaration 102 ⟩;
#endif
```

96. Here we use a classical IF template, and in **ctraits** we define a number of types. We have a type for tensor *Ttensor*, and types for each pair of folded/unfolded containers used as a member in **KOrder**.

Note that we have enumeration *fold* and *unfold*. These must have the same value as the same enumeration in **KOrder**.

```
⟨ ctraits type traits declaration 96 ⟩ ≡
  class FoldedZXContainer;
  class UnfoldedZXContainer;
  class FoldedGXContainer;
  class UnfoldedGXContainer;
  template⟨bool condition, class Then, class Else⟩ struct IF {
    typedef Then RET;
  };
  template⟨class Then, class Else⟩ struct IF⟨false, Then, Else⟩ {
    typedef Else RET;
  };
  template⟨int type⟩ class ctraits {
  public:
    enum { fold, unfold };
    typedef TYPENAME IF⟨type ≡ fold, FGSTensor, UGSTensor⟩::RET Ttensor;
    typedef TYPENAME IF⟨type ≡ fold, FFSTensor, UFSTensor⟩::RET Ttensym;
    typedef TYPENAME IF⟨type ≡ fold, FGSContainer, UGSContainer⟩::RET Tg;
    typedef TYPENAME IF⟨type ≡ fold, FGSContainer, UGSContainer⟩::RET Tgs;
    typedef TYPENAME IF⟨type ≡ fold, FGSContainer, UGSContainer⟩::RET Tgss;
    typedef TYPENAME IF⟨type ≡ fold, FGSContainer, UGSContainer⟩::RET TG;
    typedef TYPENAME IF⟨type ≡ fold, FoldedZContainer, UnfoldedZContainer⟩::RET
        TZstack;
    typedef TYPENAME IF⟨type ≡ fold, FoldedGContainer, UnfoldedGContainer⟩::RET
        TGstack;
    typedef TYPENAME IF⟨type ≡ fold, FNormalMoments, UNormalMoments⟩::RET Tm;
    typedef TYPENAME IF⟨type ≡ fold, FTensorPolynomial, UTensorPolynomial⟩::RET Tpol;
    typedef TYPENAME IF⟨type ≡ fold, FoldedZXContainer, UnfoldedZXContainer⟩::RET
        TZXstack;
    typedef TYPENAME IF⟨type ≡ fold, FoldedGXContainer, UnfoldedGXContainer⟩::RET
        TGXstack;
  };
```

This code is used in section 95.

97. The **PartitionY** class defines the partitioning of state variables y. The vector y, and subvector y^*, and y^{**} are defined.

$$y = \begin{bmatrix} \text{static} \\ \text{predeter} \\ \text{both} \\ \text{forward} \end{bmatrix}, \quad y^* = \begin{bmatrix} \text{predeter} \\ \text{both} \end{bmatrix}, \quad y^{**} = \begin{bmatrix} \text{both} \\ \text{forward} \end{bmatrix},$$

where "static" means variables appearing only at time t, "predeter" means variables appearing at time $t-1$, but not at $t+1$, "both" means variables appearing both at $t-1$ and $t+1$ (regardless appearance at t), and "forward" means variables appearing at $t+1$, but not at $t-1$.

The class maintains the four lengths, and returns the whole length, length of y^s, and length of y^{**}.

⟨ **PartitionY** struct declaration 97 ⟩ ≡

```
struct PartitionY {
  const int nstat;
  const int npred;
  const int nboth;
  const int nforw;
  PartitionY(int num_stat, int num_pred, int num_both, int num_forw)
  : nstat(num_stat), npred(num_pred), nboth(num_both), nforw(num_forw) { }
  int ny( ) const
  { return nstat + npred + nboth + nforw; }
  int nys( ) const
  { return npred + nboth; }
  int nyss( ) const
  { return nboth + nforw; }
};
```

This code is cited in section 102.

This code is used in section 95.

98. This is an abstraction for a square matrix with attached PLU factorization. It can calculate the PLU factorization and apply the inverse with some given matrix.

We use LAPACK PLU decomposition for the inverse. We store the L and U in the inv array and $ipiv$ is the permutation P.

⟨ **PLUMatrix** class declaration 98 ⟩ ≡
```
class PLUMatrix : public TwoDMatrix {
public:
  PLUMatrix(int n)
  : TwoDMatrix(n,n), inv(nrows()*ncols()), ipiv(new lapack_int[nrows()]) {}
  PLUMatrix(const PLUMatrix &plu);
  virtual ~PLUMatrix()
  {
    delete[] ipiv;
  }
  void multInv(TwoDMatrix &m) const;
private:
  Vector inv;
  lapack_int *ipiv;
protected:
  void calcPLU();
};
```
This code is used in section 95.

99. The class **MatrixA** is used for matrix $[f_y] + \left[0 \left[f_{y_+^{**}}\right] \cdot [g_{y^*}^{**}]\ 0\right]$, which is central for the perturbation method step.

⟨ **MatrixA** class declaration 99 ⟩ ≡
```
class MatrixA : public PLUMatrix {
public:
  MatrixA(const FSSparseTensor &f, const IntSequence &ss, const TwoDMatrix &gy, const
      PartitionY &ypart);
};
```
This code is cited in section 102.
This code is used in section 95.

100. The class **MatrixS** slightly differs from **MatrixA**. It is used for matrix

$$[f_y] + \left[0 \quad \left[f_{y_+^{**}}\right] \cdot [g_{y^*}^{**}] \quad 0\right] + \left[0 \quad 0 \quad \left[f_{y_+^{**}}\right]\right]$$

, which is needed when recovering g_{σ^k}.

⟨ **MatrixS** class declaration 100 ⟩ ≡
```
class MatrixS : public PLUMatrix {
public:
  MatrixS(const FSSparseTensor &f, const IntSequence &ss, const TwoDMatrix &gy, const
      PartitionY &ypart);
};
```
This code is used in section 95.

101. The B matrix is equal to $\left[f_{y_+^{**}}\right]$. We have just a constructor.

⟨ **MatrixB** class declaration 101 ⟩ ≡
```
class MatrixB : public TwoDMatrix {
public:
  MatrixB(const FSSparseTensor &f, const IntSequence &ss)
  : TwoDMatrix(FGSTensor(f, ss, IntSequence(1, 0), TensorDimens(ss, IntSequence(1, 0)))) {}
};
```
This code is cited in section 102.

This code is used in section 95.

102. Here we have the class for the higher order approximations. It contains the following data:

variable sizes ypart	**PartitionY** struct maintaining partitions of y, see ⟨ **PartitionY** struct declaration 97 ⟩
tensor variable dimension *nvs*	variable sizes of all tensors in containers, sizes of y^*, u, u' and σ
tensor containers	folded and unfolded containers for g, g_{y^*}, g_{y^**} (the latter two collect appropriate subtensors of g, they do not allocate any new space), G, G stack, Z stack
dynamic model derivatives	just a reference to the container of sparse tensors of the system derivatives, lives outside the class
moments	both folded and unfolded normal moment containers, both are calculated at initialization
matrices	matrix A, matrix S, and matrix B, see ⟨ **MatrixA** class declaration 99 ⟩ and ⟨ **MatrixB** class declaration 101 ⟩

The methods are the following:

member access	we declare template methods for accessing containers depending on *fold* and *unfold* flag, we implement their specializations
performStep	this performs k-order step provided that $k = 2$ or the $k-1$-th step has been run, this is the core method
check	this calculates residuals of all solved equations for k-order and reports their sizes, it is runnable after k-order *performStep* has been run
insertDerivative	inserts a g derivative to the g container and also creates subtensors and insert them to g_{y^*} and $g_{y^{**}}$ containers
sylvesterSolve	solve the sylvester equation (templated fold, and unfold)
faaDiBrunoZ	calculates derivatives of F by Faa Di Bruno for the sparse container of system derivatives and Z stack container
faaDiBrunoG	calculates derivatives of G by Faa Di Bruno for the dense container g^{**} and G stack
recover_y	recovers $g_{y^{*i}}$
recover_yu	recovers $g_{y^{*i}u^j}$
recover_ys	recovers $g_{y^{*i}\sigma^j}$
recover_yus	recovers $g_{y^{*i}u^j\sigma^k}$
recover_s	recovers g_{σ^i}
fillG	calculates specified derivatives of G and inserts them to the container
calcE_ijk	calculates E_{ijk}
calcD_ijk	calculates D_{ijk}

Most of the code is templated, and template types are calculated in **ctraits**. So all templated methods get a template argument T, which can be either *fold*, or *unfold*. To shorten a reference to a type calculated by **ctraits** for a particular t, we define the following macros.

```
#define _Ttensor TYPENAME ctraits⟨t⟩::Ttensor
#define _Ttensym TYPENAME ctraits⟨t⟩::Ttensym
#define _Tg TYPENAME ctraits⟨t⟩::Tg
#define _Tgs TYPENAME ctraits⟨t⟩::Tgs
```

```
#define _Tgss TYPENAME ctraits⟨t⟩::Tgss
#define _TG TYPENAME ctraits⟨t⟩::TG
#define _TZstack TYPENAME ctraits⟨t⟩::TZstack
#define _TGstack TYPENAME ctraits⟨t⟩::TGstack
#define _TZXstack TYPENAME ctraits⟨t⟩::TZXstack
#define _TGXstack TYPENAME ctraits⟨t⟩::TGXstack
#define __Tm TYPENAME ctraits⟨t⟩::Tm
#define _Tpol TYPENAME ctraits⟨t⟩::Tpol
⟨ KOrder class declaration 102 ⟩ ≡
  class KOrder {
  protected:
    const PartitionY ypart;
    const int ny;
    const int nu;
    const int maxk;
    IntSequence nvs;
    ⟨ KOrder container members 124 ⟩;
    const MatrixA matA;
    const MatrixS matS;
    const MatrixB matB;
    ⟨ KOrder member access method declarations 125 ⟩;
    Journal &journal;
  public:
    KOrder(int num_stat, int num_pred, int num_both,
        int num_forw, const TensorContainer⟨FSSparseTensor⟩ &fcont, const TwoDMatrix
        &gy, const TwoDMatrix &gu, const TwoDMatrix &v, Journal &jr);
    enum { fold, unfold };
    ⟨ KOrder::performStep templated code 118 ⟩;
    ⟨ KOrder::check templated code 119 ⟩;
    ⟨ KOrder::calcStochShift templated code 123 ⟩;
    void switchToFolded( );
    const PartitionY &getPartY( ) const
    { return ypart; }
    const FGSContainer &getFoldDers( ) const
    { return _fg; }
    const UGSContainer &getUnfoldDers( ) const
    { return _ug; }
    static bool is_even(int i)
    { return (i/2) * 2 ≡ i; }
  protected:
    ⟨ KOrder::insertDerivative templated code 103 ⟩;
    template⟨int t⟩ void sylvesterSolve(_Ttensor &der) const;
    ⟨ KOrder::faaDiBrunoZ templated code 104 ⟩;
    ⟨ KOrder::faaDiBrunoG templated code 105 ⟩;
    ⟨ KOrder::recover_y templated code 106 ⟩;
    ⟨ KOrder::recover_yu templated code 107 ⟩;
    ⟨ KOrder::recover_ys templated code 108 ⟩;
    ⟨ KOrder::recover_yus templated code 109 ⟩;
```

```
    ⟨ KOrder :: recover_s templated code 110 ⟩;
    ⟨ KOrder :: fillG templated code 111 ⟩;
    ⟨ KOrder :: calcD_ijk templated code 112 ⟩;
    ⟨ KOrder :: calcD_ik templated code 114 ⟩;
    ⟨ KOrder :: calcD_k templated code 115 ⟩;
    ⟨ KOrder :: calcE_ijk templated code 113 ⟩;
    ⟨ KOrder :: calcE_ik templated code 116 ⟩;
    ⟨ KOrder :: calcE_k templated code 117 ⟩;
  };
```

This code is cited in section 182.

This code is used in section 95.

103. Here we insert the result to the container. Along the insertion, we also create subtensors and insert as well.

```
⟨ KOrder :: insertDerivative templated code 103 ⟩ ≡
  template⟨int t⟩ void insertDerivative(_Ttensor *der)
  {
    g<t>().insert(der);
    gs<t>().insert(new _Ttensor(ypart.nstat, ypart.nys(), *der));
    gss<t>().insert(new _Ttensor(ypart.nstat + ypart.npred, ypart.nyss(), *der));
  }
```

This code is used in section 102.

104. Here we implement Faa Di Bruno formula

$$\sum_{l=1}^{k} [f_{z^l}]_{\gamma_1 \ldots \gamma_l} \sum_{c \in M_{l,k}} \prod_{m=1}^{l} \left[z_{s(c_m)}\right]^{\gamma_m},$$

where s is a given outer symmetry and k is the dimension of the symmetry.

```
⟨ KOrder :: faaDiBrunoZ templated code 104 ⟩ ≡
  template⟨int t⟩ _Ttensor *faaDiBrunoZ(const Symmetry &sym) const
  {
    JournalRecordPair pa(journal);
    pa << "Faa Di Bruno Z container for " << sym << endrec;
    _Ttensor *res = new _Ttensor(ny, TensorDimens(sym, nvs));
    FaaDiBruno bruno(journal);
    bruno.calculate(Zstack<t>(), f, *res);
    return res;
  }
```

This code is cited in section 105.

This code is used in section 102.

105. The same as ⟨ **KOrder** :: *faaDiBrunoZ* templated code 104 ⟩, but for g^{**} and G stack.

```
⟨ KOrder :: faaDiBrunoG templated code 105 ⟩ ≡
  template⟨int t⟩ _Ttensor *faaDiBrunoG(const Symmetry &sym) const
  {
    JournalRecordPair pa(journal);
    pa << "Faa Di Bruno G container for " << sym << endrec;
    TensorDimens tdims(sym, nvs);
    _Ttensor *res = new _Ttensor(ypart.nyss(), tdims);
    FaaDiBruno bruno(journal);
    bruno.calculate(Gstack<t>(), gss<t>(), *res);
    return res;
  }
```

This code is used in section 102.

106. Here we solve $\left[F_{y^i}\right] = 0$. First we calculate conditional G_{y^i} (it misses $l = 1$ and $l = i$ since g_{y^i} does not exist yet). Then calculate conditional F_{y^i} and we have the right hand side of equation. Since we miss two orders, we solve by Sylvester, and insert the solution as the derivative g_{y^i}. Then we need to update G_{y^i} running *multAndAdd* for both dimensions 1 and i.

Requires: everything at order $\leq i-1$

Provides: g_{y^i}, and G_{y^i}

```
⟨ KOrder :: recover_y templated code 106 ⟩ ≡
  template⟨int t⟩ void recover_y(int i)
  {
    Symmetry sym(i, 0, 0, 0);
    JournalRecordPair pa(journal);
    pa << "Recovering symmetry " << sym << endrec;
    _Ttensor *G_yi = faaDiBrunoG<t>(sym);
    G<t>().insert(G_yi);
    _Ttensor *g_yi = faaDiBrunoZ<t>(sym);
    g_yi->mult(-1.0);
    sylvesterSolve<t>(*g_yi);
    insertDerivative<t>(g_yi);
    _Ttensor *gss_y = gss<t>().get(Symmetry(1, 0, 0, 0));
    gs<t>().multAndAdd(*gss_y, *G_yi);
    _Ttensor *gss_yi = gss<t>().get(sym);
    gs<t>().multAndAdd(*gss_yi, *G_yi);
  }
```

This code is used in section 102.

107. Here we solve $\left[F_{y^iu^j}\right]=0$ to obtain $g_{y^iu^j}$ for $j>0$. We calculate conditional $G_{y^iu^j}$ (this misses only $l=1$) and calculate conditional $F_{y^iu^j}$ and we have the right hand side. It is solved by multiplication of inversion of A. Then we insert the result, and update $G_{y^iu^j}$ by *multAndAdd* for $l=1$.

Requires: everything at order $\le i+j-1$, $G_{y^{i+j}}$, and $g_{y^{i+j}}$.

Provides: $g_{y^iu^j}$, and $G_{y^iu^j}$

⟨ **KOrder** :: *recover_yu* templated code 107 ⟩ ≡

```
template⟨int t⟩ void recover_yu(int i, int j)
{
  Symmetry sym(i, j, 0, 0);
  JournalRecordPair pa(journal);
  pa ≪ "Recovering␣symmetry␣" ≪ sym ≪ endrec;
  _Ttensor *G_yiuj = faaDiBrunoG < t > (sym);
  G < t > ().insert(G_yiuj);
  _Ttensor *g_yiuj = faaDiBrunoZ < t > (sym);
  g_yiuj→mult(−1.0);
  matA.multInv(*g_yiuj);
  insertDerivative < t > (g_yiuj);
  gs < t > ().multAndAdd(*(gss < t > ().get(Symmetry(1, 0, 0, 0))), *G_yiuj);
}
```

This code is used in section 102.

108. Here we solve $\left[F_{y^i\sigma^j}\right]+[D_{ij}]+[E_{ij}]=0$ to obtain $g_{y^i\sigma^j}$. We calculate conditional $G_{y^i\sigma^j}$ (missing dimensions 1 and $i+j$), calculate conditional $F_{y^i\sigma^j}$. Before we can calculate D_{ij} and E_{ij}, we have to calculate $G_{y^iu'^m\sigma^{j-m}}$ for $m=1,\ldots,j$. Then we add the D_{ij} and E_{ij} to obtain the right hand side. Then we solve the sylvester to obtain $g_{y^i\sigma^j}$. Then we update $G_{y^i\sigma^j}$ for $l=1$ and $l=i+j$.

Requires: everything at order $\le i+j-1$, $g_{y^{i+j}}$, $G_{y^iu'^j}$ and $g_{y^iu^j}$ through D_{ij}, $g_{y^iu^m\sigma^{j-m}}$ for $m=1,\ldots,j-1$ through E_{ij}.

Provides: $g_{y^i\sigma^j}$ and $G_{y^i\sigma^j}$, and finally $G_{y^iu'^m\sigma^{j-m}}$ for $m=1,\ldots,j$. The latter is calculated by *fillG* before the actual calculation.

⟨ **KOrder**::*recover_ys* templated code 108 ⟩ ≡

```
template⟨int t⟩ void recover_ys(int i,int j)
{
  Symmetry sym(i,0,0,j);
  JournalRecordPair pa(journal);
  pa ≪ "Recovering␣symmetry␣" ≪ sym ≪ endrec;
  fillG<t>(i,0,j);
  if (is_even(j)) {
    _Ttensor *G_yisj = faaDiBrunoG<t>(sym);
    G<t>().insert(G_yisj);
    _Ttensor *g_yisj = faaDiBrunoZ<t>(sym);
    {
      _Ttensor *D_ij = calcD_ik<t>(i,j);
      g_yisj→add(1.0,*D_ij);
      delete D_ij;
    }
    if (j≥3) {
      _Ttensor *E_ij = calcE_ik<t>(i,j);
      g_yisj→add(1.0,*E_ij);
      delete E_ij;
    }
    g_yisj→mult(-1.0);
    sylvesterSolve<t>(*g_yisj);
    insertDerivative<t>(g_yisj);
    Gstack<t>().multAndAdd(1,gss<t>(),*G_yisj);
    Gstack<t>().multAndAdd(i+j,gss<t>(),*G_yisj);
  }
}
```

This code is used in section 102.

109. Here we solve $\left[F_{y^iu^j\sigma^k}\right]+[D_{ijk}]+[E_{ijk}]=0$ to obtain $g_{y^iu^j\sigma^k}$. First we calculate conditional $G_{y^iu^j\sigma^k}$ (missing only for dimension $l=1$), then we evaluate conditional $F_{y^iu^j\sigma^k}$. Before we can calculate D_{ijk}, and E_{ijk}, we need to insert $G_{y^iu^ju'^m\sigma^{k-m}}$ for $m=1,\ldots,k$. This is done by *fillG*. Then we have right hand side and we multiply by A^{-1} to obtain $g_{y^iu^j\sigma^k}$. Finally we have to update $G_{y^iu^j\sigma^k}$ by *multAndAdd* for dimension $l=1$.

Requires: everything at order $\le i+j+k$, $g_{y^{i+j}\sigma^k}$ through $G_{y^iu^j\sigma^k}$ involved in right hand side, then $g_{y^iu^{j+k}}$ through D_{ijk}, and $g_{y^iu^{j+m}\sigma^{k-m}}$ for $m=1,\ldots,k-1$ through E_{ijk}.

Provides: $g_{y^iu^j\sigma^k}$, $G_{y^iu^j\sigma^k}$, and $G_{y^iu^ju'^m\sigma^{k-m}}$ for $m=1,\ldots,k$

⟨ **KOrder**::*recover_yus* templated code 109 ⟩ ≡

```
template⟨int t⟩ void recover_yus(int i,int j,int k)
{
  Symmetry sym(i,j,0,k);
  JournalRecordPair pa(journal);
  pa ≪ "Recovering␣symmetry␣" ≪ sym ≪ endrec;
  fillG<t>(i,j,k);
  if (is_even(k)) {
    _Ttensor *G_yiujsk = faaDiBrunoG<t>(sym);
    G<t>().insert(G_yiujsk);
    _Ttensor *g_yiujsk = faaDiBrunoZ<t>(sym);
    {
      _Ttensor *D_ijk = calcD_ijk<t>(i,j,k);
      g_yiujsk→add(1.0,*D_ijk);
      delete D_ijk;
    }
    if (k≥3) {
      _Ttensor *E_ijk = calcE_ijk<t>(i,j,k);
      g_yiujsk→add(1.0,*E_ijk);
      delete E_ijk;
    }
    g_yiujsk→mult(-1.0);
    matA.multInv(*g_yiujsk);
    insertDerivative<t>(g_yiujsk);
    Gstack<t>().multAndAdd(1,gss<t>(),*G_yiujsk);
  }
}
```

This code is used in section 102.

110. Here we solve $[F_{\sigma^i}] + [D_i] + [E_i] = 0$ to recover g_{σ^i}. First we calculate conditional G_{σ^i} (missing dimension $l = 1$ and $l = i$), then we calculate conditional F_{σ^i}. Before we can calculate D_i and E_i, we have to obtain $G_{u'^m\sigma^{i-m}}$ for $m = 1, \ldots, i$. Than adding D_i and E_i we have the right hand side. We solve by S^{-1} multiplication and update G_{σ^i} by calling *multAndAdd* for dimension $l = 1$.

Recall that the solved equation here is:

$$[f_y]\,[g_{\sigma^k}] + \left[f_{y_+^{**}}\right]\,\left[g_{y^*}^{**}\right]\,[g_{\sigma^k}^{*}] + \left[f_{y_+^{**}}\right]\,[g_{\sigma^k}^{**}] = \text{RHS}$$

This is a sort of deficient sylvester equation (sylvester equation for dimension=0), we solve it by S^{-1}. See ⟨ **MatrixS** constructor code 132 ⟩ to see how S looks like.

Requires: everything at order $\le i-1$, g_{y^i} and $g_{y^{i-j}\sigma^j}$, then g_{u^k} through $F_{u'^k}$, and $g_{y^m u^j \sigma^k}$ for $j = 1, \ldots, i-1$ and $m + j + k = i$ through $F_{u'^j\sigma^{i-j}}$.

Provides: g_{σ^i}, G_{σ^i}, and $G_{u'^m\sigma^{i-m}}$ for $m = 1, \ldots, i$

⟨ **KOrder**::*recover_s* templated code 110 ⟩ ≡

```
template⟨int t⟩ void recover_s(int i)
{
  Symmetry sym(0,0,0,i);
  JournalRecordPair pa(journal);
  pa ≪ "Recovering symmetry " ≪ sym ≪ endrec;
  fillG<t>(0,0,i);
  if (is_even(i)) {
    _Ttensor *G_si = faaDiBrunoG<t>(sym);
    G<t>().insert(G_si);
    _Ttensor *g_si = faaDiBrunoZ<t>(sym);
    {
      _Ttensor *D_i = calcD_k<t>(i);
      g_si→add(1.0, *D_i);
      delete D_i;
    }
    if (i ≥ 3) {
      _Ttensor *E_i = calcE_k<t>(i);
      g_si→add(1.0, *E_i);
      delete E_i;
    }
    g_si→mult(-1.0);
    matS.multInv(*g_si);
    insertDerivative<t>(g_si);
    Gstack<t>().multAndAdd(1, gss<t>(), *G_si);
    Gstack<t>().multAndAdd(i, gss<t>(), *G_si);
  }
}
```

This code is used in section 102.

111. Here we calculate and insert $G_{y^i u^j u'^m \sigma^{k-m}}$ for $m = 1, \dots, k$. The derivatives are inserted only for $k - m$ being even.

```
⟨ KOrder::fillG templated code 111 ⟩ ≡
  template⟨int t⟩ void fillG(int i, int j, int k)
  {
    for (int m = 1; m ≤ k; m++) {
      if (is_even(k − m)) {
        _Ttensor *G_yiujupms = faaDiBrunoG < t > (Symmetry(i, j, m, k − m));
        G < t > ().insert(G_yiujupms);
      }
    }
  }
```

This code is used in section 102.

112. Here we calculate

$$[D_{ijk}]_{\alpha_1 \dots \alpha_i \beta_1 \dots \beta_j} = \left[F_{y^i u^j u'^k}\right]_{\alpha_1 \dots \alpha_i \beta_1 \dots \beta_j \gamma_1 \dots \gamma_k} [\Sigma]^{\gamma_1 \dots \gamma_k}$$

So it is non zero only for even k.

```
⟨ KOrder::calcD_ijk templated code 112 ⟩ ≡
  template⟨int t⟩ _Ttensor *calcD_ijk(int i, int j, int k) const
  {
    _Ttensor *res = new _Ttensor(ny, TensorDimens(Symmetry(i, j, 0, 0), nvs));
    res→zeros();
    if (is_even(k)) {
      _Ttensor *tmp = faaDiBrunoZ < t > (Symmetry(i, j, k, 0));
      tmp→contractAndAdd(2, *res, *(m < t > ().get(Symmetry(k))));
      delete tmp;
    }
    return res;
  }
```

This code is used in section 102.

113. Here we calculate

$$[E_{ijk}]_{\alpha_1\dots\alpha_i\beta_1\dots\beta_j} = \sum_{m=1}^{k-1} \binom{k}{m} \left[F_{y^i u^j u'^m \sigma^{k-m}}\right]_{\alpha_1\dots\alpha_i\beta_1\dots\beta_j\gamma_1\dots\gamma_m} [\Sigma]^{\gamma_1\dots\gamma_m}$$

The sum can sum only for even m.

⟨ **KOrder**::*calcE_ijk* templated code 113 ⟩ ≡
```
template⟨int t⟩ _Ttensor *calcE_ijk(int i, int j, int k) const
{
  _Ttensor *res = new _Ttensor(ny, TensorDimens(Symmetry(i,j,0,0), nvs));
  res→zeros( );
  for (int n = 2; n ≤ k − 1; n += 2) {
    _Ttensor *tmp = faaDiBrunoZ < t > (Symmetry(i,j,n,k − n));
    tmp→mult((double)( Tensor::noverk(k,n)));
    tmp→contractAndAdd(2, *res, *(m < t > ( ).get(Symmetry(n))));
    delete tmp;
  }
  return res;
}
```
This code is used in section 102.

114.

⟨ **KOrder**::*calcD_ik* templated code 114 ⟩ ≡
```
template⟨int t⟩ _Ttensor *calcD_ik(int i, int k) const
{
  return calcD_ijk < t > (i, 0, k);
}
```
This code is used in section 102.

115.

⟨ **KOrder**::*calcD_k* templated code 115 ⟩ ≡
```
template⟨int t⟩ _Ttensor *calcD_k(int k) const
{
  return calcD_ijk < t > (0, 0, k);
}
```
This code is used in section 102.

116.

⟨ **KOrder**::*calcE_ik* templated code 116 ⟩ ≡
```
template⟨int t⟩ _Ttensor *calcE_ik(int i, int k) const
{
  return calcE_ijk < t > (i, 0, k);
}
```
This code is used in section 102.

117.

