propag-4

BICGSTAB

Version 4.7 – June 10, 2009

1. Introduction. This is one of the documents describing the Montréal Heart Model. See the document entitled "propag" for a general introduction in its purpose and method of documentation.

This document describes a BiCGStab solver [vorst92,barrett:templates,saad96] that is adapted for use in `propag`. It is a bit longer than a BiCGStab solver proper would be, because there are several optional tests and variants implemented, such as the BiCGStab-P variant which is actually the default, restarts, an alternative expression for the β parameter, and two versions of QMRCGStab. There is even an option to continue the solution with a CG algorithm when BiCGStab has converged or failed.

The bi-conjugate gradient stabilized (BiCGStab) algorithm [vorst92] is a successor to the conjugate gradient squared (CGS) algorithm [sonne89] which is in turn a successor to the well-known bi-conjugate gradient (BCG) method [press92]. All these algorithms are generalizations for nonsymmetric matrices of the conjugate gradient (CG) method. The CG method is limited to symmetric matrices, but has the advantage of being more stable and converging more smoothly. But even for symmetric matrices, BiCGStab can be faster than CG. This is the case for our problem upto relative residuals of approximately 10^{-5}. For smaller residuals, CG gradually becomes better.

2. Further reading. Apart from the papers cited elsewhere in this document, the following may be of interest:

Freund et al. [freund92] give an overview just until the arrival of BiCGStab

Paige and Saunders [paige75] discuss MINRES and SYMMLQ

Paige and Saunders [paige82] propose the LSQR algorithm

Sleijpen and Fokkema [sleijpen93] discuss complex eigenvalues, not of our concern

Gatica and Heuer [gatica02] discuss some MINRES developments

Murphy et al. [murphy00] on preconditioning

Saad [saad89] on parallel implementation of Krylov methods

Van der Vorst [vorst89] on parallel implementation of preconditioners

Young et al [young89] on implementation of preconditioners

Numerical Recipes [press92] for the basics; including an explanation of CG

3. Interface. This function is somewhat biased to usage in `propag`; in fact

- The application must provide *atimes*( ) and *asolve*( ), see below. These functions hide all details about the structure of the problem matrix and the preconditioner. The only thing that the solver needs to know about the problem matrix is its number of elements.
- The application must implement *Warning*( ) according to the definition below
- The application must implement *logline*( )
- *bicgstab_dtype* may be set in bicgstab.h

We don't use `propag`'s *Error* function here; the solver function will always return normally.

⟨ function declarations 3 ⟩ ≡

```
extern void Warning(int code, char *fmt, ...);
```

See also section 7.

This code is used in section 6.

4. This section defines a parameter definition file for the `prm` program. It defines a group of parameters that the main program is supposed to make available as a global variable *bicgstab*, which is defined in `bicgstab_p.h`. The `prm` language is slightly adapted to make it usable in a `cweb` document; the C preprocessor does the translation using macros from `cweb_prm.h`. The `.prr` file must be explicitly preprocessed; gcc crashes if it reads this when called from `prm`.

```
⟨ bicgstab.prr 4 ⟩ ≡
  //␣$Id:␣bicgstab.web,v␣4.7␣2009/06/10␣18:51:57␣potse␣Exp␣$
#include "cweb_prm.h"      /* translates to prm code */
  structure Bicgstab
  {
    member(clamp_r0, type = int, default = 1, s_desc = "constrain␣r0")
    member(clamp_p, type = int, default = 1, s_desc = "constrain␣p")
    member(clamp_r, type = int, default = 1, s_desc = "constrain␣r")
    member(clamp_s, type = int, default = 1, s_desc = "constrain␣s")
    member(restart, type = int, default = 1, s_desc = "allow␣restarts")
    member(ttest, type = int, default = 0,
        s_desc = "report␣true␣residual␣once␣per␣ttest␣iters,␣0=never")
    member(variant, type = int, default = 0, s_desc = "black␣magic")
    member(verbose, type = int, default = 0, s_desc = "verbosity␣level")
    member(save_rr0, type = int, default = 0, s_desc = "save␣rr0␣to␣IGB␣file")
    member(save_rs, type = int, default = 0, s_desc = "save␣rs␣to␣IGB␣file")
    member(save_ss, type = int, default = 0, s_desc = "save␣ss␣to␣IGB␣file")
    member(use_bicgstab, type = int, default = 1, s_desc = "do␣BiCGStab␣iterations")
    member(use_cg, type = int, default = 0, s_desc = "do␣Conjugate␣Gradient␣iterations")
    member(use_point, type = int, default = 0, s_desc = "do␣point␣iterations")
    member(qmr, type = int, default = 0, s_desc = "QMRCGStab␣variant")
    member(varw, type = int, default = 0, s_desc = "variant␣omega␣(QMRCGStab-2)")
    member(varp, type = int, default = 1, s_desc = "BiCGStab-P␣rather␣than␣BiCGSTab")
    member(preco, type = int, default = 1, s_desc = "enable␣preconditioning")
    member(ftol, type = float, default = 1, s_desc = "tolerance␣factor␣for␣point␣iter")
  }
```