```
⟨ KOrder::calcE_k templated code 117 ⟩ ≡
  template⟨int t⟩ _Ttensor *calcE_k(int k) const
  {
    return calcE_ijk < t > (0, 0, k);
  }
```

This code is used in section 102.

118. Here is the core routine. It calls methods recovering derivatives in the right order. Recall, that the code, namely Faa Di Bruno's formula, is implemented as to be run conditionally on the current contents of containers. So, if some call of Faa Di Bruno evaluates derivatives, and some derivatives are not present in the container, then it is considered to be zero. So, we have to be very careful to put everything in the right order. The order here can be derived from dependencies, or it is in the paper.

The method recovers all the derivatives of the given *order*.

The precondition of the method is that all tensors of order $order - 1$, which are not zero, exist (including G). The postcondition of of the method is derivatives of g and G of order *order* are calculated and stored in the containers. Responsibility of precondition lays upon the constructor (for $order \equiv 2$), or upon the previous call of *performStep*.

From the code, it is clear, that all g are calculated. If one goes through all the recovering methods, he should find out that also all G are provided.

```
⟨ KOrder::performStep templated code 118 ⟩ ≡
  template⟨int t⟩ void performStep(int order)
  {
    KORD_RAISE_IF(order − 1 ≠ g < t > ().getMaxDim(), "Wrong␣order␣for␣KOrder::performStep");
    JournalRecordPair pa(journal);
    pa ≪ "Performing␣step␣for␣order␣=␣" ≪ order ≪ endrec;
    recover_y < t > (order);
    for (int i = 0; i < order; i++) {
      recover_yu < t > (i, order − i);
    }
    for (int j = 1; j < order; j++) {
      for (int i = j − 1; i ≥ 1; i−−) {
        recover_yus < t > (order − j, i, j − i);
      }
      recover_ys < t > (order − j, j);
    }
    for (int i = order − 1; i ≥ 1; i−−) {
      recover_yus < t > (0, i, order − i);
    }
    recover_s < t > (order);
  }
```

This code is used in section 102.

119. Here we check for residuals of all the solved equations at the given order. The method returns the largest residual size. Each check simply evaluates the equation.

```
⟨ KOrder::check templated code 119 ⟩ ≡
  template⟨int t⟩ double check(int dim) const
  {
    KORD_RAISE_IF(dim > g < t > ().getMaxDim(), "Wrong dimension for KOrder::check");
    JournalRecordPair pa(journal);
    pa ≪ "Checking residuals for order = " ≪ dim ≪ endrec;
    double maxerror = 0.0;
    ⟨ check for F_{y^i u^j} = 0 120 ⟩;
    ⟨ check for F_{y^i u^j u'^k} + D_{ijk} + E_{ijk} = 0 121 ⟩;
    ⟨ check for F_{σ^i} + D_i + E_i = 0 122 ⟩;
    return maxerror;
  }
```

This code is used in section 102.

120.

```
⟨ check for F_{y^i u^j} = 0 120 ⟩ ≡
  for (int i = 0; i ≤ dim; i++) {
    Symmetry sym(dim − i, i, 0, 0);
    _Ttensor *r = faaDiBrunoZ < t > (sym);
    double err = r→getData().getMax();
    JournalRecord(journal) ≪ "\terror for symmetry " ≪ sym ≪ "\tis " ≪ err ≪ endrec;
    if (err > maxerror) maxerror = err;
    delete r;
  }
```

This code is used in section 119.

121.

```
⟨ check for F_{y^i u^j u'^k} + D_{ijk} + E_{ijk} = 0 121 ⟩ ≡
  SymmetrySet ss(dim, 3);
  for (symiterator si(ss); ¬si.isEnd(); ++si) {
    int i = (*si)[0];
    int j = (*si)[1];
    int k = (*si)[2];
    if (i + j > 0 ∧ k > 0) {
      Symmetry sym(i, j, 0, k);
      _Ttensor *r = faaDiBrunoZ < t > (sym);
      _Ttensor *D_ijk = calcD_ijk < t > (i, j, k);
      r→add(1.0, *D_ijk);
      delete D_ijk;
      _Ttensor *E_ijk = calcE_ijk < t > (i, j, k);
      r→add(1.0, *E_ijk);
      delete E_ijk;
      double err = r→getData().getMax();
      JournalRecord(journal) ≪ "\terror␣for␣symmetry␣" ≪ sym ≪ "\tis␣" ≪ err ≪ endrec;
      delete r;
    }
  }
```

This code is used in section 119.

122.

```
⟨ check for F_{σ^i} + D_i + E_i = 0 122 ⟩ ≡
  _Ttensor *r = faaDiBrunoZ < t > (Symmetry(0, 0, 0, dim));
  _Ttensor *D_k = calcD_k < t > (dim);
  r→add(1.0, *D_k);
  delete D_k;
  _Ttensor *E_k = calcE_k < t > (dim);
  r→add(1.0, *E_k);
  delete E_k;
  double err = r→getData().getMax();
  Symmetry sym(0, 0, 0, dim);
  JournalRecord(journal) ≪ "\terror␣for␣symmetry␣" ≪ sym ≪ "\tis␣" ≪ err ≪ endrec;
  if (err > maxerror) maxerror = err;
  delete r;
```

This code is used in section 119.

123.

```
⟨ KOrder::calcStochShift templated code 123 ⟩ ≡
  template⟨int t⟩ Vector *calcStochShift(int order, double sigma) const
  {
    Vector *res = new Vector(ny);
    res→zeros( );
    int jfac = 1;
    for (int j = 1; j ≤ order; j++, jfac *= j)
      if (is_even(j)) {
        _Ttensor *ten = calcD_k < t > (j);
        res→add(std::pow(sigma, j)/jfac, ten→getData( ));
        delete ten;
      }
    return res;
  }
```

This code is used in section 102.

124. These are containers. The names are not important because they do not appear anywhere else since we access them by template functions.

```
⟨ KOrder container members 124 ⟩ ≡
  UGSContainer _ug;
  FGSContainer _fg;
  UGSContainer _ugs;
  FGSContainer _fgs;
  UGSContainer _ugss;
  FGSContainer _fgss;
  UGSContainer _uG;
  FGSContainer _fG;
  UnfoldedZContainer _uZstack;
  FoldedZContainer _fZstack;
  UnfoldedGContainer _uGstack;
  FoldedGContainer _fGstack;
  UNormalMoments _um;
  FNormalMoments _fm;
  const TensorContainer⟨FSSparseTensor⟩ &f;
```

This code is used in section 102.

125. These are the declarations of the template functions accessing the containers.

```
⟨ KOrder member access method declarations 125 ⟩ ≡
  template⟨int t⟩ _Tg &g( );
  template⟨int t⟩ const _Tg &g( ) const;
  template⟨int t⟩ _Tgs &gs( );
  template⟨int t⟩ const _Tgs &gs( ) const;
  template⟨int t⟩ _Tgss &gss( );
  template⟨int t⟩ const _Tgss &gss( ) const;
  template⟨int t⟩ _TG &G( );
  template⟨int t⟩ const _TG &G( ) const;
  template⟨int t⟩ _TZstack &Zstack( );
  template⟨int t⟩ const _TZstack &Zstack( ) const;
  template⟨int t⟩ _TGstack &Gstack( );
  template⟨int t⟩ const _TGstack &Gstack( ) const;
  template⟨int t⟩ __Tm &m( );
  template⟨int t⟩ const __Tm &m( ) const;
```

This code is used in section 102.

126. End of `korder.h` file.

127. Start of `korder.cpp` file.

```
#include "kord_exception.h"
#include "korder.h"
  ⟨ PLUMatrix copy constructor 128 ⟩;
  ⟨ PLUMatrix::calcPLU code 129 ⟩;
  ⟨ PLUMatrix::multInv code 130 ⟩;
  ⟨ MatrixA constructor code 131 ⟩;
  ⟨ MatrixS constructor code 132 ⟩;
  ⟨ KOrder member access method specializations 139 ⟩;
  ⟨ KOrder::sylvesterSolve unfolded specialization 136 ⟩;
  ⟨ KOrder::sylvesterSolve folded specialization 137 ⟩;
  ⟨ KOrder::switchToFolded code 138 ⟩;
  ⟨ KOrder constructor code 133 ⟩;
```

128.

```
⟨ PLUMatrix copy constructor 128 ⟩ ≡
  PLUMatrix::PLUMatrix(const PLUMatrix &plu)
  : TwoDMatrix(plu), inv(plu.inv), ipiv(new lapack_int[nrows( )]) {
    memcpy(ipiv, plu.ipiv, nrows( ) * sizeof (lapack_int));
  }
```

This code is used in section 127.

129. Here we set *ipiv* and *inv* members of the **PLUMatrix** depending on its content. It is assumed that subclasses will call this method at the end of their constructors.

```
⟨ PLUMatrix::calcPLU code 129 ⟩ ≡
  void PLUMatrix::calcPLU()
  {
    lapack_int info;
    lapack_int rows = nrows();
    inv = (const Vector &) getData();
    dgetrf(&rows, &rows, inv.base(), &rows, ipiv, &info);
  }
```

This code is used in section 127.

130. Here we just call the LAPACK machinery to multiply by the inverse.

```
⟨ PLUMatrix::multInv code 130 ⟩ ≡
  void PLUMatrix::multInv(TwoDMatrix &m) const
  {
    KORD_RAISE_IF(m.nrows() ≠ ncols(), "The matrix is not square in PLUMatrix::multInv");
    lapack_int info;
    lapack_int mcols = m.ncols();
    lapack_int mrows = m.nrows();
    double *mbase = m.getData().base();
    dgetrs("N", &mrows, &mcols, inv.base(), &mrows, ipiv, mbase, &mrows, &info);
    KORD_RAISE_IF(info ≠ 0, "Info!=0 in PLUMatrix::multInv");
  }
```

This code is used in section 127.

131. Here we construct the matrix A. Its dimension is ny, and it is

$$A = [f_y] + \left[0 \left[f_{y_+^{**}}\right] \cdot \left[g_{y^*}^{**}\right] 0\right]$$

, where the first zero spans *nstat* columns, and last zero spans *nforw* columns.

```
⟨ MatrixA constructor code 131 ⟩ ≡
  MatrixA::MatrixA(const FSSparseTensor &f, const IntSequence &ss, const TwoDMatrix
        &gy, const PartitionY &ypart)
  : PLUMatrix(ypart.ny()) {
    zeros();
    IntSequence c(1);
    c[0] = 1;
    FGSTensor f_y(f, ss, c, TensorDimens(ss, c));
    add(1.0, f_y);
    ConstTwoDMatrix gss_ys(ypart.nstat + ypart.npred, ypart.nyss(), gy);
    c[0] = 0;
    FGSTensor f_yss(f, ss, c, TensorDimens(ss, c));
    TwoDMatrix sub(*this, ypart.nstat, ypart.nys());
    sub.multAndAdd(ConstTwoDMatrix(f_yss), gss_ys);
    calcPLU();
  }
```

This code is cited in section 167.

This code is used in section 127.

132. Here we construct the matrix S. Its dimension is ny, and it is

$$S = [f_y] + \left[0 \quad \left[f_{y_+^{**}}\right] \cdot \left[g_{y^*}^{**}\right] \quad 0\right] + \left[0 \quad 0 \quad \left[f_{y_+^{**}}\right]\right]$$

It is, in fact, the matrix A plus the third summand. The first zero in the summand spans *nstat* columns, the second zero spans *npred* columns.

⟨ **MatrixS** constructor code 132 ⟩ ≡

```
MatrixS::MatrixS(const FSSparseTensor &f, const IntSequence &ss, const TwoDMatrix
        &gy, const PartitionY &ypart)
 : PLUMatrix(ypart.ny()) {
   zeros();
   IntSequence c(1);
   c[0] = 1;
   FGSTensor f_y(f, ss, c, TensorDimens(ss, c));
   add(1.0, f_y);
   ConstTwoDMatrix gss_ys(ypart.nstat + ypart.npred, ypart.nyss(), gy);
   c[0] = 0;
   FGSTensor f_yss(f, ss, c, TensorDimens(ss, c));
   TwoDMatrix sub(*this, ypart.nstat, ypart.nys());
   sub.multAndAdd(ConstTwoDMatrix(f_yss), gss_ys);
   TwoDMatrix sub2(*this, ypart.nstat + ypart.npred, ypart.nyss());
   sub2.add(1.0, f_yss);
   calcPLU();
 }
```

This code is cited in section 110.

This code is used in section 127.

133. Here is the constructor of the **KOrder** class. We pass what we have to. The partitioning of the y vector, a sparse container with model derivatives, then the first order approximation, these are g_y and g_u matrices, and covariance matrix of exogenous shocks v.

We build the members, it is nothing difficult. Note that we do not make a physical copy of sparse tensors, so during running the class, the outer world must not change them.

In the body, we have to set *nvs* array, and initialize g and G containers to comply to preconditions of *performStep*.

⟨ **KOrder** constructor code 133 ⟩ ≡

```
KOrder::KOrder(int num_stat, int num_pred, int num_both, int num_forw, const
        TensorContainer<FSSparseTensor> &fcont, const TwoDMatrix &gy, const
        TwoDMatrix &gu, const TwoDMatrix &v, Journal &jr)
: ypart(num_stat, num_pred, num_both, num_forw),
ny(ypart.ny()), nu(gu.ncols()), maxk(fcont.getMaxDim()),
nvs(4),
_ug(4), _fg(4), _ugs(4), _fgs(4), _ugss(4), _fgss(4),
_uG(4), _fG(4),
_uZstack(&_uG, ypart.nyss(), &_ug, ny, ypart.nys(), nu),
_fZstack(&_fG, ypart.nyss(), &_fg, ny, ypart.nys(), nu),
_uGstack(&_ugs, ypart.nys(), nu),
_fGstack(&_fgs, ypart.nys(), nu),
_um(maxk, v), _fm(_um), f(fcont),
matA(*(f.get(Symmetry(1)))), _uZstack.getStackSizes(), gy, ypart),
matS(*(f.get(Symmetry(1)))), _uZstack.getStackSizes(), gy, ypart),
matB(*(f.get(Symmetry(1)))), _uZstack.getStackSizes()),
journal(jr)
{
  KORD_RAISE_IF(gy.ncols() ≠ ypart.nys(),
      "Wrong number of columns in gy in KOrder constructor");
  KORD_RAISE_IF(v.ncols() ≠ nu, "Wrong number of columns of Vcov in KOrder constructor");
  KORD_RAISE_IF(nu ≠ v.nrows(), "Wrong number of rows of Vcov in KOrder constructor");
  KORD_RAISE_IF(maxk < 2,
      "Order of approximation must be at least 2 in KOrder constructor");
  KORD_RAISE_IF(gy.nrows() ≠ ypart.ny(), "Wrong number of rows in gy in KOrder constructor");
  KORD_RAISE_IF(gu.nrows() ≠ ypart.ny(),
      "Wrong number of rows in gu in KOrder constructor");
  KORD_RAISE_IF(gu.ncols() ≠ nu, "Wrong number of columns in gu in KOrder constructor");
    /* set nvs: */
  nvs[0] = ypart.nys();
  nvs[1] = nu;
  nvs[2] = nu;
  nvs[3] = 1;
  ⟨ put g_y and g_u to the container 134 ⟩;
  ⟨ put G_y, G_u and G_u' to the container 135 ⟩;
}
```

This code is used in section 127.

134. Note that g_σ is zero by the nature and we do not insert it to the container. We insert a new physical copies.

⟨ put g_y and g_u to the container 134 ⟩ ≡

```
UGSTensor *tgy = new UGSTensor(ny, TensorDimens(Symmetry(1,0,0,0), nvs));
tgy->getData() = gy.getData();
insertDerivative<unfold>(tgy);
UGSTensor *tgu = new UGSTensor(ny, TensorDimens(Symmetry(0,1,0,0), nvs));
tgu->getData() = gu.getData();
insertDerivative<unfold>(tgu);
```

This code is used in section 133.

135. Also note that since g_σ is zero, so G_σ.

⟨ put G_y, G_u and $G_{u'}$ to the container 135 ⟩ ≡

```
UGSTensor *tGy = faaDiBrunoG<unfold>(Symmetry(1,0,0,0));
G<unfold>().insert(tGy);
UGSTensor *tGu = faaDiBrunoG<unfold>(Symmetry(0,1,0,0));
G<unfold>().insert(tGu);
UGSTensor *tGup = faaDiBrunoG<unfold>(Symmetry(0,0,1,0));
G<unfold>().insert(tGup);
```

This code is used in section 133.

136. Here we have an unfolded specialization of *sylvesterSolve*. We simply create the sylvester object and solve it. Note that the g_y^* is not continuous in memory as assumed by the sylvester code, so we make a temporary copy and pass it as matrix C.

If the B matrix is empty, in other words there are now forward looking variables, then the system becomes $AX = D$ which is solved by simple *matA.multInv*( ).

If one wants to display the diagnostic messages from the Sylvester module, then after the *sylv.solve*( ) one needs to call *sylv.getParams*( ).*print*("").

⟨ **KOrder**::*sylvesterSolve* unfolded specialization 136 ⟩ ≡

```
template<>
void KOrder::sylvesterSolve<KOrder::unfold>(ctraits<unfold>::Ttensor &der) const
{
  JournalRecordPair pa(journal);
  pa << "Sylvester equation for dimension= " << der.getSym()[0] << endrec;
  if (ypart.nys() > 0 && ypart.nyss() > 0) {
    KORD_RAISE_IF(!der.isFinite(), "RHS of Sylverster is not finite");
    TwoDMatrix gs_y(*(gs<unfold>().get(Symmetry(1,0,0,0))));
    GeneralSylvester sylv(der.getSym()[0], ny, ypart.nys(), ypart.nstat + ypart.npred,
        matA.getData().base(), matB.getData().base(), gs_y.getData().base(), der.getData().base());
    sylv.solve();
  }
  else if (ypart.nys() > 0 && ypart.nyss() == 0) {
    matA.multInv(der);
  }
}
```

This code is used in section 127.

137. Here is the folded specialization of sylvester. We unfold the right hand side. Then we solve it by the unfolded version of *sylvesterSolve*, and fold it back and copy to output vector.

```
⟨ KOrder::sylvesterSolve folded specialization 137 ⟩ ≡
  template⟨⟩
  void KOrder::sylvesterSolve < KOrder::fold > (ctraits⟨fold⟩::Ttensor &der) const
  {
    ctraits⟨unfold⟩::Ttensor tmp(der);
    sylvesterSolve < unfold > (tmp);
    ctraits⟨fold⟩::Ttensor ftmp(tmp);
    der.getData() = (const Vector &)(ftmp.getData());
  }
```

This code is used in section 127.

138.

```
⟨ KOrder::switchToFolded code 138 ⟩ ≡
  void KOrder::switchToFolded()
  {
    JournalRecordPair pa(journal);
    pa ≪ "Switching␣from␣unfolded␣to␣folded" ≪ endrec;
    int maxdim = g < unfold > ().getMaxDim();
    for (int dim = 1; dim ≤ maxdim; dim++) {
      SymmetrySet ss(dim, 4);
      for (symiterator si(ss); ¬si.isEnd(); ++si) {
        if ((∗si)[2] ≡ 0 ∧ g < unfold > ().check(∗si)) {
          FGSTensor ∗ft = new FGSTensor(∗(g < unfold > ().get(∗si)));
          insertDerivative < fold > (ft);
          if (dim > 1) {
            gss < unfold > ().remove(∗si);
            gs < unfold > ().remove(∗si);
            g < unfold > ().remove(∗si);
          }
        }
        if (G < unfold > ().check(∗si)) {
          FGSTensor ∗ft = new FGSTensor(∗(G < unfold > ().get(∗si)));
          G < fold > ().insert(ft);
          if (dim > 1) {
            G < fold > ().remove(∗si);
          }
        }
      }
    }
  }
```

This code is used in section 127.

139. These are the specializations of container access methods. Nothing interesting here.

⟨ **KOrder** member access method specializations 139 ⟩ ≡

```
template⟨⟩ ctraits⟨KOrder::unfold⟩::Tg &KOrder::g < KOrder::unfold > ()
{ return _ug; }
template⟨⟩ const ctraits⟨KOrder::unfold⟩::Tg &KOrder::g < KOrder::unfold > () const
{ return _ug; }
template⟨⟩ ctraits⟨KOrder::fold⟩::Tg &KOrder::g < KOrder::fold > ()
{ return _fg; }
template⟨⟩ const ctraits⟨KOrder::fold⟩::Tg &KOrder::g < KOrder::fold > () const
{ return _fg; }
template⟨⟩ ctraits⟨KOrder::unfold⟩::Tgs &KOrder::gs < KOrder::unfold > ()
{ return _ugs; }
template⟨⟩ const ctraits⟨KOrder::unfold⟩::Tgs &KOrder::gs < KOrder::unfold > () const
{ return _ugs; }
template⟨⟩ ctraits⟨KOrder::fold⟩::Tgs &KOrder::gs < KOrder::fold > ()
{ return _fgs; }
template⟨⟩ const ctraits⟨KOrder::fold⟩::Tgs &KOrder::gs < KOrder::fold > () const
{ return _fgs; }
template⟨⟩ ctraits⟨KOrder::unfold⟩::Tgss &KOrder::gss < KOrder::unfold > ()
{ return _ugss; }
template⟨⟩ const ctraits⟨KOrder::unfold⟩::Tgss &KOrder::gss < KOrder::unfold > () const
{ return _ugss; }
template⟨⟩ ctraits⟨KOrder::fold⟩::Tgss &KOrder::gss < KOrder::fold > ()
{ return _fgss; }
template⟨⟩ const ctraits⟨KOrder::fold⟩::Tgss &KOrder::gss < KOrder::fold > () const
{ return _fgss; }
template⟨⟩ ctraits⟨KOrder::unfold⟩::TG &KOrder::G < KOrder::unfold > ()
{ return _uG; }
template⟨⟩ const ctraits⟨KOrder::unfold⟩::TG &KOrder::G < KOrder::unfold > () const
{ return _uG; }
template⟨⟩ ctraits⟨KOrder::fold⟩::TG &KOrder::G < KOrder::fold > ()
{ return _fG; }
template⟨⟩ const ctraits⟨KOrder::fold⟩::TG &KOrder::G < KOrder::fold > () const
{ return _fG; }
template⟨⟩ ctraits⟨KOrder::unfold⟩::TZstack &KOrder::Zstack < KOrder::unfold > ()
{ return _uZstack; }
template⟨⟩ const ctraits⟨KOrder::unfold⟩::TZstack &KOrder::Zstack < KOrder::unfold > ()
    const
{ return _uZstack; }
template⟨⟩ ctraits⟨KOrder::fold⟩::TZstack &KOrder::Zstack < KOrder::fold > ()
{ return _fZstack; }
template⟨⟩ const ctraits⟨KOrder::fold⟩::TZstack &KOrder::Zstack < KOrder::fold > () const
{ return _fZstack; }
template⟨⟩ ctraits⟨KOrder::unfold⟩::TGstack &KOrder::Gstack < KOrder::unfold > ()
{ return _uGstack; }
template⟨⟩ const ctraits⟨KOrder::unfold⟩::TGstack &KOrder::Gstack < KOrder::unfold > ()
    const
{ return _uGstack; }
template⟨⟩ ctraits⟨KOrder::fold⟩::TGstack &KOrder::Gstack < KOrder::fold > ()
{ return _fGstack; }
template⟨⟩ const ctraits⟨KOrder::fold⟩::TGstack &KOrder::Gstack < KOrder::fold > () const
{ return _fGstack; }
```

```
template⟨⟩ ctraits⟨KOrder::unfold⟩::Tm &KOrder::m < KOrder::unfold > ()
{ return _um; }
template⟨⟩ const ctraits⟨KOrder::unfold⟩::Tm &KOrder::m < KOrder::unfold > () const
{ return _um; }
template⟨⟩ ctraits⟨KOrder::fold⟩::Tm &KOrder::m < KOrder::fold > ()
{ return _fm; }
template⟨⟩ const ctraits⟨KOrder::fold⟩::Tm &KOrder::m < KOrder::fold > () const
{ return _fm; }
```

This code is used in section 127.

140. End of `korder.cpp` file.

141. Higher order at stochastic steady. Start of `korder_stoch.h` file.

This file defines a number of classes of which **KOrderStoch** is the main purpose. Basically, **KOrderStoch** calculates first and higher order Taylor expansion of a policy rule at $\sigma > 0$ with explicit forward g^{**}. More formally, we have to solve a policy rule g from

$$E_t\left[f(g^{**}(g^*(y_t^*, u_t, \sigma), u_{t+1}, \sigma), g(y^*, u_t, \sigma), y^*, u_t)\right]$$

As an introduction in `approximation.hweb` argues, g^{**} at tine $t+1$ must be given from outside. Let the explicit $E_t(g^{**}(y^*, u_{t+1}, \sigma)$ be equal to $h(y^*, \sigma)$. Then we have to solve

$$f(h(g^*(y^*, u, \sigma), \sigma), g(y, u, \sigma), y, u),$$

which is much easier than fully implicit system for $\sigma = 0$.

Besides the class **KOrderStoch**, we declare here also classes for the new containers corresponding to $f(h(g^*(y^*, u, \sigma), \sigma), g(y, u, \sigma), y, u)$. Further, we declare **IntegDerivs** and **StochForwardDerivs** classes which basically calculate h as an extrapolation based on an approximation to g at lower σ.

```
#include "korder.h"
#include "faa_di_bruno.h"
#include "journal.h"
⟨IntegDerivs class declaration 142⟩;
⟨StochForwardDerivs class declaration 145⟩;
⟨GXContainer class declaration 151⟩;
⟨ZXContainer class declaration 153⟩;
⟨UnfoldedGXContainer class declaration 155⟩;
⟨FoldedGXContainer class declaration 156⟩;
⟨UnfoldedZXContainer class declaration 157⟩;
⟨FoldedZXContainer class declaration 158⟩;
⟨MatrixAA class declaration 159⟩;
⟨KOrderStoch class declaration 160⟩;
```

142. This class is a container, which has a specialized constructor integrating the policy rule at given σ.

```
⟨IntegDerivs class declaration 142⟩ ≡
template⟨int t⟩ class IntegDerivs : public ctraits⟨t⟩::Tgss {
public:
  ⟨IntegDerivs constructor code 143⟩;
};
```

This code is used in section 141.

143. This constructor integrates a rule (namely its g^{**} part) with respect to $u = \tilde{\sigma}\eta$, and stores to the object the derivatives of this integral h at $(y^*, u, \sigma) = (\tilde{y}^*, 0, \tilde{\sigma})$. The original container of g^{**}, the moments of the stochastic shocks *mom* and the $\tilde{\sigma}$ are input.

The code follows the following derivation

$$\begin{aligned}
h(y,\sigma) &= E_t\left[g(y,u',\sigma)\right] = \\
&= \tilde{y} + \sum_{d=1}\frac{1}{d!}\sum_{i+j+k=d}\binom{d}{i,j,k}\left[g_{y^i u^j \sigma^k}\right](y^*-\tilde{y}^*)^i\sigma^j\Sigma^j(\sigma-\tilde{\sigma})^k \\
&= \tilde{y} + \sum_{d=1}\frac{1}{d!}\sum_{i+m+n+k=d}\binom{d}{i,m+n,k}\left[g_{y^i u^{m+n} \sigma^k}\right]\hat{y}^{*i}\Sigma^{m+n}\binom{m+n}{m,n}\tilde{\sigma}^m\hat{\sigma}^{k+n} \\
&= \tilde{y} + \sum_{d=1}\frac{1}{d!}\sum_{i+m+n+k=d}\binom{d}{i,m,n,k}\left[g_{y^i u^{m+n} \sigma^k}\right]\Sigma^{m+n}\tilde{\sigma}^m\hat{y}^{*i}\hat{\sigma}^{k+n} \\
&= \tilde{y} + \sum_{d=1}\frac{1}{d!}\sum_{i+p=d}\sum_{\substack{m=0\\n+k=p}}\binom{d}{i,m,n,k}\left[g_{y^i u^{m+n} \sigma^k}\right]\Sigma^{m+n}\tilde{\sigma}^m\hat{y}^{*i}\hat{\sigma}^{k+n} \\
&= \tilde{y} + \sum_{d=1}\frac{1}{d!}\sum_{i+p=d}\binom{d}{i,p}\left[\sum_{\substack{m=0\\n+k=p}}\binom{p}{n,k}\frac{1}{m!}\left[g_{y^i u^{m+n} \sigma^k}\right]\Sigma^{m+n}\tilde{\sigma}^m\right]\hat{y}^{*i}\hat{\sigma}^{k+n},
\end{aligned}$$

where $\binom{a}{b_1,\ldots,b_n}$ is a generalized combination number, $p = k+n$, $\hat{\sigma} = \sigma - \tilde{\sigma}$, $\hat{y}^* = y^* - \tilde{y}$, and we dropped writing the multidimensional indexes in Einstein summation.

This implies that

$$h_{y^i\sigma^p} = \sum_{\substack{m=0\\n+k=p}}\binom{p}{n,k}\frac{1}{m!}\left[g_{y^i u^{m+n} \sigma^k}\right]\Sigma^{m+n}\tilde{\sigma}^m$$

and this is exactly what the code does.

⟨ **IntegDerivs** constructor code 143 ⟩ ≡

```
IntegDerivs(int r, const IntSequence &nvs, const _Tgss &g, const __Tm &mom, double
        at_sigma)
: ctraits⟨t⟩::Tgss(4) {
  int maxd = g.getMaxDim( );
  for (int d = 1; d ≤ maxd; d++) {
    for (int i = 0; i ≤ d; i++) {
      int p = d − i;
      Symmetry sym(i, 0, 0, p);
      _Ttensor *ten = new _Ttensor(r, TensorDimens(sym, nvs));
      ⟨ calculate derivative h_{y^i σ^p} 144 ⟩;
      this→insert(ten);
    }
  }
}
```

This code is used in section 142.

144. This code calculates

$$h_{y^i\sigma^p} = \sum_{\substack{m=0\\ n+k=p}} \binom{p}{n,k} \frac{1}{m!} \left[g_{y^i u^{m+n}\sigma^k}\right] \Sigma^{m+n} \tilde{\sigma}^m$$

and stores it in *ten*.

⟨ calculate derivative $h_{y^i\sigma^p}$ 144 ⟩ ≡
```
ten→zeros( );
for (int n = 0; n ≤ p; n++) {
  int k = p − n;
  int povern = Tensor::noverk(p, n);
  int mfac = 1;
  for (int m = 0; i + m + n + k ≤ maxd; m++, mfac *= m) {
    double mult = (pow(at_sigma, m) * povern)/mfac;
    Symmetry sym_mn(i, m + n, 0, k);
    if (m + n ≡ 0 ∧ g.check(sym_mn)) ten→add(mult, *(g.get(sym_mn)));
    if (m + n > 0 ∧ KOrder::is_even(m + n) ∧ g.check(sym_mn)) {
      _Ttensor gtmp(*(g.get(sym_mn)));
      gtmp.mult(mult);
      gtmp.contractAndAdd(1, *ten, *(mom.get(Symmetry(m + n))));
    }
  }
}
```
This code is used in section 143.

145. This class calculates an extrapolation of expectation of forward derivatives. It is a container, all calculations are done in a constructor.

The class calculates derivatives of $E[g(y*, u, \sigma)]$ at $(\bar{y}^*, \bar{\sigma})$. The derivatives are extrapolated based on derivatives at $(\tilde{y}^*, \tilde{\sigma})$.

⟨ **StochForwardDerivs** class declaration 145 ⟩ ≡
```
template⟨int t⟩ class StochForwardDerivs : public ctraits⟨t⟩::Tgss {
public:
  ⟨ StochForwardDerivs constructor code 146 ⟩;
};
```
This code is used in section 141.

146. This is the constructor which performs the integration and the extrapolation. Its parameters are: g is the container of derivatives at $(\tilde{y},\tilde{\sigma})$; m are the moments of stochastic shocks; *ydelta* is a difference of the steady states $\bar{y}-\tilde{y}$; *sdelta* is a difference between new sigma and old sigma $\bar{\sigma}-\tilde{\sigma}$, and *at_sigma* is $\tilde{\sigma}$. There is no need of inputing the $\tilde{y}$.

We do the operation in four steps:

1) Integrate g^{**}, the derivatives are at $(\tilde{y},\tilde{\sigma})$
2) Form the (full symmetric) polynomial from the derivatives stacking $\begin{bmatrix} y^* \\ \sigma \end{bmatrix}$
3) Centralize this polynomial about $(\bar{y},\bar{\sigma})$
4) Recover general symmetry tensors from the (full symmetric) polynomial

```
⟨ StochForwardDerivs constructor code 146 ⟩ ≡
  StochForwardDerivs(const PartitionY &ypart, int nu, const _Tgss &g, const __Tm &m, const
          Vector &ydelta, double sdelta, double at_sigma)
  : ctraits⟨t⟩::Tgss(4) {
    int maxd = g.getMaxDim();
    int r = ypart.nyss();
    ⟨ make g_int be integral of g** at (ỹ,σ̃) 147 ⟩;
    ⟨ make g_int_sym be full symmetric polynomial from g_int 148 ⟩;
    ⟨ make g_int_cent the centralized polynomial about (ȳ,σ̄) 149 ⟩;
    ⟨ pull out general symmetry tensors from g_int_cent 150 ⟩;
  }
```

This code is used in section 145.

147. This simply constructs **IntegDerivs** class. Note that the *nvs* of the tensors has zero dimensions for shocks, this is because we need to make easily stacks of the form $\begin{bmatrix} y^* \\ \sigma \end{bmatrix}$ in the next step.

```
⟨ make g_int be integral of g** at (ỹ,σ̃) 147 ⟩ ≡
  IntSequence nvs(4);
  nvs[0] = ypart.nys();
  nvs[1] = 0;
  nvs[2] = 0;
  nvs[3] = 1;
  IntegDerivs⟨t⟩ g_int(r, nvs, g, m, at_sigma);
```

This code is used in section 146.

148. Here we just form a polynomial whose unique variable corresponds to $\begin{bmatrix} y^* \\ \sigma \end{bmatrix}$ stack.

```
⟨ make g_int_sym be full symmetric polynomial from g_int 148 ⟩ ≡
  _Tpol g_int_sym(r, ypart.nys() + 1);
  for (int d = 1; d ≤ maxd; d++) {
    _Ttensym *ten = new _Ttensym(r, ypart.nys() + 1, d);
    ten→zeros();
    for (int i = 0; i ≤ d; i++) {
      int k = d − i;
      if (g_int.check(Symmetry(i, 0, 0, k))) ten→addSubTensor(*(g_int.get(Symmetry(i, 0, 0, k))));
    }
    g_int_sym.insert(ten);
  }
```

This code is used in section 146.

149. Here we centralize the polynomial to $(\bar{y}, \bar{\sigma})$ knowing that the polynomial was centralized about $(\tilde{y}, \tilde{\sigma})$. This is done by derivating and evaluating the derivated polynomial at $(\bar{y} - \tilde{y}, \bar{\sigma} - \tilde{\sigma})$. The stack of this vector is *delta* in the code.

⟨ make *g_int_cent* the centralized polynomial about $(\bar{y}, \bar{\sigma})$ 149 ⟩ ≡

```
Vector delta(ypart.nys() + 1);
Vector dy(delta, 0, ypart.nys());
ConstVector dy_in(ydelta, ypart.nstat, ypart.nys());
dy = dy_in;
delta[ypart.nys()] = sdelta;
_Tpol g_int_cent(r, ypart.nys() + 1);
for (int d = 1; d ≤ maxd; d++) {
  g_int_sym.derivative(d − 1);
  _Ttensym *der = g_int_sym.evalPartially(d, delta);
  g_int_cent.insert(der);
}
```

This code is used in section 146.

150. Here we only recover the general symmetry derivatives from the full symmetric polynomial. Note that the derivative get the true *nvs*.

⟨ pull out general symmetry tensors from *g_int_cent* 150 ⟩ ≡

```
IntSequence ss(4);
ss[0] = ypart.nys();
ss[1] = 0;
ss[2] = 0;
ss[3] = 1;
IntSequence pp(4);
pp[0] = 0;
pp[1] = 1;
pp[2] = 2;
pp[3] = 3;
IntSequence true_nvs(nvs);
true_nvs[1] = nu;
true_nvs[2] = nu;
for (int d = 1; d ≤ maxd; d++) {
  if (g_int_cent.check(Symmetry(d))) {
    for (int i = 0; i ≤ d; i++) {
      Symmetry sym(i, 0, 0, d − i);
      IntSequence coor(sym, pp);
      _Ttensor *ten = new _Ttensor(*(g_int_cent.get(Symmetry(d))), ss, coor,
          TensorDimens(sym, true_nvs));
      this→insert(ten);
    }
  }
}
```

This code is used in section 146.

151. This container corresponds to $h(g^*(y,u,\sigma),\sigma)$. Note that in our application, the σ as a second argument to h will be its fourth variable in symmetry, so we have to do four member stack having the second and third stack dummy.

```
⟨ GXContainer class declaration 151 ⟩ ≡
  template⟨class _Ttype⟩ class GXContainer : public GContainer⟨_Ttype⟩ {
public: typedef StackContainerInterface < _Ttype > _Stype;
  typedef typename StackContainer⟨_Ttype⟩::_Ctype _Ctype;
  typedef typename StackContainer⟨_Ttype⟩::itype itype;
  GXContainer(const _Ctype *gs, int ngs, int nu)
  : GContainer⟨_Ttype⟩(gs, ngs, nu) { }
  ⟨ GXContainer::getType code 152 ⟩;
  } ;
```

This code is used in section 141.

152. This routine corresponds to this stack:

$$\begin{bmatrix} g^*(y,u,\sigma) \\ dummy \\ dummy \\ \sigma \end{bmatrix}$$

```
⟨ GXContainer::getType code 152 ⟩ ≡
  itype getType(int i, const Symmetry &s) const
  {
    if (i ≡ 0)
      if (s[2] > 0) return _Stype::zero;
      else return _Stype::matrix;
    if (i ≡ 1) return _Stype::zero;
    if (i ≡ 2) return _Stype::zero;
    if (i ≡ 3)
      if (s ≡ Symmetry(0,0,0,1)) return _Stype::unit;
      else return _Stype::zero;
    KORD_RAISE("Wrong␣stack␣index␣in␣GXContainer::getType");
  }
```

This code is used in section 151.

153. This container corresponds to $f(H(y,u,\sigma),g(y,u,sigma),y,u)$, where the H has the size (number of rows) as g^{**}. Since it is very simmilar to **ZContainer**, we inherit form it and override only *getType* method.

```
⟨ ZXContainer class declaration 153 ⟩ ≡
  template⟨class _Ttype⟩ class ZXContainer : public ZContainer⟨_Ttype⟩ {
  public:
    typedef StackContainerInterface⟨_Ttype⟩ _Stype;
    typedef typename StackContainer⟨_Ttype⟩::_Ctype _Ctype;
    typedef typename StackContainer⟨_Ttype⟩::itype itype;
    ZXContainer(const _Ctype *gss, int ngss, const _Ctype *g, int ng, int ny, int nu)
    : ZContainer⟨_Ttype⟩(gss, ngss, g, ng, ny, nu) { }
    ⟨ ZXContainer::getType code 154 ⟩;
  };
```

This code is used in section 141.

154. This *getType* method corresponds to this stack:

$$\begin{bmatrix} H(y,u,\sigma) \\ g(y,u,\sigma) \\ y \\ u \end{bmatrix}$$

⟨ **ZXContainer**::*getType* code 154 ⟩ ≡
```
itype getType(int i, const Symmetry &s) const
{
  if (i ≡ 0)
    if (s[2] > 0) return _Stype::zero;
    else return _Stype::matrix;
  if (i ≡ 1)
    if (s[2] > 0) return _Stype::zero;
    else return _Stype::matrix;
  if (i ≡ 2)
    if (s ≡ Symmetry(1,0,0,0)) return _Stype::unit;
    else return _Stype::zero;
  if (i ≡ 3)
    if (s ≡ Symmetry(0,1,0,0)) return _Stype::unit;
    else return _Stype::zero;
  KORD_RAISE("Wrong␣stack␣index␣in␣ZXContainer::getType");
}
```
This code is used in section 153.

155.

⟨ **UnfoldedGXContainer** class declaration 155 ⟩ ≡
```
class UnfoldedGXContainer : public GXContainer⟨UGSTensor⟩, public
      UnfoldedStackContainer {
public:
  typedef TensorContainer⟨UGSTensor⟩ _Ctype;
  UnfoldedGXContainer(const _Ctype *gs, int ngs, int nu)
  : GXContainer⟨UGSTensor⟩(gs, ngs, nu) { }
};
```
This code is used in section 141.

156.

⟨ **FoldedGXContainer** class declaration 156 ⟩ ≡
```
class FoldedGXContainer : public GXContainer⟨FGSTensor⟩, public FoldedStackContainer
      {
public:
  typedef TensorContainer⟨FGSTensor⟩ _Ctype;
  FoldedGXContainer(const _Ctype *gs, int ngs, int nu)
  : GXContainer⟨FGSTensor⟩(gs, ngs, nu) { }
};
```
This code is used in section 141.

157.

⟨ **UnfoldedZXContainer** class declaration 157 ⟩ ≡
```
class UnfoldedZXContainer : public ZXContainer⟨UGSTensor⟩,
        public UnfoldedStackContainer {
public:
  typedef TensorContainer⟨UGSTensor⟩ _Ctype;
  UnfoldedZXContainer(const _Ctype *gss, int ngss, const _Ctype *g, int ng, int ny, int nu)
  : ZXContainer⟨UGSTensor⟩(gss, ngss, g, ng, ny, nu) { }
};
```
This code is used in section 141.

158.

⟨ **FoldedZXContainer** class declaration 158 ⟩ ≡
```
class FoldedZXContainer : public ZXContainer⟨FGSTensor⟩, public FoldedStackContainer {
public:
  typedef TensorContainer⟨FGSTensor⟩ _Ctype;
  FoldedZXContainer(const _Ctype *gss, int ngss, const _Ctype *g, int ng, int ny, int nu)
  : ZXContainer⟨FGSTensor⟩(gss, ngss, g, ng, ny, nu) { }
};
```
This code is used in section 141.

159. This matrix corresponds to

$$[f_y] + \left[0 \left[f_{y^{**}_+}\right] \cdot \left[h^{**}_{y^*}\right] 0\right]$$

This is very the same as **MatrixA**, the only difference that the **MatrixA** is constructed from whole h_{y^*}, not only from $h^{**}_{y^*}$, hence the new abstraction.

⟨ **MatrixAA** class declaration 159 ⟩ ≡
```
class MatrixAA : public PLUMatrix {
public:
  MatrixAA(const FSSparseTensor &f, const IntSequence &ss, const TwoDMatrix
      &gyss, const PartitionY &ypart);
};
```
This code is used in section 141.

160. This class calculates derivatives of g given implicitly by $f(h(g^*(y,u,\sigma),\sigma),g(y,u,\sigma),y,u)$, where $h(y,\sigma)$ is given from outside.

Structurally, the class is very similar to **KOrder**, but calculations are much easier. The two constructors construct an object from sparse derivatives of f, and derivatives of h. The caller must ensure that the both derivatives are done at the same point.

The calculation for order k (including $k=1$) is done by a call *performStep*(k). The derivatives can be retrived by *getFoldDers*( ) or *getUnfoldDers*( ).

⟨ **KOrderStoch** class declaration 160 ⟩ ≡

```
class KOrderStoch {
protected:
  IntSequence nvs;
  PartitionY ypart;
  Journal &journal;
  UGSContainer _ug;
  FGSContainer _fg;
  UGSContainer _ugs;
  FGSContainer _fgs;
  UGSContainer _uG;
  FGSContainer _fG;
  const UGSContainer *_uh;
  const FGSContainer *_fh;
  UnfoldedZXContainer _uZstack;
  FoldedZXContainer _fZstack;
  UnfoldedGXContainer _uGstack;
  FoldedGXContainer _fGstack;
  const TensorContainer<FSSparseTensor> &f;
  MatrixAA matA;
public:
  KOrderStoch(const PartitionY &ypart, int nu, const TensorContainer<FSSparseTensor>
      &fcont, const FGSContainer &hh, Journal &jr);
  KOrderStoch(const PartitionY &ypart, int nu, const TensorContainer<FSSparseTensor>
      &fcont, const UGSContainer &hh, Journal &jr);
  ⟨ KOrderStoch::performStep templated code 163 ⟩;
  const FGSContainer &getFoldDers( ) const
  { return _fg; }
  const UGSContainer &getUnfoldDers( ) const
  { return _ug; }
protected:
  ⟨ KOrderStoch::faaDiBrunoZ templated code 161 ⟩;
  ⟨ KOrderStoch::faaDiBrunoG templated code 162 ⟩;
  ⟨ KOrderStoch convenience access methods 164 ⟩;
};
```

This code is cited in section 182.

This code is used in section 141.

161. This calculates a derivative of $f(G(y,u,\sigma),g(y,u,\sigma),y,u)$ of a given symmetry.

```
⟨ KOrderStoch::faaDiBrunoZ templated code 161 ⟩ ≡
  template⟨int t⟩ _Ttensor *faaDiBrunoZ(const Symmetry &sym) const
  {
    JournalRecordPair pa(journal);
    pa << "Faa Di Bruno ZX container for " << sym << endrec;
    _Ttensor *res = new _Ttensor(ypart.ny(), TensorDimens(sym, nvs));
    FaaDiBruno bruno(journal);
    bruno.calculate(Zstack<t>(), f, *res);
    return res;
  }
```

This code is used in section 160.

162. This calculates a derivative of $G(y,u,\sigma)=h(g^*(y,u,\sigma),\sigma)$ of a given symmetry.

```
⟨ KOrderStoch::faaDiBrunoG templated code 162 ⟩ ≡
  template⟨int t⟩ _Ttensor *faaDiBrunoG(const Symmetry &sym) const
  {
    JournalRecordPair pa(journal);
    pa << "Faa Di Bruno GX container for " << sym << endrec;
    TensorDimens tdims(sym, nvs);
    _Ttensor *res = new _Ttensor(ypart.nyss(), tdims);
    FaaDiBruno bruno(journal);
    bruno.calculate(Gstack<t>(), h<t>(), *res);
    return res;
  }
```

This code is used in section 160.

163. This retrives all g derivatives of a given dimension from implicit $f(h(g^*(y,u,\sigma),\sigma),g(y,u,\sigma),y,u)$. It suppose that all derivatives of smaller dimensions have been retrieved.

So, we go through all symmetries s, calculate G_s conditional on $g_s = 0$, insert the derivative to the G container, then calculate F_s conditional on $g_s = 0$. This is a righthand side. The left hand side is $matA \cdot g_s$. The g_s is retrieved as

$$g_s = -matA^{-1} \cdot RHS.$$

Finally we have to update G_s by calling $Gstack<t>().multAndAdd(1, h<t>(), *G\_sym)$.

⟨ **KOrderStoch**::*performStep* templated code 163 ⟩ ≡

```
template⟨int t⟩ void performStep(int order)
{
  int maxd = g<t>().getMaxDim();
  KORD_RAISE_IF(order − 1 ≠ maxd ∧ (order ≠ 1 ∨ maxd ≠ −1),
      "Wrong␣order␣for␣KOrderStoch::performStep");
  SymmetrySet ss(order, 4);
  for (symiterator si(ss); ¬si.isEnd(); ++si) {
    if ((*si)[2] ≡ 0) {
      JournalRecordPair pa(journal);
      pa ≪ "Recovering␣symmetry␣" ≪ *si ≪ endrec;
      _Ttensor *G_sym = faaDiBrunoG<t>(*si);
      G<t>().insert(G_sym);
      _Ttensor *g_sym = faaDiBrunoZ<t>(*si);
      g_sym¬mult(−1.0);
      matA.multInv(*g_sym);
      g<t>().insert(g_sym);
      gs<t>().insert(new _Ttensor(ypart.nstat, ypart.nys(), *g_sym));
      Gstack<t>().multAndAdd(1, h<t>(), *G_sym);
    }
  }
}
```

This code is used in section 160.

164.

⟨ **KOrderStoch** convenience access methods 164 ⟩ ≡

```
template⟨int t⟩ _Tg &g();
template⟨int t⟩ const _Tg &g() const;
template⟨int t⟩ _Tgs &gs();
template⟨int t⟩ const _Tgs &gs() const;
template⟨int t⟩ const _Tgss &h() const;
template⟨int t⟩ _TG &G();
template⟨int t⟩ const _TG &G() const;
template⟨int t⟩ _TZXstack &Zstack();
template⟨int t⟩ const _TZXstack &Zstack() const;
template⟨int t⟩ _TGXstack &Gstack();
template⟨int t⟩ const _TGXstack &Gstack() const;
```

This code is used in section 160.

165. End of `korder_stoch.h` file.

166. Start of `korder_stoch.cpp` file.

```
#include "korder_stoch.h"
⟨ MatrixAA constructor code 167 ⟩;
⟨ KOrderStoch folded constructor code 168 ⟩;
⟨ KOrderStoch unfolded constructor code 169 ⟩;
⟨ KOrderStoch convenience method specializations 170 ⟩;
```

167. Same as ⟨ **MatrixA** constructor code 131 ⟩, but the submatrix *gss_ys* is passed directly.

```
⟨ MatrixAA constructor code 167 ⟩ ≡
  MatrixAA::MatrixAA(const FSSparseTensor &f, const IntSequence &ss, const TwoDMatrix
          &gss_ys, const PartitionY &ypart)
  : PLUMatrix(ypart.ny()) {
    zeros();
    IntSequence c(1);
    c[0] = 1;
    FGSTensor f_y(f, ss, c, TensorDimens(ss, c));
    add(1.0, f_y);
    c[0] = 0;
    FGSTensor f_yss(f, ss, c, TensorDimens(ss, c));
    TwoDMatrix sub(*this, ypart.nstat, ypart.nys());
    sub.multAndAdd(f_yss, gss_ys);
    calcPLU();
  }
```

This code is used in section 166.

168.

```
⟨ KOrderStoch folded constructor code 168 ⟩ ≡
  KOrderStoch::KOrderStoch(const PartitionY &yp, int nu,
          const TensorContainer⟨FSSparseTensor⟩ &fcont, const FGSContainer &hh, Journal
          &jr)
  : nvs(4), ypart(yp), journal(jr),
  _ug(4), _fg(4), _ugs(4), _fgs(4), _uG(4), _fG(4),
  _uh(Λ), _fh(&hh),
  _uZstack(&_uG, ypart.nyss(), &_ug, ypart.ny(), ypart.nys(), nu),
  _fZstack(&_fG, ypart.nyss(), &_fg, ypart.ny(), ypart.nys(), nu),
  _uGstack(&_ugs, ypart.nys(), nu),
  _fGstack(&_fgs, ypart.nys(), nu),
  f(fcont),
  matA(*(fcont.get(Symmetry(1))), _uZstack.getStackSizes(), *(hh.get(Symmetry(1,0,0,0))), ypart) {
    nvs[0] = ypart.nys();
    nvs[1] = nu;
    nvs[2] = nu;
    nvs[3] = 1;
  }
```

This code is used in section 166.

169.

⟨ **KOrderStoch** unfolded constructor code 169 ⟩ ≡

```
KOrderStoch::KOrderStoch(const PartitionY &yp, int nu,
        const TensorContainer<FSSparseTensor> &fcont, const UGSContainer &hh, Journal
        &jr)
: nvs(4), ypart(yp), journal(jr),
_ug(4), _fg(4), _ugs(4), _fgs(4), _uG(4), _fG(4),
_uh(&hh), _fh(Λ),
_uZstack(&_uG, ypart.nyss( ), &_ug, ypart.ny( ), ypart.nys( ), nu),
_fZstack(&_fG, ypart.nyss( ), &_fg, ypart.ny( ), ypart.nys( ), nu),
_uGstack(&_ugs, ypart.nys( ), nu),
_fGstack(&_fgs, ypart.nys( ), nu),
f(fcont),
matA(*(fcont.get(Symmetry(1))), _uZstack.getStackSizes( ), *(hh.get(Symmetry(1, 0, 0, 0))), ypart) {
  nvs[0] = ypart.nys( );
  nvs[1] = nu;
  nvs[2] = nu;
  nvs[3] = 1;
}
```

This code is used in section 166.

170.