5. This header file may be included by the application. The *BiCGStab_solver* function solves a system $Ax = b$, where x and b are n-element vectors. The matrix A is not given as an argument; instead it is referenced through the functions *atimes* and *asolve*, which the application should provide. *atimes*( ) should compute $b = Ax$ and *asolve*( ) should perform a preconditioning step by solving x from the system $Px = b$ with P a matrix that is close to A but easy to solve. The *itol* flag determines how to interpret the *_tol* arguments, see below. Iteration proceeds until the error is lower than *final_tol*. If the initial error is lower than *start_tol*, no iterations are performed.

```
⟨ bicgstab.h 5 ⟩ ≡
  //␣$Id:␣bicgstab.web,v␣4.7␣2009/06/10␣18:51:57␣potse␣Exp␣$
      void BiCGSTAB_solver(long n, double b[], solver_t x[], int itol,
          double *final_tol, double *start_tol,
          int itmax, int *iter, double *err,
          void(*atimes)(long n, solver_t *x, solver_t *b),
          void(*asolve)(long n, solver_t *b, solver_t *yy, solver_t *x, int flags));
```

6. Implementation. The implementation consists of the *BiCGStab_solver* function and several private helper functions.

```
⟨ bicgstab.c 6 ⟩ ≡
  //␣$Id:␣bicgstab.web,v␣4.7␣2009/06/10␣18:51:57␣potse␣Exp␣$
  ⟨ Preprocessor definitions ⟩
#include <stdlib.h>     /* need this for ecc (Altix) */
#include <stdio.h>
#include <math.h>
#include "propag.h"
#include "bicgstab.h"
  ⟨ function declarations 3 ⟩
  ⟨ solver function 8 ⟩
  ⟨ helper functions 22 ⟩
```

7. Functions are declared static to ensure they are private.

```
⟨ function declarations 3 ⟩ +≡
  static solver_t norm_dt(long n, solver_t sx[], int itol);     /* private functions */
  static double norm_d(long n, double sx[], int itol);
  static double relative_error(solver_t norm, solver_t bnrm, float iter, int itol, double *threshold);
  static void copy_vector(long n, solver_t *target, solver_t *source);
  static long double inner_product(long n, solver_t *a, solver_t *b);
  static void constrain(long n, solver_t *v, int flag);
  static void true_norm(long n, solver_t *tnrm, solver_t *rnrm, solver_t *ip, solver_t *ax, double
      *b, solver_t *rs);
```

8. Solver function. This is a restarting implementation of BiCGStab. The inner loop terminates after each *restart* iterations. It also verifies at each iteration the norm of the updated residuals r and s, and breaks if it appears that convergence occurred. This causes a restart and a new calculation of the true residual. If the norm of the true residual is also small enough, the function returns.

```
⟨ solver function 8 ⟩ ≡
  void BiCGSTAB_solver(long n, double b[], solver_t x[], int itol,
          double final_tol[], double *start_tol,
          int itmax, int *iter, double *err,
          void(*atimes)(long n, solver_t *x, solver_t *b),
          void(*asolve)(long n, solver_t *b, solver_t *yy, solver_t *x, int flags))
  {
    long j, ii, restarts, quits, i;
    long double ak, akden, bk = 0.0, bkden, bknum, wk;      /* BiCGStab scalars */
    long double tet1, tet2, tau, eta1, eta2;      /* QMRCGStab scalars */
    solver_t rnrm = 0.0, snrm = 0.0, bnrm, inrm;
    static solver_t *rs = Λ, *p, *pp, *ss, *multpp, *multss, *rr0, *wb, *wa, *wc, *xx, *dd = Λ;
    if (rs ≡ Λ) ⟨ allocate and touch arrays 10 ⟩
    bnrm = norm_d(n, b, itol);      /* used to define tolerance */
    if (bicgstab.verbose) logline("BCS2", "start bnrm=%e", (double) bnrm);
    *iter = restarts = quits = 0;
    while (*iter ≤ itmax ∧ quits < 2 ∧ ¬stopflag) {
      ⟨ calculate residual rs and rr0 14 ⟩
      if (bicgstab.qmr) ⟨ initialize QMR variables 23 ⟩
      ⟨ check residual; return if small enough 15 ⟩
      if (bicgstab.use_bicgstab) {
        for (ii = 0; ii < itmax; ii++, (*iter)++) {
          if (*iter > itmax ∨ stopflag) return;
          ⟨ perform one BiCGStab iteration; break if convergence occurs 9 ⟩
        }
      }
      if (bicgstab.qmr) copy_vector(n, xx, x);      /* temporary */
      if (bicgstab.use_cg) {
        if (bicgstab.verbose) logline("BCS1", "switch to CG");
        for (ii = 0; ; ii++, (*iter)++) {
          if (*iter > itmax ∨ stopflag) return;
          ⟨ perform one CG iteration; break if convergence occurs 27 ⟩
        }
      }
      for (i = 0; i < 6; i++) final_tol[i] *= bicgstab.ftol;
      if (bicgstab.use_point) {
        if (bicgstab.verbose) logline("BCS1", "switch to point iterator");
        for (ii = 0; ; ii++, (*iter)++) {
          if (*iter > itmax ∨ stopflag) return;
          ⟨ perform a point iteration 28 ⟩
        }
      }
      if (bicgstab.verbose) logline("BCS1", "restart");
      restarts++;
    }
    if (quits > 1) Warning(20, "multiple restarts after convergence");
  }
```