```
⟨ KOrderStoch convenience method specializations 170 ⟩ ≡
  template⟨⟩ ctraits⟨KOrder::unfold⟩::Tg &KOrderStoch::g < KOrder::unfold > ()
  { return _ug; }
  template⟨⟩ const ctraits⟨KOrder::unfold⟩::Tg &KOrderStoch::g < KOrder::unfold > () const
  { return _ug; }
  template⟨⟩ ctraits⟨KOrder::fold⟩::Tg &KOrderStoch::g < KOrder::fold > ()
  { return _fg; }
  template⟨⟩ const ctraits⟨KOrder::fold⟩::Tg &KOrderStoch::g < KOrder::fold > () const
  { return _fg; }
  template⟨⟩ ctraits⟨KOrder::unfold⟩::Tgs &KOrderStoch::gs < KOrder::unfold > ()
  { return _ugs; }
  template⟨⟩ const ctraits⟨KOrder::unfold⟩::Tgs &KOrderStoch::gs < KOrder::unfold > ()
      const
  { return _ugs; }
  template⟨⟩ ctraits⟨KOrder::fold⟩::Tgs &KOrderStoch::gs < KOrder::fold > ()
  { return _fgs; }
  template⟨⟩ const ctraits⟨KOrder::fold⟩::Tgs &KOrderStoch::gs < KOrder::fold > () const
  { return _fgs; }
  template⟨⟩ const ctraits⟨KOrder::unfold⟩::Tgss &KOrderStoch::h < KOrder::unfold > ()
      const
  { return *_uh; }
  template⟨⟩ const ctraits⟨KOrder::fold⟩::Tgss &KOrderStoch::h < KOrder::fold > () const
  { return *_fh; }
  template⟨⟩ ctraits⟨KOrder::unfold⟩::TG &KOrderStoch::G < KOrder::unfold > ()
  { return _uG; }
  template⟨⟩ const ctraits⟨KOrder::unfold⟩::TG &KOrderStoch::G < KOrder::unfold > ()
      const
  { return _uG; }
  template⟨⟩ ctraits⟨KOrder::fold⟩::TG &KOrderStoch::G < KOrder::fold > ()
  { return _fG; }
  template⟨⟩ const ctraits⟨KOrder::fold⟩::TG &KOrderStoch::G < KOrder::fold > () const
  { return _fG; }
  template⟨⟩ ctraits⟨KOrder::unfold⟩::TZXstack &KOrderStoch::Zstack < KOrder::unfold > ()
  { return _uZstack; }
  template⟨⟩ const ctraits⟨KOrder::unfold⟩::TZXstack &KOrderStoch::Zstack <
          KOrder::unfold > () const
  { return _uZstack; }
  template⟨⟩ ctraits⟨KOrder::fold⟩::TZXstack &KOrderStoch::Zstack < KOrder::fold > ()
  { return _fZstack; }
  template⟨⟩ const ctraits⟨KOrder::fold⟩::TZXstack &KOrderStoch::Zstack < KOrder::fold > ()
      const
  { return _fZstack; }
  template⟨⟩ ctraits⟨KOrder::unfold⟩::TGXstack &KOrderStoch::Gstack < KOrder::unfold > ()
  { return _uGstack; }
  template⟨⟩ const ctraits⟨KOrder::unfold⟩::TGXstack &KOrderStoch::Gstack <
          KOrder::unfold > () const
  { return _uGstack; }
  template⟨⟩ ctraits⟨KOrder::fold⟩::TGXstack &KOrderStoch::Gstack < KOrder::fold > ()
  { return _fGstack; }
  template⟨⟩ const ctraits⟨KOrder::fold⟩::TGXstack &KOrderStoch::Gstack < KOrder::fold > ()
      const
```

{ **return** *_fGstack*; }

This code is used in section 166.

171. End of `korder_stoch.cpp` file.

172. Putting all together.

173. Dynamic model abstraction. Start of `dynamic_model.h` file.
This file only defines a generic interface to an SDGE model. The model takes the form:

$$E_t\left[f(g^{**}(g^*(y,u_t),u_{t+1}),g(y,u),y,u_t)\right]=0$$

The interface is defined via pure virtual class **DynamicModel**.

```
#ifndef DYNAMIC_MODEL_H
#define DYNAMIC_MODEL_H
#include "t_container.h"
#include "sparse_tensor.h"
#include "Vector.h"
  ⟨ NameList class declaration 174 ⟩;
  ⟨ DynamicModel class declaration 175 ⟩;
#endif
```

174. The class is a virtual pure class which provides an access to names of the variables.

```
⟨ NameList class declaration 174 ⟩ ≡
  class NameList {
  public:
    virtual ~NameList( )
    { }
    virtual int getNum( ) const = 0;
    virtual const char *getName(int i) const = 0;
    void print( ) const;
    void writeMat(mat_t *fd, const char *vname) const;
    void writeMatIndices(mat_t *fd, const char *prefix) const;
  };
```

This code is used in section 173.

175. This is the interface to an information on a generic SDGE model. It is sufficient for calculations of policy rule Taylor approximations at some (not necessarily deterministic) steady state.

We need to know a partitioning of endogenous variables y. We suppose that y is partitioned as

$$y = \begin{bmatrix} \text{static} \\ \text{pred} \\ \text{both} \\ \text{forward} \end{bmatrix}$$

of which we define

$$y^* = \begin{bmatrix} \text{pred} \\ \text{both} \end{bmatrix} \quad y^{**} = \begin{bmatrix} \text{both} \\ \text{forward} \end{bmatrix}$$

where "static" are meant those variables, which appear only at time t; "pred" are meant those variables, which appear only at t and $t-1$; "both" are meant those variables, which appear at least at $t-1$ and $t+1$; and "forward" are meant those variables, which appear only at t and $t+1$. This partitioning is given by methods *nstat*( ), *npred*( ), *nboth*( ), and *nforw*( ). The number of equations *numeq*( ) must be the same as a number of endogenous variables.

In order to complete description, we need to know a number of exogenous variables, which is a size of u, hence *nexog*( ) method.

The model contains an information about names of variables, the variance-covariance matrix of the shocks, the derivatives of equations of f at some steady state, and the steady state. These can be retrieved by the corresponding methods.

The derivatives of the system are calculated with respect to stacked variables, the stack looks as:

$$\begin{bmatrix} y^{**}_{t+1} \\ y_t \\ y^*_{t-1} \\ u_t \end{bmatrix}.$$

There are only three operations. The first *solveDeterministicSteady*( ) solves the deterministic steady steate which can be retrieved by *getSteady*( ) later. The method *evaluateSystem* calculates $f(y^{**}, y, y^*, u)$, where y and u are passed, or $f(y^{**}_{t+1}, y_t, y^*_{t-1}, u)$, where y^{**}_{t+1}, y_t, y^*_{t-1}, u are passed. Finally, the method *calcDerivativesAtSteady*( ) calculates derivatives of f at the current steady state, and zero shocks. The derivatives can be retrieved with *getModelDerivatives*( ). All the derivatives are done up to a given order in the model, which can be retrieved by *order*( ).

The model initialization is done in a constructor of the implementing class. The constructor usually calls a parser, which parses a given file (usually a text file), and retrieves all necessary information about the model, inluding variables, partitioning, variance-covariance matrix, information helpful for calculation of the deterministic steady state, and so on.

⟨ **DynamicModel** class declaration 175 ⟩ ≡

```
class DynamicModel {
public:
  virtual DynamicModel *clone( ) const = 0;
  virtual ~DynamicModel( )
  { }
  virtual int nstat( ) const = 0;
  virtual int nboth( ) const = 0;
  virtual int npred( ) const = 0;
  virtual int nforw( ) const = 0;
  virtual int nexog( ) const = 0;
  virtual int order( ) const = 0;
  int numeq( ) const
```

```
    { return nstat() + nboth() + npred() + nforw(); }
    virtual const NameList &getAllEndoNames() const = 0;
    virtual const NameList &getStateNames() const = 0;
    virtual const NameList &getExogNames() const = 0;
    virtual const TwoDMatrix &getVcov() const = 0;
    virtual const TensorContainer<FSSparseTensor> &getModelDerivatives() const = 0;
    virtual const Vector &getSteady() const = 0;
    virtual Vector &getSteady() = 0;
    virtual void solveDeterministicSteady() = 0;
    virtual void evaluateSystem(Vector &out, const Vector &yy, const Vector &xx) = 0;
    virtual void evaluateSystem(Vector &out, const Vector &yym, const Vector &yy, const Vector
        &yyp, const Vector &xx) = 0;
    virtual void calcDerivativesAtSteady() = 0;
  };
```

This code is used in section 173.

176. End of `dynamic_model.h` file.

177. Start of `dynamic_model.cpp` file.

```
#include "dynamic_model.h"
  ⟨ NameList::print code 178 ⟩;
  ⟨ NameList::writeMat code 179 ⟩;
  ⟨ NameList::writeMatIndices code 180 ⟩;
```

178.

```
⟨ NameList::print code 178 ⟩ ≡
  void NameList::print() const
  {
    for (int i = 0; i < getNum(); i++) printf("%s\n", getName(i));
  }
```

This code is used in section 177.

179.

```
⟨ NameList::writeMat code 179 ⟩ ≡
  void NameList::writeMat(mat_t *fd, const char *vname) const
  {
    int maxlen = 0;
    for (int i = 0; i < getNum(); i++)
      if (maxlen < (int) strlen(getName(i))) maxlen = (int) strlen(getName(i));
    if (maxlen ≡ 0) return;
    char *m = new char[getNum() * maxlen];
    for (int i = 0; i < getNum(); i++)
      for (int j = 0; j < maxlen; j++)
        if (j < (int) strlen(getName(i))) m[j * getNum() + i] = getName(i)[j];
        else m[j * getNum() + i] = ' ';
#if MATIO_MAJOR_VERSION > 1 ∨ (MATIO_MAJOR_VERSION ≡ 1 ∧ MATIO_MINOR_VERSION ≥ 5)
    size_t dims[2];
    const matio_compression compression = MAT_COMPRESSION_NONE;
#else
    int dims[2];
    const int compression = COMPRESSION_NONE;
#endif
    dims[0] = getNum();
    dims[1] = maxlen;
    matvar_t *v = Mat_VarCreate(vname, MAT_C_CHAR, MAT_T_UINT8, 2, dims, m, 0);
    Mat_VarWrite(fd, v, compression);
    Mat_VarFree(v);
    delete[] m;
  }
```

This code is used in section 177.

180.

```
⟨ NameList::writeMatIndices code 180 ⟩ ≡
  void NameList::writeMatIndices(mat_t *fd, const char *prefix) const
  {
    char tmp[100];
    TwoDMatrix aux(1, 1);
    for (int i = 0; i < getNum(); i++) {
      sprintf(tmp, "%s_i_%s", prefix, getName(i));
      aux.get(0, 0) = i + 1;
      aux.writeMat(fd, tmp);
    }
  }
```

This code is used in section 177.

181. End of `dynamic_model.cpp` file.

182. Approximating model solution. Start of `approximation.h` file.

The class **Approximation** in this file is a main interface to the algorithms calculating approximations to the decision rule about deterministic and stochastic steady states.

The approximation about a deterministic steady state is solved by classes ⟨ **FirstOrder** class declaration 76 ⟩ and ⟨ **KOrder** class declaration 102 ⟩. The approximation about the stochastic steady state is solved by class ⟨ **KOrderStoch** class declaration 160 ⟩ together with a method of **Approximation** class ⟨ **Approximation** :: *walkStochSteady* code 194 ⟩.

The approximation about the stochastic steady state is done with explicit expression of forward derivatives of g^{**}. More formally, we have to solve the decision rule g from the implicit system:

$$E_t(f(g^{**}(g^*(y^*,u_t,\sigma),u_{t+1},\sigma),g(y^*,u_t,\sigma),y_t,u_t))=0$$

The term within the expectations can be Taylor expanded, and the expectation can be driven into the formula. However, when doing this at $\sigma\neq 0$, the term g^{**} at $\sigma\neq 0$ is dependent on u_{t+1} and thus the integral of its approximation includes all derivatives wrt. u of g^{**}. Note that for $\sigma=0$, the derivatives of g^{**} in this context are constant. This is the main difference between the approximation at deterministic steady ($\sigma=0$), and stochastic steady ($\sigma\neq 0$). This means that k-order derivative of the above equation at $\sigma\neq 0$ depends of all derivatives of g^*∗ (including those with order greater than k).

The explicit expression of the forward g^{**} means that the derivatives of g are not solved simultaneously, but that the forward derivatives of g^{**} are calculated as an extrapolation based on the approximation at lower σ. This is exactly what does the ⟨ **Approximation** :: *walkStochSteady* code 194 ⟩. It starts at the deterministic steady state, and in a few steps it adds to σ explicitly expressing forward g^{**} from a previous step.

Further details on the both solution methods are given in (todo: put references here when they exist).

Very important note: all classes here used for calculation of decision rule approximation are folded. For the time being, it seems that faa Di Bruno formula is quicker for folded tensors, and that is why we stick to folded tensors here. However, when the calcs are done, we calculate also its unfolded versions, to be available for simulations and so on.

```
#ifndef APPROXIMATION_H
#define APPROXIMATION_H
#include "dynamic_model.h"
#include "decision_rule.h"
#include "korder.h"
#include "journal.h"
  ⟨ ZAuxContainer class declaration 183 ⟩;
  ⟨ Approximation class declaration 184 ⟩;
#endif
```

183. This class is used to calculate derivatives by faa Di Bruno of the

$$f(g^{**}(g^*(y^*,u,\sigma),u',\sigma),g(y^*,u,\sigma),y^*,u)$$

with respect u'. In order to keep it as simple as possible, the class represents an equivalent (with respect to u') container for $f(g^{**}(y^*,u',\sigma),0,0,0)$. The class is used only for evaluation of approximation error in **Approximation** class, which is calculated in **Approximation**::*calcStochShift* method.

Since it is a folded version, we inherit from **StackContainer**⟨**FGSTensor**⟩ and **FoldedStackContainer**. To construct it, we need only the g^{**} container and size of stacks.

⟨ **ZAuxContainer** class declaration 183 ⟩ ≡

```
class ZAuxContainer : public StackContainer<FGSTensor>, public FoldedStackContainer {
public:
  typedef StackContainer<FGSTensor>::_Ctype _Ctype;
  typedef StackContainer<FGSTensor>::itype itype;
  ZAuxContainer(const _Ctype *gss, int ngss, int ng, int ny, int nu);
  itype getType(int i, const Symmetry &s) const;
};
```

This code is cited in section 202.

This code is used in section 182.

184. This class provides an interface to approximation algorithms. The core method is *walkStochSteady* which calculates the approximation about stochastic steady state in a given number of steps. The number is given as a parameter *ns* of the constructor. If the number is equal to zero, the resulted approximation is about the deterministic steady state.

An object is constructed from the **DynamicModel**, and the number of steps *ns*. Also, we pass a reference to journal. That's all. The result of the core method *walkStochSteady* is a decision rule *dr* and a matrix *ss* whose columns are steady states for increasing σ during the walk. Both can be retrived by public methods. The first column of the matrix is the deterministic steady state, the last is the stochastic steady state for the full size shocks.

The method *walkStochSteady* calls the following methods: *approxAtSteady* calculates an initial approximation about the deterministic steady, *saveRuleDerivs* saves derivatives of a rule for the following step in *rule_ders* and *rule_ders_ss* (see ⟨ **Approximation** :: *saveRuleDerivs* code 201 ⟩ for their description), *check* reports an error of the current approximation and *calcStochShift* (called from *check*) calculates a shift of the system equations due to uncertainity.

dr_centralize is a new option. dynare++ was automatically expressing results around the fixed point instead of the deterministic steady state. dr_centralize controls this behavior.

⟨ **Approximation** class declaration 184 ⟩ ≡

```
class Approximation {
  DynamicModel &model;
  Journal &journal;
  FGSContainer *rule_ders;
  FGSContainer *rule_ders_ss;
  FoldDecisionRule *fdr;
  UnfoldDecisionRule *udr;
  const PartitionY ypart;
  const FNormalMoments mom;
  IntSequence nvs;
  int steps;
  bool dr_centralize;
  double qz_criterium;
  TwoDMatrix ss;
public:
  Approximation(DynamicModel &m, Journal &j, int ns, bool dr_centr, double qz_crit);
  virtual ~Approximation();
  const FoldDecisionRule &getFoldDecisionRule() const;
  const UnfoldDecisionRule &getUnfoldDecisionRule() const;
  const TwoDMatrix &getSS() const
  { return ss; }
  const DynamicModel &getModel() const
  { return model; }
  void walkStochSteady();
  TwoDMatrix *calcYCov() const;
  const FGSContainer *get_rule_ders() const
  { return rule_ders; }
  const FGSContainer *get_rule_ders_ss() const
  { return rule_ders; }
protected:
  void approxAtSteady();
  void calcStochShift(Vector &out, double at_sigma) const;
  void saveRuleDerivs(const FGSContainer &g);
```

```
    void check(double at_sigma) const;
  };
```
This code is used in section 182.

185. End of `approximation.h` file.

186. Start of `approximation.cpp` file.

```
#include "kord_exception.h"
#include "approximation.h"
#include "first_order.h"
#include "korder_stoch.h"
  ⟨ ZAuxContainer constructor code 187 ⟩;
  ⟨ ZAuxContainer::getType code 188 ⟩;
  ⟨ Approximation constructor code 189 ⟩;
  ⟨ Approximation destructor code 190 ⟩;
  ⟨ Approximation::getFoldDecisionRule code 191 ⟩;
  ⟨ Approximation::getUnfoldDecisionRule code 192 ⟩;
  ⟨ Approximation::approxAtSteady code 193 ⟩;
  ⟨ Approximation::walkStochSteady code 194 ⟩;
  ⟨ Approximation::saveRuleDerivs code 201 ⟩;
  ⟨ Approximation::calcStochShift code 202 ⟩;
  ⟨ Approximation::check code 205 ⟩;
  ⟨ Approximation::calcYCov code 206 ⟩;
```

187.

```
⟨ ZAuxContainer constructor code 187 ⟩ ≡
  ZAuxContainer::ZAuxContainer(const _Ctype *gss, int ngss, int ng, int ny, int nu)
  : StackContainer⟨FGSTensor⟩(4, 1) {
    stack_sizes[0] = ngss;
    stack_sizes[1] = ng;
    stack_sizes[2] = ny;
    stack_sizes[3] = nu;
    conts[0] = gss;
    calculateOffsets( );
  }
```
This code is used in section 186.

188. The *getType* method corresponds to $f(g^{**}(y^*, u', \sigma), 0, 0, 0)$. For the first argument we return *matrix*, for other three we return *zero*.

```
⟨ ZAuxContainer::getType code 188 ⟩ ≡
  ZAuxContainer::itype ZAuxContainer::getType(int i, const Symmetry &s) const
  {
    if (i ≡ 0)
      if (s[2] > 0) return zero;
      else return matrix;
    return zero;
  }
```
This code is used in section 186.

189.

⟨ **Approximation** constructor code 189 ⟩ ≡
```
Approximation::Approximation(DynamicModel &m, Journal &j, int ns, bool dr_centr, double
        qz_crit)
  : model(m), journal(j), rule_ders(Λ), rule_ders_ss(Λ),
      fdr(Λ), udr(Λ), ypart(model.nstat(), model.npred(), model.nboth(), model.nforw()),
      mom(UNormalMoments(model.order(), model.getVcov())), nvs(4), steps(ns),
      dr_centralize(dr_centr), qz_criterium(qz_crit), ss(ypart.ny(), steps + 1) {
  nvs[0] = ypart.nys();
  nvs[1] = model.nexog();
  nvs[2] = model.nexog();
  nvs[3] = 1;
  ss.nans();
}
```
This code is used in section 186.

190.

⟨ **Approximation** destructor code 190 ⟩ ≡
```
Approximation::~Approximation()
{
  if (rule_ders_ss) delete rule_ders_ss;
  if (rule_ders) delete rule_ders;
  if (fdr) delete fdr;
  if (udr) delete udr;
}
```
This code is used in section 186.

191. This just returns *fdr* with a check that it is created.

⟨ **Approximation**::*getFoldDecisionRule* code 191 ⟩ ≡
```
const FoldDecisionRule &Approximation::getFoldDecisionRule() const
{
  KORD_RAISE_IF(fdr ≡ Λ,
      "Folded␣decision␣rule␣has␣not␣been␣created␣in␣Approximation::getFoldDecisionRule");
  return *fdr;
}
```
This code is used in section 186.

192. This just returns *udr* with a check that it is created.

⟨ **Approximation**::*getUnfoldDecisionRule* code 192 ⟩ ≡
```
const UnfoldDecisionRule &Approximation::getUnfoldDecisionRule() const
{
  KORD_RAISE_IF(udr ≡ Λ, "Unfolded␣decision␣rule␣has␣not␣been␣created␣in␣Approximatio\
      n::getUnfoldDecisionRule");
  return *udr;
}
```
This code is used in section 186.

193. This methods assumes that the deterministic steady state is *model.getSteady*( ). It makes an approximation about it and stores the derivatives to *rule_ders* and *rule_ders_ss*. Also it runs a *check* for $\sigma = 0$.

```
⟨ Approximation::approxAtSteady code 193 ⟩ ≡
  void Approximation::approxAtSteady( )
  {
    model.calcDerivativesAtSteady( );
    FirstOrder fo(model.nstat( ), model.npred( ), model.nboth( ), model.nforw( ), model.nexog( ),
        *(model.getModelDerivatives( ).get(Symmetry(1))), journal, qz_criterium);
    KORD_RAISE_IF_X(¬fo.isStable( ), "The␣model␣is␣not␣Blanchard-Kahn␣stable",
        KORD_MD_NOT_STABLE);
    if (model.order( ) ≥ 2) {
      KOrder korder(model.nstat( ), model.npred( ), model.nboth( ), model.nforw( ),
          model.getModelDerivatives( ), fo.getGy( ), fo.getGu( ), model.getVcov( ), journal);
      korder.switchToFolded( );
      for (int k = 2; k ≤ model.order( ); k++) korder.performStep < KOrder::fold > (k);
      saveRuleDerivs(korder.getFoldDers( ));
    }
    else {
      FirstOrderDerivs⟨KOrder::fold⟩ fo_ders(fo);
      saveRuleDerivs(fo_ders);
    }
    check(0.0);
  }
```

This code is used in section 186.

194. This is the core routine of **Approximation** class.

First we solve for the approximation about the deterministic steady state. Then we perform *steps* cycles toward the stochastic steady state. Each cycle moves the size of shocks by $dsigma = 1.0/steps$. At the end of a cycle, we have *rule_ders* being the derivatives at stochastic steady state for $\sigma = sigma\_so\_far + dsigma$ and $model.getSteady()$ being the steady state.

If the number of *steps* is zero, the decision rule *dr* at the bottom is created from derivatives about deterministic steady state, with size of $\sigma = 1$. Otherwise, the *dr* is created from the approximation about stochastic steady state with $\sigma = 0$.

Within each cycle, we first make a backup of the last steady (from initialization or from a previous cycle), then we calculate the fix point of the last rule with $\sigma = dsigma$. This becomes a new steady state at the $\sigma = sigma\_so\_far + dsigma$. We calculate expectations of $g^{**}(y, \sigma\eta_{t+1}, \sigma$ expressed as a Taylor expansion around the new σ and the new steady state. Then we solve for the decision rule with explicit g^{**} at $t+1$ and save the rule.

After we reached $\sigma = 1$, the decision rule is formed.

The biproduct of this method is the matrix *ss*, whose columns are steady states for subsequent σs. The first column is the deterministic steady state, the last column is the stochastic steady state for a full size of shocks ($\sigma = 1$). There are $steps + 1$ columns.

```
⟨ Approximation::walkStochSteady code 194 ⟩ ≡
  void Approximation::walkStochSteady()
  {
    ⟨ initial approximation at deterministic steady 195 ⟩;
    double sigma_so_far = 0.0;
    double dsigma = (steps == 0) ? 0.0 : 1.0/steps;
    for (int i = 1; i <= steps; i++) {
      JournalRecordPair pa(journal);
      pa << "Approximation about stochastic steady for sigma=" << sigma_so_far+dsigma << endrec;
      Vector last_steady((const Vector &) model.getSteady());
      ⟨ calculate fix-point of the last rule for dsigma 196 ⟩;
      ⟨ calculate hh as expectations of the last g** 197 ⟩;
      ⟨ form KOrderStoch, solve and save 198 ⟩;
      check(sigma_so_far + dsigma);
      sigma_so_far += dsigma;
    }
    ⟨ construct the resulting decision rules 199 ⟩;
  }
```

This code is cited in section 182.

This code is used in section 186.

195. Here we solve for the deterministic steady state, calculate approximation at the deterministic steady and save the steady state to *ss*.

```
⟨ initial approximation at deterministic steady 195 ⟩ ≡
  model.solveDeterministicSteady();
  approxAtSteady();
  Vector steady0(ss, 0);
  steady0 = (const Vector &) model.getSteady();
```

This code is used in section 194.

196. We form the **DRFixPoint** object from the last rule with $\sigma = dsigma$. Then we save the steady state to ss. The new steady is also put to $model.getSteady()$.

```
⟨ calculate fix-point of the last rule for dsigma 196 ⟩ ≡
  DRFixPoint⟨KOrder::fold⟩ fp(*rule_ders, ypart, model.getSteady(), dsigma);
  bool converged = fp.calcFixPoint(DecisionRule::horner, model.getSteady());
  JournalRecord rec(journal);
  rec << "Fix point calcs: iter=" << fp.getNumIter() << ", newton_iter="
      << fp.getNewtonTotalIter() << ", last_newton_iter=" << fp.getNewtonLastIter() << ".";
  if (converged) rec << " Converged." << endrec;
  else {
    rec << " Not converged!!" << endrec;
    KORD_RAISE_X("Fix point calculation not converged", KORD_FP_NOT_CONV);
  }
  Vector steadyi(ss, i);
  steadyi = (const Vector &) model.getSteady();
```

This code is used in section 194.

197. We form the steady state shift dy, which is the new steady state minus the old steady state. Then we create **StochForwardDerivs** object, which calculates the derivatives of g^{**} expectations at new sigma and new steady.

```
⟨ calculate hh as expectations of the last g** 197 ⟩ ≡
  Vector dy((const Vector &) model.getSteady());
  dy.add(-1.0, last_steady);
  StochForwardDerivs⟨KOrder::fold⟩ hh(ypart, model.nexog(), *rule_ders_ss, mom, dy, dsigma,
      sigma_so_far);
  JournalRecord rec1(journal);
  rec1 << "Calculation of g** expectations done" << endrec;
```

This code is used in section 194.

198. We calculate derivatives of the model at the new steady, form **KOrderStoch** object and solve, and save the rule.

```
⟨ form KOrderStoch, solve and save 198 ⟩ ≡
  model.calcDerivativesAtSteady();
  KOrderStoch korder_stoch(ypart, model.nexog(), model.getModelDerivatives(), hh, journal);
  for (int d = 1; d ≤ model.order(); d++) {
    korder_stoch.performStep < KOrder::fold > (d);
  }
  saveRuleDerivs(korder_stoch.getFoldDers());
```

This code is used in section 194.

199.

```
⟨ construct the resulting decision rules 199 ⟩ ≡
  if (fdr) {
    delete fdr;
    fdr = Λ;
  }
  if (udr) {
    delete udr;
    udr = Λ;
  }
  fdr = new FoldDecisionRule(*rule_ders, ypart, model.nexog(), model.getSteady(), 1.0 − sigma_so_far);
  if (steps ≡ 0 ∧ dr_centralize) {
    ⟨ centralize decision rule for zero steps 200 ⟩;
  }
```

This code is used in section 194.

200.

```
⟨ centralize decision rule for zero steps 200 ⟩ ≡
  DRFixPoint⟨KOrder::fold⟩ fp(*rule_ders, ypart, model.getSteady(), 1.0);
  bool converged = fp.calcFixPoint(DecisionRule::horner, model.getSteady());
  JournalRecord rec(journal);
  rec ≪ "Fix␣point␣calcs:␣iter=" ≪ fp.getNumIter() ≪ ",␣newton_iter=" ≪
      fp.getNewtonTotalIter() ≪ ",␣last_newton_iter=" ≪ fp.getNewtonLastIter() ≪ ".";
  if (converged) rec ≪ "␣Converged." ≪ endrec;
  else {
    rec ≪ "␣Not␣converged!!" ≪ endrec;
    KORD_RAISE_X("Fix␣point␣calculation␣not␣converged", KORD_FP_NOT_CONV);
  }
  {
    JournalRecordPair recp(journal);
    recp ≪ "Centralizing␣about␣fix-point." ≪ endrec;
    FoldDecisionRule *dr_backup = fdr;
    fdr = new FoldDecisionRule(*dr_backup, model.getSteady());
    delete dr_backup;
  }
```

This code is used in section 199.

201. Here we simply make a new hardcopy of the given rule *rule_ders*, and make a new container of in-place subtensors of the derivatives corresponding to forward looking variables. The given container comes from a temporary object and will be destroyed.

```
⟨ Approximation::saveRuleDerivs code 201 ⟩ ≡
  void Approximation::saveRuleDerivs(const FGSContainer &g)
  {
    if (rule_ders) {
      delete rule_ders;
      delete rule_ders_ss;
    }
    rule_ders = new FGSContainer(g);
    rule_ders_ss = new FGSContainer(4);
    for (FGSContainer::iterator run = (*rule_ders).begin(); run ≠ (*rule_ders).end(); ++run) {
      FGSTensor *ten = new FGSTensor(ypart.nstat + ypart.npred, ypart.nyss(), *((*run).second));
      rule_ders_ss→insert(ten);
    }
  }
```

This code is cited in section 184.

This code is used in section 186.

202. This method calculates a shift of the system equations due to integrating shocks at a given σ and current steady state. More precisely, if

$$F(y,u,u',\sigma) = f(g^{**}(g^{*}(y,u,\sigma),u',\sigma),g(y,u,\sigma),y,u)$$

then the method returns a vector

$$\sum_{d=1}\frac{1}{d!}\sigma^d\left[F_{u'^d}\right]_{\alpha_1\ldots\alpha_d}\Sigma^{\alpha_1\ldots\alpha_d}$$

For a calculation of $[F_{u'^d}]$ we use ⟨ **ZAuxContainer** class declaration 183 ⟩, so we create its object. In each cycle we calculate $[F_{u'^d}]$, and then multiply with the shocks, and add the $\frac{\sigma^d}{d!}$ multiple to the result.

```
⟨ Approximation::calcStochShift code 202 ⟩ ≡
  void Approximation::calcStochShift(Vector &out, double at_sigma) const
  {
    KORD_RAISE_IF(out.length() ≠ ypart.ny(),
        "Wrong length of output vector for Approximation::calcStochShift");
    out.zeros();
    ZAuxContainer zaux(rule_ders_ss, ypart.nyss(), ypart.ny(), ypart.nys(), model.nexog());
    int dfac = 1;
    for (int d = 1; d ≤ rule_ders→getMaxDim(); d++, dfac *= d) {
      if (KOrder::is_even(d)) {
        Symmetry sym(0, d, 0, 0);
        ⟨ calculate F_{u'^d} via ZAuxContainer 203 ⟩;
        ⟨ multiply with shocks and add to result 204 ⟩;
      }
    }
  }
```

This code is used in section 186.

203.

⟨ calculate $F_{u'^d}$ via **ZAuxContainer** 203 ⟩ ≡

```
FGSTensor *ten = new FGSTensor(ypart.ny(), TensorDimens(sym, nvs));
ten->zeros();
for (int l = 1; l ≤ d; l++) {
  const FSSparseTensor *f = model.getModelDerivatives().get(Symmetry(l));
  zaux.multAndAdd(*f, *ten);
}
```

This code is used in section 202.

204.

⟨ multiply with shocks and add to result 204 ⟩ ≡

```
FGSTensor *tmp = new FGSTensor(ypart.ny(), TensorDimens(Symmetry(0,0,0,0), nvs));
tmp->zeros();
ten->contractAndAdd(1, *tmp, *(mom.get(Symmetry(d))));
out.add(pow(at_sigma, d)/dfac, tmp->getData());
delete ten;
delete tmp;
```

This code is used in section 202.

205. This method calculates and reports

$$f(\bar{y}) + \sum_{d=1} \frac{1}{d!} \sigma^d \left[F_{u'^d}\right]_{\alpha_1 \ldots \alpha_d} \Sigma^{\alpha_1 \ldots \alpha_d}$$

at $\bar{y}$, zero shocks and σ. This number should be zero.

We evaluate the error both at a given σ and $\sigma = 1.0$.

⟨ **Approximation**::*check* code 205 ⟩ ≡

```
void Approximation::check(double at_sigma) const
{
  Vector stoch_shift(ypart.ny());
  Vector system_resid(ypart.ny());
  Vector xx(model.nexog());
  xx.zeros();
  model.evaluateSystem(system_resid, model.getSteady(), xx);
  calcStochShift(stoch_shift, at_sigma);
  stoch_shift.add(1.0, system_resid);
  JournalRecord rec1(journal);
  rec1 << "Error of current approximation for shocks at sigma " << at_sigma << " is "
       << stoch_shift.getMax() << endrec;
  calcStochShift(stoch_shift, 1.0);
  stoch_shift.add(1.0, system_resid);
  JournalRecord rec2(journal);
  rec2 << "Error of current approximation for full shocks is " << stoch_shift.getMax()
       << endrec;
}
```

This code is used in section 186.

206. The method returns unconditional variance of endogenous variables based on the first order. The first order approximation looks like

$$\hat{y}_t = g_{y^*}\hat{y}^*_{t-1} + g_u u_t$$

where $\hat{y}$ denotes a deviation from the steady state. It can be written as

$$\hat{y}_t = [0\, g_{y^*}\, 0]\, \hat{y}_{t-1} + g_u u_t$$

which yields unconditional covariance V for which

$$V = GVG^T + g_u \Sigma g_u^T,$$

where $G = [0\, g_{y^*}\, 0]$ and Σ is the covariance of the shocks.

For solving this Lyapunov equation we use the Sylvester module, which solves equation of the type

$$AX + BX(C \otimes \cdots \otimes C) = D$$

So we invoke the Sylvester solver for the first dimension with $A = I$, $B = -G$, $C = G^T$ and $D = g_u \Sigma g_u^T$.

⟨ **Approximation**::*calcYCov* code 206 ⟩ ≡

```
TwoDMatrix *Approximation::calcYCov() const
{
  const TwoDMatrix &gy = *(rule_ders→get(Symmetry(1,0,0,0)));
  const TwoDMatrix &gu = *(rule_ders→get(Symmetry(0,1,0,0)));
  TwoDMatrix G(model.numeq(), model.numeq());
  G.zeros();
  G.place(gy, 0, model.nstat());
  TwoDMatrix B((const TwoDMatrix &) G);
  B.mult(-1.0);
  TwoDMatrix C(G, "transpose");
  TwoDMatrix A(model.numeq(), model.numeq());
  A.zeros();
  for (int i = 0; i < model.numeq(); i++) A.get(i,i) = 1.0;
  TwoDMatrix guSigma(gu, model.getVcov());
  TwoDMatrix guTrans(gu, "transpose");
  TwoDMatrix *X = new TwoDMatrix(guSigma, guTrans);
  GeneralSylvester gs(1, model.numeq(), model.numeq(), 0, A.base(), B.base(), C.base(), X→base());
  gs.solve();
  return X;
}
```

This code is used in section 186.

207. End of `approximation.cpp` file.

208. Decision rule and simulation. Start of `decision_rule.h` file.

The main purpose of this file is a decision rule representation which can run a simulation. So we define an interface for classes providing realizations of random shocks, and define the class **DecisionRule**. The latter basically takes tensor container of derivatives of policy rules, and adds them up with respect to σ. The class allows to specify the σ different from 1.

In addition, we provide classes for running simulations and storing the results, calculating some statistics and generating IRF. The class **DRFixPoint** allows for calculation of the fix point of a given decision rule.

```
#ifndef DECISION_RULE_H
#define DECISION_RULE_H
#include <matio.h>
#include "kord_exception.h"
#include "korder.h"
#include "normal_conjugate.h"
#include "mersenne_twister.h"
  ⟨ ShockRealization class declaration 209 ⟩;
  ⟨ DecisionRule class declaration 210 ⟩;
  ⟨ DecisionRuleImpl class declaration 211 ⟩;
  ⟨ FoldDecisionRule class declaration 224 ⟩;
  ⟨ UnfoldDecisionRule class declaration 225 ⟩;
  ⟨ DRFixPoint class declaration 226 ⟩;
  ⟨ SimResults class declaration 233 ⟩;
  ⟨ SimResultsStats class declaration 234 ⟩;
  ⟨ SimResultsDynamicStats class declaration 235 ⟩;
  ⟨ SimResultsIRF class declaration 236 ⟩;
  ⟨ RTSimResultsStats class declaration 237 ⟩;
  ⟨ IRFResults class declaration 238 ⟩;
  ⟨ SimulationWorker class declaration 239 ⟩;
  ⟨ SimulationIRFWorker class declaration 240 ⟩;
  ⟨ RTSimulationWorker class declaration 241 ⟩;
  ⟨ RandomShockRealization class declaration 242 ⟩;
  ⟨ ExplicitShockRealization class declaration 243 ⟩;
  ⟨ GenShockRealization class declaration 244 ⟩;
#endif
```

209. This is a general interface to a shock realizations. The interface has only one method returning the shock realizations at the given time. This method is not constant, since it may change a state of the object.

```
⟨ ShockRealization class declaration 209 ⟩ ≡
  class ShockRealization {
  public:
    virtual ~ShockRealization( ) { }
    virtual void get(int n, Vector &out) = 0;
    virtual int numShocks( ) const = 0;
  };
```

This code is used in section 208.

210. This class is an abstract interface to decision rule. Its main purpose is to define a common interface for simulation of a decision rule. We need only a simulate, evaluate, cetralized clone and output method. The *simulate* method simulates the rule for a given realization of the shocks. *eval* is a primitive evaluation (it takes a vector of state variables (predetermined, both and shocks) and returns the next period variables. Both input and output are in deviations from the rule's steady. *evaluate* method makes only one step of simulation (in terms of absolute values, not deviations). *centralizedClone* returns a new copy of the decision rule, which is centralized about provided fix-point. And finally *writeMat* writes the decision rule to the MAT file.

⟨ **DecisionRule** class declaration 210 ⟩ ≡

```
class DecisionRule {
public:
  enum emethod { horner, trad };
  virtual ~DecisionRule( ) { }
  virtual TwoDMatrix *simulate(emethod em, int np, const Vector &ystart, ShockRealization
      &sr) const = 0;
  virtual void eval(emethod em, Vector &out, const ConstVector &v) const = 0;
  virtual void evaluate(emethod em, Vector &out, const ConstVector &ys, const ConstVector
      &u) const = 0;
  virtual void writeMat(mat_t *fd, const char *prefix) const = 0;
  virtual DecisionRule *centralizedClone(const Vector &fixpoint) const = 0;
  virtual const Vector &getSteady( ) const = 0;
  virtual int nexog( ) const = 0;
  virtual const PartitionY &getYPart( ) const = 0;
};
```

This code is used in section 208.

211. The main purpose of this class is to implement **DecisionRule** interface, which is a simulation. To be able to do this we have to know the partitioning of state vector y since we will need to pick only predetermined part y^*. Also, we need to know the steady state.

The decision rule will take the form:

$$y_t - \bar{y} = \sum_{i=0}^{n} \left[g_{(yu)^i}\right]_{\alpha_1 \dots \alpha_i} \prod_{m=1}^{i} \begin{bmatrix} y^*_{t-1} - \bar{y}^* \\ u_t \end{bmatrix}^{\alpha_m},$$

where the tensors $\left[g_{(yu)^i}\right]$ are tensors of the constructed container, and $\bar{y}$ is the steady state.

If we know the fix point of the rule (conditional zero shocks) $\tilde{y}$, the rule can be transformed to so called "centralized" form. This is very similar to the form above but the zero dimensional tensor is zero:

$$y_t - \tilde{y} = \sum_{i=1}^{n} \left[\tilde{g}_{(yu)^i}\right]_{\alpha_1 \dots \alpha_i} \prod_{m=1}^{i} \begin{bmatrix} y^*_{t-1} - \tilde{y}^* \\ u_t \end{bmatrix}^{\alpha_m}.$$

We provide a method and a constructor to transform a rule to the centralized form.

The class is templated, the template argument is either **KOrder**::*fold* or **KOrder**::*unfold*. So, there are two implementations of **DecisionRule** interface.

⟨ **DecisionRuleImpl** class declaration 211 ⟩ ≡

```
template⟨int t⟩ class DecisionRuleImpl : public ctraits⟨t⟩::Tpol, public DecisionRule {
protected:
  typedef typename ctraits⟨t⟩::Tpol _Tparent;
  const Vector ysteady;
  const PartitionY ypart;
  const int nu;
public:
  DecisionRuleImpl(const _Tparent &pol, const PartitionY &yp, int nuu, const Vector &ys)
  : ctraits⟨t⟩::Tpol(pol), ysteady(ys), ypart(yp), nu(nuu) { }
  DecisionRuleImpl(_Tparent &pol, const PartitionY &yp, int nuu, const Vector &ys)
  : ctraits⟨t⟩::Tpol(0, yp.ny(), pol), ysteady(ys), ypart(yp), nu(nuu) { }
  DecisionRuleImpl(const _Tg &g, const PartitionY &yp, int nuu, const Vector &ys, double
        sigma)
  : ctraits⟨t⟩::Tpol(yp.ny(), yp.nys() + nuu), ysteady(ys), ypart(yp), nu(nuu) {
    fillTensors(g, sigma); }
  DecisionRuleImpl(const DecisionRuleImpl⟨t⟩ &dr, const ConstVector &fixpoint)
  : ctraits⟨t⟩::Tpol(dr.ypart.ny(), dr.ypart.nys() + dr.nu), ysteady(fixpoint), ypart(dr.ypart),
      nu(dr.nu) { centralize(dr); }
  const Vector &getSteady() const
  { return ysteady; }
  ⟨ DecisionRuleImpl::simulate code 215 ⟩;
  ⟨ DecisionRuleImpl::evaluate code 220 ⟩;
  ⟨ DecisionRuleImpl::centralizedClone code 221 ⟩;
  ⟨ DecisionRuleImpl::writeMat code 223 ⟩;
  int nexog() const
  { return nu; }
  const PartitionY &getYPart() const
  { return ypart;
  }
protected:
```

⟨ **DecisionRuleImpl** :: *fillTensors* code 212 ⟩;
⟨ **DecisionRuleImpl** :: *centralize* code 214 ⟩;
⟨ **DecisionRuleImpl** :: *eval* code 222 ⟩;
};

This code is used in section 208.

212. Here we have to fill the tensor polynomial. This involves two separated actions. First is to evaluate the approximation at a given σ, the second is to compile the tensors $[g_{(yu)^{i+j}}]$ from $[g_{y^iu^j}]$. The first action is done here, the second is done by method *addSubTensor* of a full symmetry tensor.

The way how the evaluation is done is described here:

The $q-$order approximation to the solution can be written as:

$$\begin{aligned} y_t - \bar{y} &= \sum_{l=1}^{q} \frac{1}{l!} \left[\sum_{i+j+k=l} \binom{l}{i,j,k} \left[g_{y^iu^j\sigma^k}\right]_{\alpha_1\ldots\alpha_j\beta_1\ldots\beta_j} \prod_{m=1}^{i} [y^*_{t-1} - \bar{y}^*]^{\alpha_m} \prod_{n=1}^{j} [u_t]^{\beta_m} \sigma^k \right] \\ &= \sum_{l=1}^{q} \left[\sum_{i+j\le l} \binom{i+j}{i} \left[\sum_{k=0}^{l-i-j} \frac{1}{l!} \binom{l}{k} \left[g_{y^iu^j\sigma^k}\right] \sigma^k \right] \prod_{m=1}^{i} [y^*_{t-1} - \bar{y}^*]^{\alpha_m} \prod_{n=1}^{j} [u_t]^{\beta_m} \sigma^k \right] \end{aligned}$$

This means that for each $i+j+k=l$ we have to add

$$\frac{1}{l!}\binom{l}{k}\left[g_{y^iu^j\sigma^k}\right]\cdot\sigma^k = \frac{1}{(i+j)!k!}\left[g_{y^iu^j\sigma^k}\right]\cdot\sigma^k$$

to $g_{(yu)^{i+j}}$. In addition, note that the multiplier $\binom{i+j}{i}$ is applied when the fully symmetric tensor $[g_{(yu)^{i+j}}]$ is evaluated.

So we go through $i+j=d=0\ldots q$ and in each loop we form the fully symmetric tensor $[g_{(yu)^l}]$ and insert it to the container.

⟨ **DecisionRuleImpl** :: *fillTensors* code 212 ⟩ ≡

```
void fillTensors(const _Tg &g, double sigma)
{
  IntSequence tns(2);
  tns[0] = ypart.nys();
  tns[1] = nu;
  int dfact = 1;
  for (int d = 0; d ≤ g.getMaxDim(); d++, dfact *= d) {
    _Ttensym *g_yud = new _Ttensym(ypart.ny(), ypart.nys() + nu, d);
    g_yud→zeros();
    ⟨ fill tensor of g_yud of dimension d 213 ⟩;
    this→insert(g_yud);
  }
}
```

This code is cited in section 213.

This code is used in section 211.

213. Here we have to fill the tensor $\left[g_{(yu)^d}\right]$. So we go through all pairs (i,j) giving $i+j=d$, and through all k from zero up to maximal dimension minus d. In this way we go through all symmetries of $g_{y^i u^j \sigma^k}$ which will be added to $g_{(yu)^d}$.

Note that at the beginning, *dfact* is a factorial of d. We calculate *kfact* is equal to $k!$. As indicated in ⟨ **DecisionRuleImpl**::*fillTensors* code 212 ⟩, the added tensor is thus multiplied with $\frac{1}{d!k!}\sigma^k$.

```
⟨ fill tensor of g_yud of dimension d 213 ⟩ ≡
  for (int i = 0; i ≤ d; i++) {
    int j = d − i;
    int kfact = 1;
    _Ttensor tmp(ypart.ny(), TensorDimens(Symmetry(i, j), tns));
    tmp.zeros();
    for (int k = 0; k + d ≤ g.getMaxDim(); k++, kfact *= k) {
      Symmetry sym(i, j, 0, k);
      if (g.check(sym)) {
        double mult = pow(sigma, k)/dfact/kfact;
        tmp.add(mult, *(g.get(sym)));
      }
    }
    g_yud→addSubTensor(tmp);
  }
```

This code is used in section 212.

214. The centralization is straightforward. We suppose here that the object's steady state is the fix point $\tilde{y}$. It is clear that the new derivatives $\left[\tilde{g}_{(yu)^i}\right]$ will be equal to the derivatives of the original decision rule dr at the new steady state $\tilde{y}$. So, the new derivatives are obtained by derivating the given decision rule dr and evaluating its polynomial at

$$dstate = \begin{bmatrix} \tilde{y}^* - \bar{y}^* \\ 0 \end{bmatrix},$$

where $\bar{y}$ is the steady state of the original rule dr.

```
⟨ DecisionRuleImpl::centralize code 214 ⟩ ≡
  void centralize(const DecisionRuleImpl &dr)
  {
    Vector dstate(ypart.nys() + nu);
    dstate.zeros();
    Vector dstate_star(dstate, 0, ypart.nys());
    ConstVector newsteady_star(ysteady, ypart.nstat, ypart.nys());
    ConstVector oldsteady_star(dr.ysteady, ypart.nstat, ypart.nys());
    dstate_star.add(1.0, newsteady_star);
    dstate_star.add(−1.0, oldsteady_star);
    _Tpol pol(dr);
    int dfac = 1;
    for (int d = 1; d ≤ dr.getMaxDim(); d++, dfac *= d) {
      pol.derivative(d − 1);
      _Ttensym *der = pol.evalPartially(d, dstate);
      der→mult(1.0/dfac);
      this→insert(der);
    }
  }
```

This code is used in section 211.

215. Here we evaluate repeatedly the polynomial storing results in the created matrix. For exogenous shocks, we use **ShockRealization** class, for predetermined variables, we use *ystart* as the first state. The *ystart* vector is required to be all state variables *ypart.ny*( ), although only the predetermined part of *ystart* is used.

We simulate in terms of Δy, this is, at the beginning the *ysteady* is canceled from *ystart*, we simulate, and at the end *ysteady* is added to all columns of the result.

⟨ **DecisionRuleImpl**::*simulate* code 215 ⟩ ≡

```
TwoDMatrix *simulate(emethod em, int np, const Vector &ystart, ShockRealization &sr) const
{
  KORD_RAISE_IF(ysteady.length( ) ≠ ystart.length( ),
      "Start and steady lengths differ in DecisionRuleImpl::simulate");
  TwoDMatrix *res = new TwoDMatrix(ypart.ny( ), np);
  ⟨ initialize vectors and subvectors for simulation 216 ⟩;
  ⟨ perform the first step of simulation 217 ⟩;
  ⟨ perform all other steps of simulations 218 ⟩;
  ⟨ add the steady state to columns of res 219 ⟩;
  return res;
}
```

This code is used in section 211.

216. Here allocate the stack vector $(\Delta y^*, u)$, define the subvectors dy, and u, then we pickup predetermined parts of *ystart* and *ysteady*.

⟨ initialize vectors and subvectors for simulation 216 ⟩ ≡

```
Vector dyu(ypart.nys( ) + nu);
ConstVector ystart_pred(ystart, ypart.nstat, ypart.nys( ));
ConstVector ysteady_pred(ysteady, ypart.nstat, ypart.nys( ));
Vector dy(dyu, 0, ypart.nys( ));
Vector u(dyu, ypart.nys( ), nu);
```

See also section 277.

This code is used in sections 215 and 276.

217. We cancel *ysteady* from *ystart*, get realization to u, and evaluate the polynomial.

⟨ perform the first step of simulation 217 ⟩ ≡

```
dy = ystart_pred;
dy.add(−1.0, ysteady_pred);
sr.get(0, u);
Vector out(*res, 0);
eval(em, out, dyu);
```

This code is used in section 215.

218. Also clear. If the result at some period is not finite, we pad the rest of the matrix with zeros.

```
⟨ perform all other steps of simulations 218 ⟩ ≡
  int i = 1;
  while (i < np) {
    ConstVector ym(*res, i − 1);
    ConstVector dym(ym, ypart.nstat, ypart.nys());
    dy = dym;
    sr.get(i, u);
    Vector out(*res, i);
    eval(em, out, dyu);
    if (¬out.isFinite()) {
      if (i + 1 < np) {
        TwoDMatrix rest(*res, i + 1, np − i − 1);
        rest.zeros();
      }
      break;
    }
    i++;
  }
```

This code is used in section 215.

219. Even clearer. We add the steady state to the numbers computed above and leave the padded columns to zero.

```
⟨ add the steady state to columns of res 219 ⟩ ≡
  for (int j = 0; j < i; j++) {
    Vector col(*res, j);
    col.add(1.0, ysteady);
  }
```

This code is used in section 215.

220. This is one period evaluation of the decision rule. The simulation is a sequence of repeated one period evaluations with a difference, that the steady state (fix point) is cancelled and added once. Hence we have two special methods.

```
⟨ DecisionRuleImpl::evaluate code 220 ⟩ ≡
  void evaluate(emethod em, Vector &out, const ConstVector &ys, const ConstVector &u) const
  {
    KORD_RAISE_IF(ys.length() ≠ ypart.nys() ∨ u.length() ≠ nu,
        "Wrong dimensions of input vectors in DecisionRuleImpl::evaluate");
    KORD_RAISE_IF(out.length() ≠ ypart.ny(),
        "Wrong dimension of output vector in DecisionRuleImpl::evaluate");
    ConstVector ysteady_pred(ysteady, ypart.nstat, ypart.nys());
    Vector ys_u(ypart.nys() + nu);
    Vector ys_u1(ys_u, 0, ypart.nys());
    ys_u1 = ys;
    ys_u1.add(−1.0, ysteady_pred);
    Vector ys_u2(ys_u, ypart.nys(), nu);
    ys_u2 = u;
    eval(em, out, ys_u);
    out.add(1.0, ysteady);
  }
```

This code is used in section 211.

221. This is easy. We just return the newly created copy using the centralized constructor.

```
⟨ DecisionRuleImpl::centralizedClone code 221 ⟩ ≡
  DecisionRule *centralizedClone(const Vector &fixpoint) const
  {
    return new DecisionRuleImpl⟨t⟩(*this, fixpoint);
  }
```

This code is used in section 211.

222. Here we only encapsulate two implementations to one, deciding according to the parameter.

```
⟨ DecisionRuleImpl::eval code 222 ⟩ ≡
  void eval(emethod em, Vector &out, const ConstVector &v) const
  {
    if (em ≡ DecisionRule::horner) _Tparent::evalHorner(out, v);
    else _Tparent::evalTrad(out, v);
  }
```

This code is used in section 211.

223. Write the decision rule and steady state to the MAT file.

```
⟨ DecisionRuleImpl::writeMat code 223 ⟩ ≡
  void writeMat(mat_t * fd, const char *prefix) const
  {
    ctraits⟨t⟩::Tpol::writeMat(fd, prefix);
    TwoDMatrix dum(ysteady.length(), 1);
    dum.getData() = ysteady;
    char tmp[100];
    sprintf(tmp, "%s_ss", prefix);
    ConstTwoDMatrix(dum).writeMat(fd, tmp);
  }
```

This code is used in section 211.

224. This is exactly the same as **DecisionRuleImpl**⟨**KOrder**::*fold*⟩. The only difference is that we have a conversion from **UnfoldDecisionRule**, which is exactly **DecisionRuleImpl**⟨**KOrder**::*unfold*⟩.