This code is used in section 6.

9. The comments follow Barrett's notation [barrett:templates]. The organization of the loop was inspired by the BCG routine in Numerical Recipes [press92].

```
⟨ perform one BiCGStab iteration; break if convergence occurs 9 ⟩ ≡
  {
    if (¬bicgstab.variant ∨ ii ≡ 0) bknum = inner_product(n, rs, rr0);      /* ρ_{i−1} = r̃ · r_{i−1} */
    ⟨ check for bknum 16 ⟩
    if (ii ≡ 0) copy_vector(n, p, rs);      /* p^1 = r^0 */
    else {
      bk = (bknum/bkden) * (ak/wk);      /* β_{i−1} = (ρ_{i−1}/ρ_{i−2})(α_{i−1}/ω_{i−1}) */
#pragma omp parallel␣for␣schedule(static)␣private(j)
      for (j = 0; j < n; j++) p[j] = rs[j] + bk * (p[j] − wk * multpp[j]);
          /* p_i = r_{i−1} + β_{i−1}(p_{i−1} − ω_{i−1}v_{i−1}) */
    }
    bkden = bknum;
    constrain(n, p, bicgstab.clamp_p);
    precondition(p, Λ, pp);      /* solve Âp̂ = p_i */
    atimes(n, pp, multpp);      /* v_i = Âp̂ */
    ⟨ compute and check akden 17 ⟩;
    ak = bknum/akden;      /* α_i = ρ_{i−1}/(r̃ · v_i) */
#pragma omp parallel␣for␣schedule(static)␣private(j)
    for (j = 0; j < n; j++) {
      rs[j] −= ak * multpp[j];      /* s = r_{i−1} − α_i v_i */
      x[j] += ak * pp[j];      /* update x immediately, for monitoring */
    }
    constrain(n, rs, bicgstab.clamp_s);
    snrm = norm_dt(n, rs, itol);
    if (bicgstab.qmr) ⟨ first quasi-minimization and update 24 ⟩
    if (bicgstab.ttest ∧ (*iter % bicgstab.ttest ≡ 0)) ⟨ report on the true residual 21 ⟩
    ⟨ check for norm of s 19 ⟩      /* new: after constraining */
    precondition(rs, wb, ss);      /* solve ŝ from Âŝ = s */
    atimes(n, ss, multss);      /* t = Âŝ */
    if (bicgstab.save_rs) save_data_d(rs, "rs", SAVE_NODES, *iter, Λ);
    ⟨ compute wk = ω_i = (t · s)/(t · t) 18 ⟩
    if (bicgstab.variant) {      /* magic variant, Van der Vorst page 637 [vorst92] */
      bknum = −wk * inner_product(n, rr0, multss);      /* ρ_i = −ω_i(r̃, t) */
    }
#pragma omp parallel␣for␣schedule(static)␣private(j)
    for (j = 0; j < n; j++) {
      x[j] += wk * ss[j];      /* x_i = x_{i−1} + α_i p̂ (done above) + ω_i ŝ */
      rs[j] −= wk * multss[j];      /* r_i = s − ω_i t */
    }
    constrain(n, rs, bicgstab.clamp_r);
    rnrm = norm_dt(n, rs, itol);      /* norm of rs = r */
    if (bicgstab.qmr) ⟨ second quasi-minimization and update 25 ⟩
    if (bicgstab.ttest ∧ (*iter % bicgstab.ttest ≡ 0)) ⟨ report on the true residual 21 ⟩
    ⟨ check for norm of r 20 ⟩
    ⟨ check for wk 11 ⟩
  }
```

This code is used in section 8.