```
⟨ FoldDecisionRule class declaration 224 ⟩ ≡
  class UnfoldDecisionRule;
  class FoldDecisionRule : public DecisionRuleImpl⟨KOrder::fold⟩ {
    friend class UnfoldDecisionRule;
  public:
    FoldDecisionRule(const ctraits⟨KOrder::fold⟩::Tpol &pol, const PartitionY &yp, int
          nuu, const Vector &ys)
    : DecisionRuleImpl⟨KOrder::fold⟩(pol, yp, nuu, ys) { }
    FoldDecisionRule(ctraits⟨KOrder::fold⟩::Tpol &pol, const PartitionY &yp, int nuu, const
          Vector &ys)
    : DecisionRuleImpl⟨KOrder::fold⟩(pol, yp, nuu, ys) { }
    FoldDecisionRule(const ctraits⟨KOrder::fold⟩::Tg &g, const PartitionY &yp, int nuu, const
          Vector &ys, double sigma)
    : DecisionRuleImpl⟨KOrder::fold⟩(g, yp, nuu, ys, sigma) { }
    FoldDecisionRule(const DecisionRuleImpl⟨KOrder::fold⟩ &dr, const ConstVector &fixpoint)
    : DecisionRuleImpl⟨KOrder::fold⟩(dr, fixpoint) { }
    FoldDecisionRule(const UnfoldDecisionRule &udr);
  };
```

This code is used in section 208.

225. This is exactly the same as **DecisionRuleImpl**⟨**KOrder**::*unfold*⟩, but with a conversion from **FoldDecisionRule**, which is exactly **DecisionRuleImpl**⟨**KOrder**::*fold*⟩.

```
⟨ UnfoldDecisionRule class declaration 225 ⟩ ≡
  class UnfoldDecisionRule : public DecisionRuleImpl⟨KOrder::unfold⟩ {
    friend class FoldDecisionRule;
  public:
    UnfoldDecisionRule(const ctraits⟨KOrder::unfold⟩::Tpol &pol, const PartitionY &yp, int
          nuu, const Vector &ys)
    : DecisionRuleImpl⟨KOrder::unfold⟩(pol, yp, nuu, ys) { }
    UnfoldDecisionRule(ctraits⟨KOrder::unfold⟩::Tpol &pol, const PartitionY &yp, int
          nuu, const Vector &ys)
    : DecisionRuleImpl⟨KOrder::unfold⟩(pol, yp, nuu, ys) { }
    UnfoldDecisionRule(const ctraits⟨KOrder::unfold⟩::Tg &g, const PartitionY &yp, int
          nuu, const Vector &ys, double sigma)
    : DecisionRuleImpl⟨KOrder::unfold⟩(g, yp, nuu, ys, sigma) { }
    UnfoldDecisionRule(const DecisionRuleImpl⟨KOrder::unfold⟩ &dr, const ConstVector
          &fixpoint)
    : DecisionRuleImpl⟨KOrder::unfold⟩(dr, fixpoint) { }
    UnfoldDecisionRule(const FoldDecisionRule &udr);
  };
```

This code is used in section 208.

226. This class serves for calculation of the fix point of the decision rule given that the shocks are zero. The class is very similar to the **DecisionRuleImpl**. Besides the calculation of the fix point, the only difference between **DRFixPoint** and **DecisionRuleImpl** is that the derivatives wrt. shocks are ignored (since shocks are zero during the calculations). That is why have a different *fillTensor* method.

The solution algorithm is Newton and is described in ⟨ **DRFixPoint**::*solveNewton* code 230 ⟩. It solves $F(y) = 0$, where $F = g(y, 0) - y$. The function F is given by its derivatives *bigf*. The Jacobian of the solved system is given by derivatives stored in *bigfder*.

```
⟨ DRFixPoint class declaration 226 ⟩ ≡
  template⟨int t⟩ class DRFixPoint : public ctraits⟨t⟩::Tpol {
    typedef typename ctraits⟨t⟩::Tpol _Tparent;
    static int max_iter;
    static int max_newton_iter;
    static int newton_pause;
    static double tol;
    const Vector ysteady;
    const PartitionY ypart;
    _Tparent *bigf;
    _Tparent *bigfder;
  public:
    typedef typename DecisionRule::emethod emethod;
    ⟨ DRFixPoint constructor code 227 ⟩;
    ⟨ DRFixPoint destructor code 228 ⟩;
    ⟨ DRFixPoint::calcFixPoint code 232 ⟩;
    int getNumIter() const
    { return iter; }
    int getNewtonLastIter() const
    { return newton_iter_last; }
    int getNewtonTotalIter() const
    { return newton_iter_total; }
  protected:
    ⟨ DRFixPoint::fillTensors code 229 ⟩;
    ⟨ DRFixPoint::solveNewton code 230 ⟩;
  private:
    int iter;
    int newton_iter_last;
    int newton_iter_total;
  };
```

This code is used in section 208.

227. Here we have to setup the function $F = g(y, 0) - y$ and $\frac{\partial F}{\partial y}$. The former is taken from the given derivatives of g where a unit matrix is subtracted from the first derivative (**Symmetry**(1)). Then the derivative of the F polynomial is calculated.

```
⟨ DRFixPoint constructor code 227 ⟩ ≡
  DRFixPoint(const _Tg &g, const PartitionY &yp, const Vector &ys, double sigma)
  : ctraits⟨t⟩::Tpol(yp.ny(), yp.nys()), ysteady(ys), ypart(yp), bigf(Λ), bigfder(Λ) {
    fillTensors(g, sigma);
    _Tparent yspol(ypart.nstat, ypart.nys(), *this);
    bigf = new _Tparent((const _Tparent &) yspol);
    _Ttensym *frst = bigf→get(Symmetry(1));
    for (int i = 0; i < ypart.nys(); i++) frst→get(i, i) = frst→get(i, i) − 1;
    bigfder = new _Tparent(*bigf, 0);
  }
```

This code is used in section 226.

228.

```
⟨ DRFixPoint destructor code 228 ⟩ ≡
  virtual ~DRFixPoint()
  {
    if (bigf) delete bigf;
    if (bigfder) delete bigfder;
  }
```

This code is used in section 226.

229. Here we fill the tensors for the **DRFixPoint** class. We ignore the derivatives $g_{y^i u^j \sigma^k}$ for which $j > 0$. So we go through all dimensions d, and all k such that $d + k$ is between the maximum dimension and d, and add $\frac{\sigma^k}{d!k!} g_{y^d \sigma^k}$ to the tensor $g_{(y)^d}$.

```
⟨ DRFixPoint::fillTensors code 229 ⟩ ≡
  void fillTensors(const _Tg &g, double sigma)
  {
    int dfact = 1;
    for (int d = 0; d ≤ g.getMaxDim(); d++, dfact *= d) {
      _Ttensym *g_yd = new _Ttensym(ypart.ny(), ypart.nys(), d);
      g_yd→zeros();
      int kfact = 1;
      for (int k = 0; d + k ≤ g.getMaxDim(); k++, kfact *= k) {
        if (g.check(Symmetry(d, 0, 0, k))) {
          const _Ttensor *ten = g.get(Symmetry(d, 0, 0, k));
          double mult = pow(sigma, k)/dfact/kfact;
          g_yd→add(mult, *ten);
        }
      }
      this→insert(g_yd);
    }
  }
```

This code is used in section 226.

230. This tries to solve polynomial equation $F(y) = 0$, where F polynomial is *bigf* and its derivative is in *bigfder*. It returns true if the Newton converged. The method takes the given vector as initial guess, and rewrites it with a solution. The method guarantees to return the vector, which has smaller norm of the residual. That is why the input/output vector y is always changed.

The method proceeds with a Newton step, if the Newton step improves the residual error. So we track residual errors in *flastnorm* and *fnorm* (former and current). In addition, at each step we search for an underrelaxation parameter *urelax*, which improves the residual. If *urelax* is less that *urelax_threshold*, we stop searching and stop the Newton.

```
⟨ DRFixPoint::solveNewton code 230 ⟩ ≡
  bool solveNewton(Vector &y)
  {
    const double urelax_threshold = 1. · 10^-5;
    Vector sol((const Vector &) y);
    Vector delta(y.length());
    newton_iter_last = 0;
    bool delta_finite = true;
    double flastnorm = 0.0;
    double fnorm = 0.0;
    bool converged = false;
    double urelax = 1.0;
    do {
      _Ttensym *jacob = bigfder→evalPartially(1, sol);
      bigf→evalHorner(delta, sol);
      if (newton_iter_last ≡ 0) flastnorm = delta.getNorm();
      delta_finite = delta.isFinite();
      if (delta_finite) {
        ConstTwoDMatrix(*jacob).multInvLeft(delta);
        ⟨ find urelax improving residual 231 ⟩;
        sol.add(-urelax, delta);
        delta_finite = delta.isFinite();
      }
      delete jacob;
      newton_iter_last++;
      converged = delta_finite ∧ fnorm < tol;
      flastnorm = fnorm;
    } while (¬converged ∧ newton_iter_last < max_newton_iter ∧ urelax > urelax_threshold);
    newton_iter_total += newton_iter_last;
    if (¬converged) newton_iter_last = 0;
    y = (const Vector &) sol;
    return converged;
  }
```

This code is cited in section 226.

This code is used in section 226.

231. Here we find the *urelax*. We cycle as long as the new residual size *fnorm* is greater than last residual size *flastnorm*. If the urelax is less than *urelax_threshold* we give up. The *urelax* is damped by the ratio of *flastnorm* and *fnorm*. It the ratio is close to one, we damp by one half.

```
⟨ find urelax improving residual 231 ⟩ ≡
  bool urelax_found = false;
  urelax = 1.0;
  while (¬urelax_found ∧ urelax > urelax_threshold) {
    Vector soltmp((const Vector &) sol);
    soltmp.add(-urelax, delta);
    Vector f(sol.length());
    bigf→evalHorner(f, soltmp);
    fnorm = f.getNorm();
    if (fnorm ≤ flastnorm) urelax_found = true;
    else urelax *= std::min(0.5, flastnorm/fnorm);
  }
```

This code is used in section 230.

232. This method solves the fix point of the no-shocks rule $y_{t+1} = f(y_t)$. It combines dull steps with Newton attempts. The dull steps correspond to evaluations setting $y_{t+1} = f(y_t)$. For reasonable models the dull steps converge to the fix-point but very slowly. That is why we make Newton attempt from time to time. The frequency of the Newton attempts is given by *newton_pause*. We perform the calculations in deviations from the steady state. So, at the end, we have to add the steady state.

The method also sets the members *iter*, *newton_iter_last* and *newton_iter_total*. These numbers can be examined later.

The *out* vector is not touched if the algorithm has not convered.

```
⟨ DRFixPoint::calcFixPoint code 232 ⟩ ≡
  bool calcFixPoint(emethod em, Vector &out)
  {
    KORD_RAISE_IF(out.length() ≠ ypart.ny(), "Wrong length of out in DRFixPoint::calcFixPoint");
    Vector delta(ypart.nys());
    Vector ystar(ypart.nys());
    ystar.zeros();
    iter = 0;
    newton_iter_last = 0;
    newton_iter_total = 0;
    bool converged = false;
    do {
      if ((iter/newton_pause) * newton_pause ≡ iter) converged = solveNewton(ystar);
      if (¬converged) {
        bigf¬evalHorner(delta, ystar);
        KORD_RAISE_IF_X(¬delta.isFinite(), "NaN or Inf asserted in DRFixPoint::calcFixPoint",
            KORD_FP_NOT_FINITE);
        ystar.add(1.0, delta);
        converged = delta.getNorm() < tol;
      }
      iter++;
    } while (iter < max_iter ∧ ¬converged);
    if (converged) {
      _Tparent::evalHorner(out, ystar);
      out.add(1.0, ysteady);
    }
    return converged;
  }
```

This code is used in section 226.

233. This is a basically a number of matrices of the same dimensions, which can be obtained as simulation results from a given decision rule and shock realizations. We also store the realizations of shocks.

```
⟨ SimResults class declaration 233 ⟩ ≡
  class ExplicitShockRealization;
  class SimResults {
  protected:
    int num_y;
    int num_per;
    int num_burn;
    vector⟨TwoDMatrix *⟩ data;
    vector⟨ExplicitShockRealization *⟩ shocks;
  public:
    SimResults(int ny, int nper, int nburn = 0)
    : num_y(ny), num_per(nper), num_burn(nburn) { }
    virtual ~SimResults( );
    void simulate(int num_sim, const DecisionRule &dr, const Vector &start, const TwoDMatrix
        &vcov, Journal &journal);
    void simulate(int num_sim, const DecisionRule &dr, const Vector &start, const TwoDMatrix
        &vcov);
    int getNumPer( ) const
    { return num_per; }
    int getNumBurn( ) const
    { return num_burn; }
    int getNumSets( ) const
    { return (int) data.size( ); }
    const TwoDMatrix &getData(int i) const
    { return *(data[i]); }
    const ExplicitShockRealization &getShocks(int i) const
    { return *(shocks[i]); }
    bool addDataSet(TwoDMatrix *d, ExplicitShockRealization *sr);
    void writeMat(const char *base, const char *lname) const;
    void writeMat(mat_t * fd, const char *lname) const;
  };
```

This code is used in section 208.

234. This does the same as **SimResults** plus it calculates means and covariances of the simulated data.

```
⟨ SimResultsStats class declaration 234 ⟩ ≡
  class SimResultsStats : public SimResults {
  protected:
    Vector mean;
    TwoDMatrix vcov;
  public:
    SimResultsStats(int ny, int nper, int nburn = 0)
    : SimResults(ny, nper, nburn), mean(ny), vcov(ny, ny) { }
    void simulate(int num_sim, const DecisionRule &dr, const Vector &start, const TwoDMatrix
        &vcov, Journal &journal);
    void writeMat(mat_t * fd, const char *lname) const;
  protected:
    void calcMean( );
    void calcVcov( );
  };
```

This code is used in section 208.

235. This does the similar thing as **SimResultsStats** but the statistics are not calculated over all periods but only within each period. Then we do not calculate covariances with periods but only variances.

```
⟨ SimResultsDynamicStats class declaration 235 ⟩ ≡
  class SimResultsDynamicStats : public SimResults {
  protected:
    TwoDMatrix mean;
    TwoDMatrix variance;
  public:
    SimResultsDynamicStats(int ny, int nper, int nburn = 0)
    : SimResults(ny, nper, nburn), mean(ny, nper), variance(ny, nper) { }
    void simulate(int num_sim, const DecisionRule &dr, const Vector &start, const TwoDMatrix
        &vcov, Journal &journal);
    void writeMat(mat_t * fd, const char *lname) const;
  protected:
    void calcMean( );
    void calcVariance( );
  };
```

This code is used in section 208.

236. This goes through control simulation results, and for each control it adds a given impulse to a given shock and runs a simulation. The control simulation is then cancelled and the result is stored. After that these results are averaged with variances calculated.

The means and the variances are then written to the MAT-4 file.

⟨ **SimResultsIRF** class declaration 236 ⟩ ≡

```
class SimulationIRFWorker;
class SimResultsIRF : public SimResults {
  friend class SimulationIRFWorker;
protected:
  const SimResults &control;
  int ishock;
  double imp;
  TwoDMatrix means;
  TwoDMatrix variances;
public:
  SimResultsIRF(const SimResults &cntl, int ny, int nper, int i, double impulse)
  : SimResults(ny, nper, 0), control(cntl), ishock(i), imp(impulse), means(ny, nper),
      variances(ny, nper) { }
  void simulate(const DecisionRule &dr, Journal &journal);
  void simulate(const DecisionRule &dr);
  void writeMat(mat_t *fd, const char *lname) const;
protected:
  void calcMeans( );
  void calcVariances( );
};
```

This code is used in section 208.

237. This simulates and gathers all statistics from the real time simulations. In the *simulate* method, it runs **RTSimulationWorker**s which accummulate information from their own estimates. The estimation is done by means of **NormalConj** class, which is a conjugate family of densities for normal distibutions.

```
⟨ RTSimResultsStats class declaration 237 ⟩ ≡
  class RTSimulationWorker;
  class RTSimResultsStats {
    friend class RTSimulationWorker;
  protected:
    Vector mean;
    TwoDMatrix vcov;
    int num_per;
    int num_burn;
    NormalConj nc;
    int incomplete_simulations;
    int thrown_periods;
  public:
    RTSimResultsStats(int ny, int nper, int nburn = 0)
    : mean(ny), vcov(ny, ny), num_per(nper), num_burn(nburn), nc(ny), incomplete_simulations(0),
        thrown_periods(0) { }
    void simulate(int num_sim, const DecisionRule &dr, const Vector &start, const TwoDMatrix
        &vcov, Journal &journal);
    void simulate(int num_sim, const DecisionRule &dr, const Vector &start, const TwoDMatrix
        &vcov);
    void writeMat(mat_t * fd, const char *lname);
  };
```

This code is used in section 208.

238. For each shock, this simulates plus and minus impulse. The class maintains a vector of simulation results, each gets a particular shock and sign (positive/negative). The results of type **SimResultsIRF** are stored in a vector so that even ones are positive, odd ones are negative.

The constructor takes a reference to the control simulations, which must be finished before the constructor is called. The control simulations are passed to all **SimResultsIRF**s.

The constructor also takes the vector of indices of exogenous variables (*ili*) for which the IRFs are generated. The list is kept (as *irf_list_ind*) for other methods.

```
⟨ IRFResults class declaration 238 ⟩ ≡
  class DynamicModel;
  class IRFResults {
    vector⟨SimResultsIRF *⟩ irf_res;
    const DynamicModel &model;
    vector⟨int⟩ irf_list_ind;
  public:
    IRFResults(const DynamicModel &mod, const DecisionRule &dr, const SimResults
        &control, const vector⟨int⟩ &ili, Journal &journal);
    ~IRFResults( );
    void writeMat(mat_t * fd, const char *prefix) const;
  };
```

This code is used in section 208.

239. This worker simulates the given decision rule and inserts the result to **SimResults**.

```
⟨ SimulationWorker class declaration 239 ⟩ ≡
  class SimulationWorker : public THREAD
  {
  protected:
    SimResults &res;
    const DecisionRule &dr;
    DecisionRule::emethod em;
    int np;
    const Vector &st;
    ShockRealization &sr;
  public:
    SimulationWorker(SimResults &sim_res, const DecisionRule &dec_rule,
                     DecisionRule::emethod emet, int num_per, const Vector &start, ShockRealization
                     &shock_r)
    : res(sim_res), dr(dec_rule), em(emet), np(num_per), st(start), sr(shock_r) { }
    void operator()();
  }
  ;
```

This code is used in section 208.

240. This worker simulates a given impulse *imp* to a given shock *ishock* based on a given control simulation with index *idata*. The control simulations are contained in **SimResultsIRF** which is passed to the constructor.

```
⟨ SimulationIRFWorker class declaration 240 ⟩ ≡
  class SimulationIRFWorker : public THREAD
  {
    SimResultsIRF &res;
    const DecisionRule &dr;
    DecisionRule::emethod em;
    int np;
    int idata;
    int ishock;
    double imp;
  public:
    SimulationIRFWorker(SimResultsIRF &sim_res, const DecisionRule &dec_rule,
                        DecisionRule::emethod emet, int num_per, int id, int ishck, double impulse)
    : res(sim_res), dr(dec_rule), em(emet), np(num_per), idata(id), ishock(ishck), imp(impulse) { }
    void operator()();
  }
  ;
```

This code is used in section 208.

241. This class does the real time simulation job for **RTSimResultsStats**. It simulates the model period by period. It accummulates the information in the **RTSimResultsStats**::*nc*. If NaN or Inf is observed, it ends the simulation and adds to the *thrown_periods* of **RTSimResultsStats**.

```
⟨ RTSimulationWorker class declaration 241 ⟩ ≡
  class RTSimulationWorker : public THREAD
  {
  protected:
    RTSimResultsStats &res;
    const DecisionRule &dr;
    DecisionRule::emethod em;
    int np;
    const Vector &ystart;
    ShockRealization &sr;
  public:
    RTSimulationWorker(RTSimResultsStats &sim_res, const DecisionRule &dec_rule,
                       DecisionRule::emethod emet, int num_per, const Vector &start, ShockRealization
                       &shock_r)
    : res(sim_res), dr(dec_rule), em(emet), np(num_per), ystart(start), sr(shock_r) { }
    void operator()();
  }
  ;
```

This code is used in section 208.

242. This class generates draws from Gaussian distribution with zero mean and the given variance-covariance matrix. It stores the factor of vcov V matrix, yielding $FF^T = V$.

```
⟨ RandomShockRealization class declaration 242 ⟩ ≡
  class RandomShockRealization : virtual public ShockRealization {
  protected:
    MersenneTwister mtwister;
    TwoDMatrix factor;
  public:
    RandomShockRealization(const TwoDMatrix &v, unsigned int iseed)
    : mtwister(iseed), factor(v.nrows(),v.nrows()) { schurFactor(v); }
    RandomShockRealization(const RandomShockRealization &sr)
    : mtwister(sr.mtwister), factor(sr.factor) { }
    virtual ~RandomShockRealization() { }
    void get(int n, Vector &out);
    int numShocks() const
    { return factor.nrows(); }
  protected:
    void choleskyFactor(const TwoDMatrix &v);
    void schurFactor(const TwoDMatrix &v);
  };
```

This code is used in section 208.

243. This is just a matrix of finite numbers. It can be constructed from any **ShockRealization** with a given number of periods.

```
⟨ ExplicitShockRealization class declaration 243 ⟩ ≡
  class ExplicitShockRealization : virtual public ShockRealization {
    TwoDMatrix shocks;
  public:
    ExplicitShockRealization(const TwoDMatrix &sh)
    : shocks(sh) { }
    ExplicitShockRealization(const ConstTwoDMatrix &sh)
    : shocks(sh) { }
    ExplicitShockRealization(const ExplicitShockRealization &sr)
    : shocks(sr.shocks) { }
    ExplicitShockRealization(ShockRealization &sr, int num_per);
    void get(int n, Vector &out);
    int numShocks( ) const
    { return shocks.nrows( ); }
    const TwoDMatrix &getShocks( )
    { return shocks; }
    void addToShock(int ishock, int iper, double val);
    void print( ) const
    { shocks.print( ); }
  };
```

This code is used in section 208.

244. This represents a user given shock realization. The first matrix of the constructor is a covariance matrix of shocks, the second matrix is a rectangular matrix, where columns correspond to periods, rows to shocks. If an element of the matrix is `NaN`, or `Inf`, or `-Inf`, then the random shock is taken instead of that element.

In this way it is a generalization of both **RandomShockRealization** and **ExplicitShockRealization**.

```
⟨ GenShockRealization class declaration 244 ⟩ ≡
  class GenShockRealization : public RandomShockRealization,
        public ExplicitShockRealization {
  public:
    GenShockRealization(const TwoDMatrix &v, const TwoDMatrix &sh, int seed)
    : RandomShockRealization(v, seed), ExplicitShockRealization(sh) {
      KORD_RAISE_IF(sh.nrows( ) ≠ v.nrows( ) ∨ v.nrows( ) ≠ v.ncols( ),
          "Wrong dimension of input matrix in GenShockRealization constructor");
    }
    void get(int n, Vector &out);
    int numShocks( ) const
    { return RandomShockRealization::numShocks( ); }
  };
```

This code is used in section 208.

245. End of `decision_rule.h` file.

246. Start of `decision_rule.cpp` file.

```
#include "kord_exception.h"
#include "decision_rule.h"
#include "dynamic_model.h"
#include "SymSchurDecomp.h"
#include <dynlapack.h>
#include <limits>
  template<> int DRFixPoint<KOrder::fold>::max_iter = 10000;
  template<> int DRFixPoint<KOrder::unfold>::max_iter = 10000;
  template<> double DRFixPoint<KOrder::fold>::tol = 1.*10^-10;
  template<> double DRFixPoint<KOrder::unfold>::tol = 1.*10^-10;
  template<> int DRFixPoint<KOrder::fold>::max_newton_iter = 50;
  template<> int DRFixPoint<KOrder::unfold>::max_newton_iter = 50;
  template<> int DRFixPoint<KOrder::fold>::newton_pause = 100;
  template<> int DRFixPoint<KOrder::unfold>::newton_pause = 100;
  ⟨ FoldDecisionRule conversion from UnfoldDecisionRule 247 ⟩;
  ⟨ UnfoldDecisionRule conversion from FoldDecisionRule 248 ⟩;
  ⟨ SimResults destructor 249 ⟩;
  ⟨ SimResults::simulate code1 250 ⟩;
  ⟨ SimResults::simulate code2 251 ⟩;
  ⟨ SimResults::addDataSet code 252 ⟩;
  ⟨ SimResults::writeMat code1 253 ⟩;
  ⟨ SimResults::writeMat code2 254 ⟩;
  ⟨ SimResultsStats::simulate code 255 ⟩;
  ⟨ SimResultsStats::writeMat code 256 ⟩;
  ⟨ SimResultsStats::calcMean code 257 ⟩;
  ⟨ SimResultsStats::calcVcov code 258 ⟩;
  ⟨ SimResultsDynamicStats::simulate code 259 ⟩;
  ⟨ SimResultsDynamicStats::writeMat code 260 ⟩;
  ⟨ SimResultsDynamicStats::calcMean code 261 ⟩;
  ⟨ SimResultsDynamicStats::calcVariance code 262 ⟩;
  ⟨ SimResultsIRF::simulate code1 263 ⟩;
  ⟨ SimResultsIRF::simulate code2 264 ⟩;
  ⟨ SimResultsIRF::calcMeans code 265 ⟩;
  ⟨ SimResultsIRF::calcVariances code 266 ⟩;
  ⟨ SimResultsIRF::writeMat code 267 ⟩;
  ⟨ RTSimResultsStats::simulate code1 268 ⟩;
  ⟨ RTSimResultsStats::simulate code2 269 ⟩;
  ⟨ RTSimResultsStats::writeMat code 270 ⟩;
  ⟨ IRFResults constructor 271 ⟩;
  ⟨ IRFResults destructor 272 ⟩;
  ⟨ IRFResults::writeMat code 273 ⟩;
  ⟨ SimulationWorker::operator()() code 274 ⟩;
  ⟨ SimulationIRFWorker::operator()() code 275 ⟩;
  ⟨ RTSimulationWorker::operator()() code 276 ⟩;
  ⟨ RandomShockRealization::choleskyFactor code 280 ⟩;
  ⟨ RandomShockRealization::schurFactor code 281 ⟩;
  ⟨ RandomShockRealization::get code 282 ⟩;
  ⟨ ExplicitShockRealization constructor code 283 ⟩;
  ⟨ ExplicitShockRealization::get code 284 ⟩;
  ⟨ ExplicitShockRealization::addToShock code 285 ⟩;
  ⟨ GenShockRealization::get code 286 ⟩;
```

247.

```
⟨ FoldDecisionRule conversion from UnfoldDecisionRule 247 ⟩ ≡
  FoldDecisionRule::FoldDecisionRule(const UnfoldDecisionRule &udr)
  : DecisionRuleImpl⟨KOrder::fold⟩(ctraits⟨KOrder::fold⟩::Tpol(udr.nrows(), udr.nvars()),
        udr.ypart, udr.nu, udr.ysteady) {
    for (ctraits⟨KOrder::unfold⟩::Tpol::const_iterator it = udr.begin(); it ≠ udr.end(); ++it) {
      insert(new ctraits⟨KOrder::fold⟩::Ttensym(*((*it).second)));
    }
  }
```

This code is used in section 246.

248.

```
⟨ UnfoldDecisionRule conversion from FoldDecisionRule 248 ⟩ ≡
  UnfoldDecisionRule::UnfoldDecisionRule(const FoldDecisionRule &fdr)
  : DecisionRuleImpl⟨KOrder::unfold⟩(ctraits⟨KOrder::unfold⟩::Tpol(fdr.nrows(), fdr.nvars()),
        fdr.ypart, fdr.nu, fdr.ysteady) {
    for (ctraits⟨KOrder::fold⟩::Tpol::const_iterator it = fdr.begin(); it ≠ fdr.end(); ++it) {
      insert(new ctraits⟨KOrder::unfold⟩::Ttensym(*((*it).second)));
    }
  }
```

This code is used in section 246.

249.

```
⟨ SimResults destructor 249 ⟩ ≡
  SimResults::~SimResults()
  {
    for (int i = 0; i < getNumSets(); i++) {
      delete data[i];
      delete shocks[i];
    }
  }
```

This code is used in section 246.

250. This runs simulations with an output to journal file. Note that we report how many simulations had to be thrown out due to Nan or Inf.

```
⟨ SimResults::simulate code1 250 ⟩ ≡
  void SimResults::simulate(int num_sim, const DecisionRule &dr, const Vector &start, const
        TwoDMatrix &vcov, Journal &journal)
  {
    JournalRecordPair paa(journal);
    paa << "Performing " << num_sim << " stochastic simulations for " << num_per <<
        " periods burning " << num_burn << " initial periods" << endrec;
    simulate(num_sim, dr, start, vcov);
    int thrown = num_sim - data.size();
    if (thrown > 0) {
      JournalRecord rec(journal);
      rec << "I had to throw " << thrown << " simulations away due to Nan or Inf" << endrec;
    }
  }
```

This code is used in section 246.

251. This runs a given number of simulations by creating **SimulationWorker** for each simulation and inserting them to the thread group.

```
⟨ SimResults::simulate code2 251 ⟩ ≡
  void SimResults::simulate(int num_sim, const DecisionRule &dr, const Vector &start, const
          TwoDMatrix &vcov)
  {
    std::vector⟨RandomShockRealization⟩ rsrs;
    rsrs.reserve(num_sim);
    THREAD_GROUP gr;
    for (int i = 0; i < num_sim; i++) {
      RandomShockRealization sr(vcov, system_random_generator.int_uniform());
      rsrs.push_back(sr);
      THREAD * worker = new SimulationWorker(*this, dr, DecisionRule::horner,
          num_per + num_burn, start, rsrs.back());
      gr.insert(worker);
    }
    gr.run();
  }
```

This code is used in section 246.

252. This adds the data with the realized shocks. It takes only periods which are not to be burnt. If the data is not finite, the both data and shocks are thrown away.

```
⟨ SimResults::addDataSet code 252 ⟩ ≡
  bool SimResults::addDataSet(TwoDMatrix *d, ExplicitShockRealization *sr)
  {
    KORD_RAISE_IF(d→nrows() ≠ num_y,
        "Incompatible␣number␣of␣rows␣for␣SimResults::addDataSets");
    KORD_RAISE_IF(d→ncols() ≠ num_per + num_burn,
        "Incompatible␣number␣of␣cols␣for␣SimResults::addDataSets");
    bool ret = false;
    if (d→isFinite()) {
      data.push_back(new TwoDMatrix((const TwoDMatrix &)(*d), num_burn, num_per));
      shocks.push_back(new ExplicitShockRealization(ConstTwoDMatrix(sr→getShocks(),
          num_burn, num_per)));
      ret = true;
    }
    delete d;
    delete sr;
    return ret;
  }
```

This code is used in section 246.

253.

```
⟨ SimResults::writeMat code1 253 ⟩ ≡
  void SimResults::writeMat(const char *base, const char *lname) const
  {
    char matfile_name[100];
    sprintf(matfile_name, "%s.mat", base);
    mat_t * matfd = Mat_Create(matfile_name, Λ);
    if (matfd ≠ Λ) {
      writeMat(matfd, lname);
      Mat_Close(matfd);
    }
  }
```

This code is used in section 246.

254. This save the results as matrices with given prefix and with index appended. If there is only one matrix, the index is not appended.

```
⟨ SimResults::writeMat code2 254 ⟩ ≡
  void SimResults::writeMat(mat_t * fd, const char *lname) const
  {
    char tmp[100];
    for (int i = 0; i < getNumSets(); i++) {
      if (getNumSets() > 1) sprintf(tmp, "%s_data%d", lname, i + 1);
      else sprintf(tmp, "%s_data", lname);
      ConstTwoDMatrix m(*(data[i]));
      m.writeMat(fd, tmp);
    }
  }
```

This code is used in section 246.

255.

```
⟨ SimResultsStats::simulate code 255 ⟩ ≡
  void SimResultsStats::simulate(int num_sim, const DecisionRule &dr, const Vector
        &start, const TwoDMatrix &vcov, Journal &journal)
  {
    SimResults::simulate(num_sim, dr, start, vcov, journal);
    {
      JournalRecordPair paa(journal);
      paa ≪ "Calculating␣means␣from␣the␣simulations." ≪ endrec;
      calcMean();
    }
    {
      JournalRecordPair paa(journal);
      paa ≪ "Calculating␣covariances␣from␣the␣simulations." ≪ endrec;
      calcVcov();
    }
  }
```

This code is used in section 246.

256. Here we do not save the data itself, we save only mean and vcov.

```
⟨ SimResultsStats::writeMat code 256 ⟩ ≡
  void SimResultsStats::writeMat(mat_t *fd, const char *lname) const
  {
    char tmp[100];
    sprintf(tmp, "%s_mean", lname);
    ConstTwoDMatrix m(num_y, 1, mean.base());
    m.writeMat(fd, tmp);
    sprintf(tmp, "%s_vcov", lname);
    ConstTwoDMatrix(vcov).writeMat(fd, tmp);
  }
```

This code is used in section 246.

257.

```
⟨ SimResultsStats::calcMean code 257 ⟩ ≡
  void SimResultsStats::calcMean()
  {
    mean.zeros();
    if (data.size() * num_per > 0) {
      double mult = 1.0/data.size()/num_per;
      for (unsigned int i = 0; i < data.size(); i++) {
        for (int j = 0; j < num_per; j++) {
          ConstVector col(*data[i], j);
          mean.add(mult, col);
        }
      }
    }
  }
```

This code is used in section 246.

258.

```
⟨ SimResultsStats::calcVcov code 258 ⟩ ≡
  void SimResultsStats::calcVcov()
  {
    if (data.size() * num_per > 1) {
      vcov.zeros();
      double mult = 1.0/(data.size() * num_per - 1);
      for (unsigned int i = 0; i < data.size(); i++) {
        const TwoDMatrix &d = *(data[i]);
        for (int j = 0; j < num_per; j++) {
          for (int m = 0; m < num_y; m++) {
            for (int n = m; n < num_y; n++) {
              double s = (d.get(m, j) - mean[m]) * (d.get(n, j) - mean[n]);
              vcov.get(m, n) += mult * s;
              if (m ≠ n) vcov.get(n, m) += mult * s;
            }
          }
        }
      }
    }
    else {
      vcov.infs();
    }
  }
```

This code is used in section 246.

259.

```
⟨ SimResultsDynamicStats::simulate code 259 ⟩ ≡
  void SimResultsDynamicStats::simulate(int num_sim, const DecisionRule &dr, const Vector
        &start, const TwoDMatrix &vcov, Journal &journal)
  {
    SimResults::simulate(num_sim, dr, start, vcov, journal);
    {
      JournalRecordPair paa(journal);
      paa ≪ "Calculating␣means␣of␣the␣conditional␣simulations." ≪ endrec;
      calcMean();
    }
    {
      JournalRecordPair paa(journal);
      paa ≪ "Calculating␣variances␣of␣the␣conditional␣simulations." ≪ endrec;
      calcVariance();
    }
  }
```

This code is used in section 246.

260.

```
⟨ SimResultsDynamicStats::writeMat code 260 ⟩ ≡
  void SimResultsDynamicStats::writeMat(mat_t *fd, const char *lname) const
  {
    char tmp[100];
    sprintf(tmp, "%s_cond_mean", lname);
    ConstTwoDMatrix(mean).writeMat(fd, tmp);
    sprintf(tmp, "%s_cond_variance", lname);
    ConstTwoDMatrix(variance).writeMat(fd, tmp);
  }
```

This code is used in section 246.

261.

```
⟨ SimResultsDynamicStats::calcMean code 261 ⟩ ≡
  void SimResultsDynamicStats::calcMean()
  {
    mean.zeros();
    if (data.size() > 0) {
      double mult = 1.0/data.size();
      for (int j = 0; j < num_per; j++) {
        Vector meanj(mean, j);
        for (unsigned int i = 0; i < data.size(); i++) {
          ConstVector col(*data[i], j);
          meanj.add(mult, col);
        }
      }
    }
  }
```

This code is used in section 246.

262.

```
⟨ SimResultsDynamicStats::calcVariance code 262 ⟩ ≡
  void SimResultsDynamicStats::calcVariance()
  {
    if (data.size() > 1) {
      variance.zeros();
      double mult = 1.0/(data.size() - 1);
      for (int j = 0; j < num_per; j++) {
        ConstVector meanj(mean, j);
        Vector varj(variance, j);
        for (int i = 0; i < (int) data.size(); i++) {
          Vector col(ConstVector((*data[i]), j));
          col.add(-1.0, meanj);
          for (int k = 0; k < col.length(); k++) col[k] = col[k] * col[k];
          varj.add(mult, col);
        }
      }
    }
    else {
      variance.infs();
    }
  }
```

This code is used in section 246.

263.

```
⟨ SimResultsIRF::simulate code1 263 ⟩ ≡
  void SimResultsIRF::simulate(const DecisionRule &dr, Journal &journal)
  {
    JournalRecordPair paa(journal);
    paa << "Performing " << control.getNumSets() << " IRF simulations for " << num_per <<
        " periods; shock=" << ishock << ", impulse=" << imp << endrec;
    simulate(dr);
    int thrown = control.getNumSets() - data.size();
    if (thrown > 0) {
      JournalRecord rec(journal);
      rec << "I had to throw " << thrown << " simulations away due to Nan or Inf" << endrec;
    }
    calcMeans();
    calcVariances();
  }
```

This code is used in section 246.

264.

```
⟨ SimResultsIRF::simulate code2 264 ⟩ ≡
  void SimResultsIRF::simulate(const DecisionRule &dr)
  {
    THREAD_GROUP gr;
    for (int idata = 0; idata < control.getNumSets(); idata++) {
      THREAD * worker = new SimulationIRFWorker(*this, dr, DecisionRule::horner, num_per,
          idata, ishock, imp);
      gr.insert(worker);
    }
    gr.run();
  }
```

This code is used in section 246.

265.

```
⟨ SimResultsIRF::calcMeans code 265 ⟩ ≡
  void SimResultsIRF::calcMeans()
  {
    means.zeros();
    if (data.size() > 0) {
      for (unsigned int i = 0; i < data.size(); i++) means.add(1.0, *(data[i]));
      means.mult(1.0/data.size());
    }
  }
```

This code is used in section 246.

266.

```
⟨ SimResultsIRF::calcVariances code 266 ⟩ ≡
  void SimResultsIRF::calcVariances()
  {
    if (data.size() > 1) {
      variances.zeros();
      for (unsigned int i = 0; i < data.size(); i++) {
        TwoDMatrix d((const TwoDMatrix &)(*(data[i])));
        d.add(-1.0, means);
        for (int j = 0; j < d.nrows(); j++)
          for (int k = 0; k < d.ncols(); k++) variances.get(j, k) += d.get(j, k) * d.get(j, k);
        d.mult(1.0/(data.size() - 1));
      }
    }
    else {
      variances.infs();
    }
  }
```

This code is used in section 246.

267.

```
⟨ SimResultsIRF::writeMat code 267 ⟩ ≡
  void SimResultsIRF::writeMat(mat_t * fd, const char *lname) const
  {
    char tmp[100];
    sprintf(tmp, "%s_mean", lname);
    means.writeMat(fd, tmp);
    sprintf(tmp, "%s_var", lname);
    variances.writeMat(fd, tmp);
  }
```

This code is used in section 246.

268.

```
⟨ RTSimResultsStats::simulate code1 268 ⟩ ≡
  void RTSimResultsStats::simulate(int num_sim, const DecisionRule &dr, const Vector
          &start, const TwoDMatrix &v, Journal &journal)
  {
    JournalRecordPair paa(journal);
    paa << "Performing " << num_sim << " real-time stochastic simulations for " << num_per <<
        " periods" << endrec;
    simulate(num_sim, dr, start, v);
    mean = nc.getMean();
    mean.add(1.0, dr.getSteady());
    nc.getVariance(vcov);
    if (thrown_periods > 0) {
      JournalRecord rec(journal);
      rec << "I had to throw " << thrown_periods << " periods away due to Nan or Inf" << endrec;
      JournalRecord rec1(journal);
      rec1 << "This affected " << incomplete_simulations << " out of " << num_sim <<
          " simulations" << endrec;
    }
  }
```

This code is used in section 246.

269.

```
⟨ RTSimResultsStats::simulate code2 269 ⟩ ≡
  void RTSimResultsStats::simulate(int num_sim, const DecisionRule &dr, const Vector
          &start, const TwoDMatrix &vcov)
  {
    std::vector⟨RandomShockRealization⟩ rsrs;
    rsrs.reserve(num_sim);
    THREAD_GROUP gr;
    for (int i = 0; i < num_sim; i++) {
      RandomShockRealization sr(vcov, system_random_generator.int_uniform());
      rsrs.push_back(sr);
      THREAD * worker = new RTSimulationWorker(*this, dr, DecisionRule::horner, num_per, start,
          rsrs.back());
      gr.insert(worker);
    }
    gr.run();
  }
```

This code is used in section 246.

270.

```
⟨ RTSimResultsStats::writeMat code 270 ⟩ ≡
  void RTSimResultsStats::writeMat(mat_t *fd, const char *lname)
  {
    char tmp[100];
    sprintf(tmp, "%s_rt_mean", lname);
    ConstTwoDMatrix m(nc.getDim(), 1, mean.base());
    m.writeMat(fd, tmp);
    sprintf(tmp, "%s_rt_vcov", lname);
    ConstTwoDMatrix(vcov).writeMat(fd, tmp);
  }
```

This code is used in section 246.

271.

```
⟨ IRFResults constructor 271 ⟩ ≡
  IRFResults::IRFResults(const DynamicModel &mod, const DecisionRule &dr, const
        SimResults &control, const vector⟨int⟩ &ili, Journal &journal)
  : model(mod), irf_list_ind(ili) {
    int num_per = control.getNumPer();
    JournalRecordPair pa(journal);
    pa << "Calculating␣IRFs␣against␣control␣for␣" << (int)
       irf_list_ind.size() << "␣shocks␣and␣for␣" << num_per << "␣periods" << endrec;
    const TwoDMatrix &vcov = mod.getVcov();
    for (unsigned int ii = 0; ii < irf_list_ind.size(); ii++) {
      int ishock = irf_list_ind[ii];
      double stderror = sqrt(vcov.get(ishock, ishock));
      irf_res.push_back(new SimResultsIRF(control, model.numeq(), num_per, ishock, stderror));
      irf_res.push_back(new SimResultsIRF(control, model.numeq(), num_per, ishock, -stderror));
    }
    for (unsigned int ii = 0; ii < irf_list_ind.size(); ii++) {
      irf_res[2 * ii]->simulate(dr, journal);
      irf_res[2 * ii + 1]->simulate(dr, journal);
    }
  }
```

This code is used in section 246.

272.

```
⟨ IRFResults destructor 272 ⟩ ≡
  IRFResults::~IRFResults()
  {
    for (unsigned int i = 0; i < irf_res.size(); i++) delete irf_res[i];
  }
```

This code is used in section 246.

273.

```
⟨ IRFResults::writeMat code 273 ⟩ ≡
  void IRFResults::writeMat(mat_t *fd, const char *prefix) const
  {
    for (unsigned int i = 0; i < irf_list_ind.size(); i++) {
      char tmp[100];
      int ishock = irf_list_ind[i];
      const char *shockname = model.getExogNames().getName(ishock);
      sprintf(tmp, "%s_irfp_%s", prefix, shockname);
      irf_res[2 * i]->writeMat(fd, tmp);
      sprintf(tmp, "%s_irfm_%s", prefix, shockname);
      irf_res[2 * i + 1]->writeMat(fd, tmp);
    }
  }
```

This code is used in section 246.

274.

```
⟨ SimulationWorker::operator()() code 274 ⟩ ≡
  void SimulationWorker::operator()()
  {
    ExplicitShockRealization *esr = new ExplicitShockRealization(sr, np);
    TwoDMatrix *m = dr.simulate(em, np, st, *esr);
    {
      SYNCHRO syn(&res, "simulation");
      res.addDataSet(m, esr);
    }
  }
```

This code is used in section 246.

275. Here we create a new instance of **ExplicitShockRealization** of the corresponding control, add the impulse, and simulate.

```
⟨ SimulationIRFWorker::operator()() code 275 ⟩ ≡
  void SimulationIRFWorker::operator()()
  {
    ExplicitShockRealization *esr = new ExplicitShockRealization(res.control.getShocks(idata));
    esr→addToShock(ishock, 0, imp);
    const TwoDMatrix &data = res.control.getData(idata);
    ConstVector st(data, res.control.getNumBurn());
    TwoDMatrix *m = dr.simulate(em, np, st, *esr);
    m→add(-1.0, res.control.getData(idata));
    {
      SYNCHRO syn(&res, "simulation");
      res.addDataSet(m, esr);
    }
  }
```

This code is used in section 246.

276.

```
⟨ RTSimulationWorker::operator()() code 276 ⟩ ≡
  void RTSimulationWorker::operator()()
  {
    NormalConj nc(res.nc.getDim());
    const PartitionY &ypart = dr.getYPart();
    int nu = dr.nexog();
    const Vector &ysteady = dr.getSteady();
    ⟨ initialize vectors and subvectors for simulation 216 ⟩;
    ⟨ simulate the first real-time period 278 ⟩;
    ⟨ simulate other real-time periods 279 ⟩;
    {
      SYNCHRO syn(&res, "rtsimulation");
      res.nc.update(nc);
      if (res.num_per − ip > 0) {
        res.incomplete_simulations++;
        res.thrown_periods += res.num_per − ip;
      }
    }
  }
```

This code is used in section 246.

277.

```
⟨ initialize vectors and subvectors for simulation 216 ⟩ +≡
  Vector dyu(ypart.nys() + nu);
  ConstVector ystart_pred(ystart, ypart.nstat, ypart.nys());
  ConstVector ysteady_pred(ysteady, ypart.nstat, ypart.nys());
  Vector dy(dyu, 0, ypart.nys());
  Vector u(dyu, ypart.nys(), nu);
  Vector y(nc.getDim());
  ConstVector ypred(y, ypart.nstat, ypart.nys());
```

278.

```
⟨ simulate the first real-time period 278 ⟩ ≡
  int ip = 0;
  dy = ystart_pred;
  dy.add(−1.0, ysteady_pred);
  sr.get(ip, u);
  dr.eval(em, y, dyu);
  if (ip ≥ res.num_burn) nc.update(y);
```

This code is used in section 276.

279.

```
⟨ simulate other real-time periods 279 ⟩ ≡
  while (y.isFinite() ∧ ip < res.num_burn + res.num_per) {
    ip++;
    dy = ypred;
    sr.get(ip, u);
    dr.eval(em, y, dyu);
    if (ip ≥ res.num_burn) nc.update(y);
  }
```

This code is used in section 276.

280. This calculates factorization $FF^T = V$ in the Cholesky way. It does not work for semidefinite matrices.

```
⟨ RandomShockRealization::choleskyFactor code 280 ⟩ ≡
  void RandomShockRealization::choleskyFactor(const TwoDMatrix &v)
  {
    factor = v;
    lapack_int rows = factor.nrows();
    for (int i = 0; i < rows; i++)
      for (int j = i + 1; j < rows; j++) factor.get(i, j) = 0.0;
    lapack_int info;
    dpotrf("L", &rows, factor.base(), &rows, &info);
    KORD_RAISE_IF(info ≠ 0, "Info!=0 in RandomShockRealization::choleskyFactor");
  }
```

This code is used in section 246.

281. This calculates $FF^T = V$ factorization by symmetric Schur decomposition. It works for semidifinite matrices.

```
⟨ RandomShockRealization::schurFactor code 281 ⟩ ≡
  void RandomShockRealization::schurFactor(const TwoDMatrix &v)
  {
    SymSchurDecomp ssd(v);
    ssd.getFactor(factor);
  }
```

This code is used in section 246.

282.

```
⟨ RandomShockRealization::get code 282 ⟩ ≡
  void RandomShockRealization::get(int n, Vector &out)
  {
    KORD_RAISE_IF(out.length() ≠ numShocks(),
        "Wrong length of out vector in RandomShockRealization::get");
    Vector d(out.length());
    for (int i = 0; i < d.length(); i++) {
      d[i] = mtwister.normal();
    }
    out.zeros();
    factor.multaVec(out, ConstVector(d));
  }
```

This code is used in section 246.

283.

```
⟨ ExplicitShockRealization constructor code 283 ⟩ ≡
  ExplicitShockRealization::ExplicitShockRealization(ShockRealization &sr, int num_per)
  : shocks(sr.numShocks(), num_per) {
    for (int j = 0; j < num_per; j++) {
      Vector jcol(shocks, j);
      sr.get(j, jcol);
    }
  }
```

This code is used in section 246.

284.

```
⟨ ExplicitShockRealization::get code 284 ⟩ ≡
  void ExplicitShockRealization::get(int n, Vector &out)
  {
    KORD_RAISE_IF(out.length() ≠ numShocks(),
        "Wrong length of out vector in ExplicitShockRealization::get");
    int i = n % shocks.ncols();
    ConstVector icol(shocks, i);
    out = icol;
  }
```

This code is used in section 246.

285.

```
⟨ ExplicitShockRealization::addToShock code 285 ⟩ ≡
  void ExplicitShockRealization::addToShock(int ishock, int iper, double val)
  {
    KORD_RAISE_IF(ishock < 0 ∨ ishock > numShocks(),
        "Wrong index of shock in ExplicitShockRealization::addToShock");
    int j = iper % shocks.ncols();
    shocks.get(ishock, j) += val;
  }
```

This code is used in section 246.

286.

```
⟨ GenShockRealization::get code 286 ⟩ ≡
  void GenShockRealization::get(int n, Vector &out)
  {
    KORD_RAISE_IF(out.length() ≠ numShocks(),
        "Wrong length of out vector in GenShockRealization::get");
    ExplicitShockRealization::get(n, out);
    Vector r(numShocks());
    RandomShockRealization::get(n, r);
    for (int j = 0; j < numShocks(); j++)
      if (¬isfinite(out[j])) out[j] = r[j];
  }
```

This code is used in section 246.

287. End of `decision_rule.cpp` file.

288. Global check. Start of `global_check.h` file.

The purpose of this file is to provide classes for checking error of approximation. If $y_t = g(y^*_{t-1}, u)$ is an approximate solution, then we check for the error of residuals of the system equations. Let $F(y^*, u, u') = f(g^{**}(g^*(y^*, u'), u), g(y^*, u), y^*, u)$, then we calculate integral

$$E[F(y^*, u, u')]$$

which we want to be zero for all y^*, and u.

There are a few possibilities how and where the integral is evaluated. Currently we offer the following:

1) Along shocks. The y^* is set to steady state, and u is set to zero but one element is going from minus through plus shocks in few steps. The user gives the scaling factor, for instance interval $\langle < -3\sigma, 3\sigma \rangle$ (where *sigma* is a standard error of the shock), and a number of steps. This is repeated for each shock (element of the u vector).
2) Along simulation. Some random simulation is run, and for each realization of y^* and u along the path we evaluate the residual.
3) On ellipse. Let $V = AA^T$ be a covariance matrix of the predetermined variables y^* based on linear approximation, then we calculate integral for points on the ellipse $\{Ax | \, \|x\|_2 = 1\}$. The points are selected by means of low discrepancy method and polar transformation. The shock u are zeros.
4) Unconditional distribution.

```
#ifndef GLOBAL_CHECK_H
#define GLOBAL_CHECK_H
#include <matio.h>
#include "vector_function.h"
#include "quadrature.h"
#include "dynamic_model.h"
#include "journal.h"
#include "approximation.h"
  ⟨ ResidFunction class declaration 289 ⟩;
  ⟨ GResidFunction class declaration 290 ⟩;
  ⟨ GlobalChecker class declaration 291 ⟩;
  ⟨ ResidFunctionSig class declaration 292 ⟩;
#endif
```

289. This is a class for implementing **VectorFunction** interface evaluating the residual of equations, this is

$$F(y^*, u, u') = f(g^{**}(g^*(y^*, u), u'), y^*, u)$$

is written as a function of u'.

When the object is constructed, one has to specify (y^*, u), this is done by *setYU* method. The object has basically two states. One is after construction and before call to *setYU*. The second is after call *setYU*. We distinguish between the two states, an object in the second state contains *yplus*, *ystar*, *u*, and *hss*.

The vector *yplus* is $g^*(y^*, u)$. *ystar* is y^*, and polynomial *hss* is partially evaluated $g^* * (yplus, u)$.

The pointer to **DynamicModel** is important, since the **DynamicModel** evaluates the function f. When copying the object, we have to make also a copy of **DynamicModel**.

⟨ **ResidFunction** class declaration 289 ⟩ ≡

```
class ResidFunction : public VectorFunction {
protected:
  const Approximation &approx;
  DynamicModel *model;
  Vector *yplus;
  Vector *ystar;
  Vector *u;
  FTensorPolynomial *hss;
public:
  ResidFunction(const Approximation &app);
  ResidFunction(const ResidFunction &rf);
  virtual ~ResidFunction();
  virtual VectorFunction *clone() const
  { return new ResidFunction(*this); }
  virtual void eval(const Vector &point, const ParameterSignal &sig, Vector &out);
  void setYU(const Vector &ys, const Vector &xx);
};
```

This code is used in section 288.

290. This is a **ResidFunction** wrapped with *GaussConverterFunction*.

⟨ **GResidFunction** class declaration 290 ⟩ ≡

```
class GResidFunction : public GaussConverterFunction
{
public:
  GResidFunction(const Approximation &app)
  : GaussConverterFunction(new ResidFunction(app), app.getModel().getVcov()) { }
  GResidFunction(const GResidFunction &rf)
  : GaussConverterFunction(rf) { }
  virtual ~GResidFunction() { }
  virtual VectorFunction *clone() const
  { return new GResidFunction(*this); }
  void setYU(const Vector &ys, const Vector &xx)
  { ((ResidFunction *) func)->setYU(ys, xx);
  }
}
;
```

This code is used in section 288.

291. This is a class encapsulating checking algorithms. Its core routine is *check*, which calculates integral $E[F(y^*, u, u')|y^*, u]$ for given realizations of y^* and u. The both are given in matrices. The methods checking along shocks, on ellipse and anlong a simulation path, just fill the matrices and call the core *check*.

The method *checkUnconditionalAndSave* evaluates unconditional $E[F(y, u, u')]$.

The object also maintains a set of **GResidFunction** functions *vfs* in order to save (possibly expensive) copying of **DynamicModel**s.

⟨ **GlobalChecker** class declaration 291 ⟩ ≡
```
class GlobalChecker {
  const Approximation &approx;
  const DynamicModel &model;
  Journal &journal;
  GResidFunction rf;
  VectorFunctionSet vfs;
public:
  GlobalChecker(const Approximation &app, int n, Journal &jr)
  : approx(app), model(approx.getModel()), journal(jr), rf(approx), vfs(rf, n) { }
  void check(int max_evals, const ConstTwoDMatrix &y, const ConstTwoDMatrix
      &x, TwoDMatrix &out);
  void checkAlongShocksAndSave(mat_t *fd, const char *prefix, int m, double mult, int max_evals);
  void checkOnEllipseAndSave(mat_t *fd, const char *prefix, int m, double mult, int max_evals);
  void checkAlongSimulationAndSave(mat_t *fd, const char *prefix, int m, int max_evals);
  void checkUnconditionalAndSave(mat_t *fd, const char *prefix, int m, int max_evals);
protected:
  void check(const Quadrature &quad, int level, const ConstVector &y, const ConstVector
      &x, Vector &out);
};
```
This code is used in section 288.

292. Signalled resid function. Not implemented yet. todo:

⟨ **ResidFunctionSig** class declaration 292 ⟩ ≡
```
class ResidFunctionSig : public ResidFunction {
public:
  ResidFunctionSig(const Approximation &app, const Vector &ys, const Vector &xx);
};
```
This code is used in section 288.

293. End of `global_check.h` file.

294. Start of `global_check.cpp` file.

```
#include "SymSchurDecomp.h"
#include "global_check.h"
#include "smolyak.h"
#include "product.h"
#include "quasi_mcarlo.h"
#ifdef __MINGW32__
#define __CROSS_COMPILATION__
#endif
#ifdef __MINGW64__
#define __CROSS_COMPILATION__
#endif
#ifdef __CROSS_COMPILATION__
#define M_PI 3.14159265358979323846
#endif
```

⟨ **ResidFunction** constructor code 295 ⟩;
⟨ **ResidFunction** copy constructor code 296 ⟩;
⟨ **ResidFunction** destructor code 297 ⟩;
⟨ **ResidFunction**::*setYU* code 299 ⟩;
⟨ **ResidFunction**::*eval* code 303 ⟩;
⟨ **GlobalChecker**::*check* vector code 304 ⟩;
⟨ **GlobalChecker**::*check* matrix code 305 ⟩;
⟨ **GlobalChecker**::*checkAlongShocksAndSave* code 309 ⟩;
⟨ **GlobalChecker**::*checkOnEllipseAndSave* code 313 ⟩;
⟨ **GlobalChecker**::*checkAlongSimulationAndSave* code 318 ⟩;

295. Here we just set a reference to the approximation, and create a new **DynamicModel**.

⟨ **ResidFunction** constructor code 295 ⟩ ≡

```
ResidFunction::ResidFunction(const Approximation &app)
  : VectorFunction(app.getModel().nexog(), app.getModel().numeq()), approx(app),
    model(app.getModel().clone()), yplus(Λ), ystar(Λ), u(Λ), hss(Λ) { }
```

This code is used in section 294.

296.

⟨ **ResidFunction** copy constructor code 296 ⟩ ≡

```
ResidFunction::ResidFunction(const ResidFunction &rf)
  : VectorFunction(rf), approx(rf.approx), model(rf.model→clone()), yplus(Λ), ystar(Λ), u(Λ),
      hss(Λ) {
  if (rf.yplus) yplus = new Vector(*(rf.yplus));
  if (rf.ystar) ystar = new Vector(*(rf.ystar));
  if (rf.u) u = new Vector(*(rf.u));
  if (rf.hss) hss = new FTensorPolynomial(*(rf.hss));
}
```

This code is used in section 294.

297.

```
⟨ ResidFunction destructor code 297 ⟩ ≡
  ResidFunction::~ResidFunction()
  {
    delete model;
    ⟨ delete y and u dependent data 298 ⟩;
  }
```

This code is used in section 294.

298.

```
⟨ delete y and u dependent data 298 ⟩ ≡
  if (yplus) delete yplus;
  if (ystar) delete ystar;
  if (u) delete u;
  if (hss) delete hss;
```

This code is used in sections 297 and 299.

299. This sets y^* and u. We have to create *ystar*, *u*, *yplus* and *hss*.

```
⟨ ResidFunction::setYU code 299 ⟩ ≡
  void ResidFunction::setYU(const Vector &ys, const Vector &xx)
  {
    ⟨ delete y and u dependent data 298 ⟩;
    ystar = new Vector(ys);
    u = new Vector(xx);
    yplus = new Vector(model→numeq());
    approx.getFoldDecisionRule().evaluate(DecisionRule::horner, *yplus, *ystar, *u);
    ⟨ make a tensor polynomial of in-place subtensors from decision rule 300 ⟩;
    ⟨ make ytmp_star be a difference of yplus from steady 301 ⟩;
    ⟨ make hss and add steady to it 302 ⟩;
  }
```

This code is used in section 294.

300. Here we use a dirty tricky of converting **const** to non-**const** to obtain a polynomial of subtensor corresponding to non-predetermined variables. However, this new non-**const** polynomial will be used for a construction of *hss* and will be used in **const** context. So this dirty thing is safe.

Note, that there is always a folded decision rule in **Approximation**.

```
⟨ make a tensor polynomial of in-place subtensors from decision rule 300 ⟩ ≡
  union {
    const FoldDecisionRule *c;
    FoldDecisionRule *n;
  } dr;
  dr.c = &(approx.getFoldDecisionRule());
  FTensorPolynomial dr_ss(model→nstat() + model→npred(), model→nboth() + model→nforw(), *(dr.n));
```

This code is used in section 299.

301.

⟨ make *ytmp_star* be a difference of *yplus* from steady 301 ⟩ ≡
```
Vector ytmp_star(ConstVector(*yplus, model->nstat( ), model->npred( ) + model->nboth( )));
ConstVector ysteady_star(dr.c->getSteady( ), model->nstat( ), model->npred( ) + model->nboth( ));
ytmp_star.add(-1.0, ysteady_star);
```
This code is used in section 299.

302. Here is the **const** context of *dr_ss*.

⟨ make *hss* and add steady to it 302 ⟩ ≡
```
hss = new FTensorPolynomial(dr_ss, ytmp_star);
ConstVector ysteady_ss(dr.c->getSteady( ), model->nstat( ) + model->npred( ),
    model->nboth( ) + model->nforw( ));
if (hss->check(Symmetry(0))) {
  hss->get(Symmetry(0))->getData( ).add(1.0, ysteady_ss);
}
else {
  FFSTensor *ten = new FFSTensor(hss->nrows( ), hss->nvars( ), 0);
  ten->getData( ) = ysteady_ss;
  hss->insert(ten);
}
```
This code is used in section 299.

303. Here we evaluate the residual $F(y^*, u, u')$. We have to evaluate *hss* for u' =*point* and then we evaluate the system f.

⟨ **ResidFunction**::*eval* code 303 ⟩ ≡
```
void ResidFunction::eval(const Vector &point, const ParameterSignal &sig, Vector &out)
{
  KORD_RAISE_IF(point.length( ) ≠ hss->nvars( ),
      "Wrong dimension of input vector in ResidFunction::eval");
  KORD_RAISE_IF(out.length( ) ≠ model->numeq( ),
      "Wrong dimension of output vector in ResidFunction::eval");
  Vector yss(hss->nrows( ));
  hss->evalHorner(yss, point);
  model->evaluateSystem(out, *ystar, *yplus, yss, *u);
}
```
This code is used in section 294.

304. This checks the $E[F(y^*, u, u')]$ for a given y^* and u by integrating with a given quadrature. Note that the input *ys* is y^* not whole y.

⟨ **GlobalChecker**::*check* vector code 304 ⟩ ≡
```
void GlobalChecker::check(const Quadrature &quad, int level, const ConstVector &ys, const
        ConstVector &x, Vector &out)
{
  for (int ifunc = 0; ifunc < vfs.getNum( ); ifunc++)
    ((GResidFunction &)(vfs.getFunc(ifunc))).setYU(ys, x);
  quad.integrate(vfs, level, out);
}
```
This code is cited in section 305.
This code is used in section 294.

305. This method is a bulk version of ⟨ **GlobalChecker** :: *check* vector code 304 ⟩. It decides between Smolyak and product quadrature according to *max_evals* constraint.

Note that y can be either full (all endogenous variables including static and forward looking), or just y^* (state variables). The method is able to recognize it.

```
⟨ GlobalChecker :: check matrix code 305 ⟩ ≡
  void GlobalChecker::check(int max_evals, const ConstTwoDMatrix &y, const
          ConstTwoDMatrix &x, TwoDMatrix &out)
  {
    JournalRecordPair pa(journal);
    pa ≪ "Checking␣approximation␣error␣for␣" ≪ y.ncols() ≪ "␣states␣with␣at␣most␣" ≪
        max_evals ≪ "␣evaluations" ≪ endrec;
    ⟨ decide about type of quadrature 306 ⟩;
    Quadrature *quad;
    int lev;
    ⟨ create the quadrature and report the decision 307 ⟩;
    ⟨ check all column of y and x 308 ⟩;
    delete quad;
  }
```

This code is used in section 294.

306.

```
⟨ decide about type of quadrature 306 ⟩ ≡
  GaussHermite gh;
  SmolyakQuadrature dummy_sq(model.nexog(), 1, gh);
  int smol_evals;
  int smol_level;
  dummy_sq.designLevelForEvals(max_evals, smol_level, smol_evals);
  ProductQuadrature dummy_pq(model.nexog(), gh);
  int prod_evals;
  int prod_level;
  dummy_pq.designLevelForEvals(max_evals, prod_level, prod_evals);
  bool take_smolyak = (smol_evals < prod_evals) ∧ (smol_level ≥ prod_level − 1);
```

This code is used in section 305.

307.

⟨ create the quadrature and report the decision 307 ⟩ ≡

```
if (take_smolyak) {
  quad = new SmolyakQuadrature(model.nexog(), smol_level, gh);
  lev = smol_level;
  JournalRecord rec(journal);
  rec << "Selected Smolyak (level,evals)=(" << smol_level << "," << smol_evals <<
      ") over product (" << prod_level << "," << prod_evals << ")" << endrec;
}
else {
  quad = new ProductQuadrature(model.nexog(), gh);
  lev = prod_level;
  JournalRecord rec(journal);
  rec << "Selected product (level,evals)=(" << prod_level << "," << prod_evals <<
      ") over Smolyak (" << smol_level << "," << smol_evals << ")" << endrec;
}
```

This code is used in section 305.

308.

⟨ check all column of y and x 308 ⟩ ≡

```
int first_row = (y.nrows() ≡ model.numeq()) ? model.nstat() : 0;
ConstTwoDMatrix ysmat(y, first_row, 0, model.npred() + model.nboth(), y.ncols());
for (int j = 0; j < y.ncols(); j++) {
  ConstVector yj(ysmat, j);
  ConstVector xj(x, j);
  Vector outj(out, j);
  check(*quad, lev, yj, xj, outj);
}
```

This code is used in section 305.

309. This method checks an error of the approximation by evaluating residual $E[F(y^*, u, u')|y^*, u]$ for y^* being the steady state, and changing u. We go through all elements of u and vary them from $-mult \cdot \sigma$ to $mult \cdot \sigma$ in m steps.

⟨ **GlobalChecker**::*checkAlongShocksAndSave* code 309 ⟩ ≡

```
void GlobalChecker::checkAlongShocksAndSave(mat_t *fd, const char *prefix, int m, double
        mult, int max_evals)
{
  JournalRecordPair pa(journal);
  pa << "Calculating errors along shocks +/- " << mult << " std errors, granularity " <<
      m << endrec;
  ⟨ setup y_mat of steady states for checking 310 ⟩;
  ⟨ setup exo_mat for checking 311 ⟩;
  TwoDMatrix errors(model.numeq(), 2 * m * model.nexog() + 1);
  check(max_evals, y_mat, exo_mat, errors);
  ⟨ report errors along shock and save them 312 ⟩;
}
```

This code is used in section 294.

310.

⟨ setup *y_mat* of steady states for checking 310 ⟩ ≡

```
TwoDMatrix y_mat(model.numeq(), 2 * m * model.nexog() + 1);
for (int j = 0; j < 2 * m * model.nexog() + 1; j++) {
  Vector yj(y_mat, j);
  yj = (const Vector &) model.getSteady();
}
```

This code is used in section 309.

311.

⟨ setup *exo_mat* for checking 311 ⟩ ≡

```
TwoDMatrix exo_mat(model.nexog(), 2 * m * model.nexog() + 1);
exo_mat.zeros();
for (int ishock = 0; ishock < model.nexog(); ishock++) {
  double max_sigma = sqrt(model.getVcov().get(ishock, ishock));
  for (int j = 0; j < 2 * m; j++) {
    int jmult = (j < m) ? j - m : j - m + 1;
    exo_mat.get(ishock, 1 + 2 * m * ishock + j) = mult * jmult * max_sigma/m;
  }
}
```

This code is used in section 309.

312.

⟨ report errors along shock and save them 312 ⟩ ≡

```
TwoDMatrix res(model.nexog(), 2 * m + 1);
JournalRecord rec(journal);
rec ≪ "Shock␣␣␣␣value␣␣␣␣␣␣␣␣error" ≪ endrec;
ConstVector err0(errors, 0);
char shock[9];
char erbuf[17]; for (int ishock = 0; ishock < model.nexog(); ishock++) { TwoDMatrix
    err_out(model.numeq(), 2 * m + 1);
sprintf(shock, "%-8s", model.getExogNames().getName(ishock)); for (int j = 0; j < 2 * m + 1; j++) {
    int jj; Vector
error (err_out, j) ;
if (j ≠ m) {
if (j < m) jj = 1 + 2 * m * ishock + j;
else jj = 1 + 2 * m * ishock + j - 1;
ConstVector coljj(errors, jj); error = coljj; } else { jj = 0; error = err0; } JournalRecord
    rec1(journal); sprintf (erbuf, "%12.7g␣␣␣␣", error . getMax() ) ;
rec1 ≪ shock ≪ "␣" ≪ exo_mat.get(ishock, jj) ≪ "\t" ≪ erbuf ≪ endrec; } char tmp[100];
sprintf(tmp, "%s_shock_%s_errors", prefix, model.getExogNames().getName(ishock));
err_out.writeMat(fd, tmp); }
```

This code is used in section 309.

313. This method checks errors on ellipse of endogenous states (predetermined variables). The ellipse is shaped according to covariance matrix of endogenous variables based on the first order approximation and scaled by *mult*. The points on the ellipse are chosen as polar images of the low discrepancy grid in a cube.

The method works as follows: First we calculate symmetric Schur factor of covariance matrix of the states. Second we generate low discrepancy points on the unit sphere. Third we transform the sphere with the variance-covariance matrix factor and multiplier *mult* and initialize matrix of u_t to zeros. Fourth we run the *check* method and save the results.

⟨ **GlobalChecker** :: *checkOnEllipseAndSave* code 313 ⟩ ≡

```
void GlobalChecker::checkOnEllipseAndSave(mat_t *fd, const char *prefix, int m, double mult, int
        max_evals)
{
  JournalRecordPair pa(journal);
  pa << "Calculating errors at " << m << " ellipse points scaled by " << mult << endrec;
  ⟨ make factor of covariance of variables 314 ⟩;
  ⟨ put low discrepancy sphere points to ymat 315 ⟩;
  ⟨ transform sphere ymat and prepare umat for checking 316 ⟩;
  ⟨ check on ellipse and save 317 ⟩;
}
```

This code is used in section 294.

314. Here we set *ycovfac* to the symmetric Schur decomposition factor of a submatrix of covariances of all endogenous variables. The submatrix corresponds to state variables (predetermined plus both).

⟨ make factor of covariance of variables 314 ⟩ ≡

```
TwoDMatrix *ycov = approx.calcYCov();
TwoDMatrix ycovpred((const TwoDMatrix &) *ycov, model.nstat(), model.nstat(),
    model.npred() + model.nboth(), model.npred() + model.nboth());
delete ycov;
SymSchurDecomp ssd(ycovpred);
ssd.correctDefinitness(1.e-05);
TwoDMatrix ycovfac(ycovpred.nrows(), ycovpred.ncols());
KORD_RAISE_IF(¬ssd.isPositiveSemidefinite(), "Covariance matrix of the states not pos\
    itive    semidefinite in GlobalChecker::checkOnEllipseAndSave");
ssd.getFactor(ycovfac);
```

This code is used in section 313.

315. Here we first calculate dimension d of the sphere, which is a number of state variables minus one. We go through the d-dimensional cube $\langle 0,1\rangle^d$ by **QMCarloCubeQuadrature** and make a polar transformation to the sphere. The polar transformation f^i can be written recursively wrt. the dimension i as:

$$f^0() = [1]$$
$$f^i(x_1,\dots,x_i) = \begin{bmatrix} cos(2\pi x_i)\cdot f^{i-1}(x_1,\dots,x_{i-1}) \\ sin(2\pi x_i) \end{bmatrix}$$

⟨ put low discrepancy sphere points to *ymat* 315 ⟩ ≡

```
int d = model.npred() + model.nboth() − 1;
TwoDMatrix ymat(model.npred() + model.nboth(), (d ≡ 0) ? 2 : m);
if (d ≡ 0) {
  ymat.get(0, 0) = 1;
  ymat.get(0, 1) = −1;
}
else {
  int icol = 0;
  ReversePerScheme ps;
  QMCarloCubeQuadrature qmc(d, m, ps);
  qmcpit beg = qmc.start(m);
  qmcpit end = qmc.end(m);
  for (qmcpit run = beg; run ≠ end; ++run, icol++) {
    Vector ycol(ymat, icol);
    Vector x(run.point());
    x.mult(2 ∗ M_PI);
    ycol[0] = 1;
    for (int i = 0; i < d; i++) {
      Vector subsphere(ycol, 0, i + 1);
      subsphere.mult(cos(x[i]));
      ycol[i + 1] = sin(x[i]);
    }
  }
}
```

This code is used in section 313.

316. Here we multiply the sphere points in *ymat* with the Cholesky factor to obtain the ellipse, scale the ellipse by the given *mult*, and initialize matrix of shocks *umat* to zero.

⟨ transform sphere *ymat* and prepare *umat* for checking 316 ⟩ ≡

```
TwoDMatrix umat(model.nexog(), ymat.ncols());
umat.zeros();
ymat.mult(mult);
ymat.multLeft(ycovfac);
ConstVector ys(model.getSteady(), model.nstat(), model.npred() + model.nboth());
for (int icol = 0; icol < ymat.ncols(); icol++) {
  Vector ycol(ymat, icol);
  ycol.add(1.0, ys);
}
```

This code is used in section 313.

317. Here we check the points and save the results to MAT-4 file.

⟨ check on ellipse and save 317 ⟩ ≡
```
TwoDMatrix out(model.numeq(), ymat.ncols());
check(max_evals, ymat, umat, out);
char tmp[100];
sprintf(tmp, "%s_ellipse_points", prefix);
ymat.writeMat(fd, tmp);
sprintf(tmp, "%s_ellipse_errors", prefix);
out.writeMat(fd, tmp);
```
This code is used in section 313.

318. Here we check the errors along a simulation. We simulate, then set x to zeros, check and save results.

⟨ **GlobalChecker**::*checkAlongSimulationAndSave* code 318 ⟩ ≡
```
void GlobalChecker::checkAlongSimulationAndSave(mat_t * fd, const char *prefix, int m, int
        max_evals)
{
  JournalRecordPair pa(journal);
  pa ≪ "Calculating errors at " ≪ m ≪ " simulated points" ≪ endrec;
  RandomShockRealization sr(model.getVcov(), system_random_generator.int_uniform());
  TwoDMatrix *y = approx.getFoldDecisionRule().simulate(DecisionRule::horner, m,
      model.getSteady(), sr);
  TwoDMatrix x(model.nexog(), m);
  x.zeros();
  TwoDMatrix out(model.numeq(), m);
  check(max_evals, *y, x, out);
  char tmp[100];
  sprintf(tmp, "%s_simul_points", prefix);
  y->writeMat(fd, tmp);
  sprintf(tmp, "%s_simul_errors", prefix);
  out.writeMat(fd, tmp);
  delete y;
}
```
This code is used in section 294.

319. End of `global_check.cpp` file.

320. Index.

⟨ **Approximation** :: *calcStochShift* code 202 ⟩ Used in section 186.
⟨ **Approximation** :: *calcYCov* code 206 ⟩ Used in section 186.
⟨ **Approximation** :: *check* code 205 ⟩ Used in section 186.
⟨ **Approximation** :: *getFoldDecisionRule* code 191 ⟩ Used in section 186.
⟨ **Approximation** :: *getUnfoldDecisionRule* code 192 ⟩ Used in section 186.
⟨ **Approximation** :: *saveRuleDerivs* code 201 ⟩ Cited in section 184. Used in section 186.
⟨ **Approximation** :: *walkStochSteady* code 194 ⟩ Cited in section 182. Used in section 186.
⟨ **Approximation** class declaration 184 ⟩ Used in section 182.
⟨ **Approximation** constructor code 189 ⟩ Used in section 186.
⟨ **Approximation** destructor code 190 ⟩ Used in section 186.
⟨ **DRFixPoint** :: *calcFixPoint* code 232 ⟩ Used in section 226.
⟨ **DRFixPoint** :: *fillTensors* code 229 ⟩ Used in section 226.
⟨ **DRFixPoint** :: *solveNewton* code 230 ⟩ Cited in section 226. Used in section 226.
⟨ **DRFixPoint** class declaration 226 ⟩ Used in section 208.
⟨ **DRFixPoint** constructor code 227 ⟩ Used in section 226.
⟨ **DRFixPoint** destructor code 228 ⟩ Used in section 226.
⟨ **DecisionRuleImpl** :: *centralizedClone* code 221 ⟩ Used in section 211.
⟨ **DecisionRuleImpl** :: *centralize* code 214 ⟩ Used in section 211.
⟨ **DecisionRuleImpl** :: *evaluate* code 220 ⟩ Used in section 211.
⟨ **DecisionRuleImpl** :: *eval* code 222 ⟩ Used in section 211.
⟨ **DecisionRuleImpl** :: *fillTensors* code 212 ⟩ Cited in section 213. Used in section 211.
⟨ **DecisionRuleImpl** :: *simulate* code 215 ⟩ Used in section 211.
⟨ **DecisionRuleImpl** :: *writeMat* code 223 ⟩ Used in section 211.
⟨ **DecisionRuleImpl** class declaration 211 ⟩ Used in section 208.
⟨ **DecisionRule** class declaration 210 ⟩ Used in section 208.
⟨ **DynamicModel** class declaration 175 ⟩ Used in section 173.
⟨ **ExplicitShockRealization** :: *addToShock* code 285 ⟩ Used in section 246.
⟨ **ExplicitShockRealization** :: *get* code 284 ⟩ Used in section 246.
⟨ **ExplicitShockRealization** class declaration 243 ⟩ Used in section 208.
⟨ **ExplicitShockRealization** constructor code 283 ⟩ Used in section 246.
⟨ **FaaDiBruno** :: *calculate* folded dense code 69 ⟩ Used in section 67.
⟨ **FaaDiBruno** :: *calculate* folded sparse code 68 ⟩ Cited in sections 65 and 70. Used in section 67.
⟨ **FaaDiBruno** :: *calculate* unfolded dense code 71 ⟩ Used in section 67.
⟨ **FaaDiBruno** :: *calculate* unfolded sparse code 70 ⟩ Used in section 67.
⟨ **FaaDiBruno** :: *estimRefinment* code 72 ⟩ Used in section 67.
⟨ **FaaDiBruno** class declaration 65 ⟩ Used in section 64.
⟨ **FirstOrder** :: *journalEigs* code 93 ⟩ Used in section 79.
⟨ **FirstOrder** :: *solve* code 81 ⟩ Used in section 79.
⟨ **FirstOrderDerivs** class declaration 77 ⟩ Used in section 75.
⟨ **FirstOrder** class declaration 76 ⟩ Cited in section 182. Used in section 75.
⟨ **FoldDecisionRule** class declaration 224 ⟩ Used in section 208.
⟨ **FoldDecisionRule** conversion from **UnfoldDecisionRule** 247 ⟩ Used in section 246.
⟨ **FoldedGXContainer** class declaration 156 ⟩ Used in section 141.
⟨ **FoldedZXContainer** class declaration 158 ⟩ Used in section 141.
⟨ **GResidFunction** class declaration 290 ⟩ Used in section 288.
⟨ **GXContainer** :: *getType* code 152 ⟩ Used in section 151.
⟨ **GXContainer** class declaration 151 ⟩ Used in section 141.
⟨ **GenShockRealization** :: *get* code 286 ⟩ Used in section 246.
⟨ **GenShockRealization** class declaration 244 ⟩ Used in section 208.
⟨ **GlobalChecker** :: *checkAlongShocksAndSave* code 309 ⟩ Used in section 294.
⟨ **GlobalChecker** :: *checkAlongSimulationAndSave* code 318 ⟩ Used in section 294.
⟨ **GlobalChecker** :: *checkOnEllipseAndSave* code 313 ⟩ Used in section 294.

⟨ **ZAuxContainer** constructor code 187 ⟩ Used in section 186.
⟨ **ZXContainer** :: *getType* code 154 ⟩ Used in section 153.
⟨ **ZXContainer** class declaration 153 ⟩ Used in section 141.
⟨ **ctraits** type traits declaration 96 ⟩ Used in section 95.
⟨ *endrec* code 26 ⟩ Used in section 13.
⟨ *order_eigs* function code 80 ⟩ Used in section 79.
⟨ *sysconf* Win32 implementation 28 ⟩ Used in section 13.

Dynare++