10. Touching *pp* and *ss* before *asolve*( ) touches them makes both solver and preconditioner about 10 % faster. Touching also *multss* and *multpp* gives an extra gain of about 1 % for the solver.

All arrays have an element −1. This allows the *asolve*( ) and *atimes*( ) routines to omit some range checks. It does no harm. To avoid NaNs and denormal floats, these elements are initialized.

```
⟨ allocate and touch arrays 10 ⟩ ≡
  {
    long n1 = n + 1;      /* allocate an extra element [−1] */
    logline("", "BiCGStab_solver␣$Revision:␣4.7␣$␣initializing...");
    rs = MALLOC(n1, solver_t, "BiCGSTAB:r/s") + 1;
    rr0 = MALLOC(n1, solver_t, "BiCGSTAB:rr0") + 1;
    p = MALLOC(n1, solver_t, "BiCGSTAB:p") + 1;
    pp = MALLOC(n1, solver_t, "BiCGSTAB:pp") + 1;
    ss = MALLOC(n1, solver_t, "BiCGSTAB:ss") + 1;
    multpp = MALLOC(n1, solver_t, "BiCGSTAB:multpp") + 1;
    multss = MALLOC(n1, solver_t, "BiCGSTAB:multss") + 1;
    if (bicgstab.varp) wa = wb = wc = Λ;      /* must be null for asolve( ) */
    else {
      wa = MALLOC(n1, solver_t, "BiCGSTAB:wa") + 1;
      wb = MALLOC(n1, solver_t, "BiCGSTAB:wb") + 1;
      wc = MALLOC(n1, solver_t, "BiCGSTAB:wc") + 1;
    }
    if (bicgstab.qmr) dd = MALLOC(n1, solver_t, "BICGSTAB::dd") + 1;
    if (bicgstab.qmr) xx = MALLOC(n1, solver_t, "BICGSTAB::xx") + 1;      /* temporary */
#pragma omp parallel␣for␣schedule(static)␣private(j)
    for (j = 0; j < n; j++) {
      pp[j] = ss[j] = 0;      /* touch before asolve */
      multpp[j] = multss[j] = 0;      /* touch before atimes */
    }
    rs[−1] = rr0[−1] = p[−1] = pp[−1] = ss[−1] = multpp[−1] = multss[−1] = 0.0;
  }
```

This code is used in section 8.

11. ⟨ check for *wk* 11 ⟩ ≡

```
  {
    if (wk ≡ 0) {
      Warning(23, "bicgstab␣stops␣because␣wk=0");      /* continuation condition */
      break;
    }
  }
```

This code is used in section 9.

12. A limit can be set on the number of iterations that are preconditioned, but it does not seem useful after all.

Also, switching off the preconditioner after a restart does not improve performance (exp208m102 vs exp208m101). The same goes for switching from ILU to a Jacobi preconditioner; in particular the irregular convergence (with occasional large divergences) is still there, perhaps worse than with ILU.

```
#define precondition(v, yy, vv)
        {
          bcg_protect_mem;
          if (bicgstab.preco) asolve(n, v, yy, vv, 0);      /* solve Av̂ = v_i */
          else copy_vector(n, vv, v);      /* v̂ = v */
          bcg_release_mem;
        }
```

13. Debug: this will cause a SEGV if the preconditioner touches our memory. The preconditioner is supposed to touch *ss* and *pp*. Even though the manual page suggests that write access can be granted without granting read access, experiments show that setting `PROT_WRITE` allows reading, while setting `PROT_NONE` does not allow it. Doesn't matter, we want to prevent illegal writes.

Memory protection can degrade performance. This is especially the case for the *pp* and *ss* arrays: protecting them makes the ILU preconditioner two times slower. So usually the program should run with the *protect_memory* parameter set to false in `propag.web`.

```
#define bcg_protect_mem
        {
          Protect(rs − 1, PROT_READ);
          Protect(rr0 − 1, PROT_READ);
          Protect(p − 1, PROT_READ);
          Protect(pp − 1, PROT_WRITE);
          Protect(ss − 1, PROT_WRITE);
          Protect(multpp − 1, PROT_READ);
          Protect(multss − 1, PROT_READ);
        }
#define bcg_release_mem
        {
          Protect(rs − 1, PROT_READ | PROT_WRITE);
          Protect(rr0 − 1, PROT_READ | PROT_WRITE);
          Protect(p − 1, PROT_READ | PROT_WRITE);
          Protect(pp − 1, PROT_READ);
          Protect(ss − 1, PROT_READ);
          Protect(multpp − 1, PROT_READ | PROT_WRITE);
          Protect(multss − 1, PROT_READ | PROT_WRITE);
        }
```

14. *rr0* is arbitrary except that $(r_0, rr0)$ must be ≤ 0, and there may be effects on convergence and breakdown. With very small blocks, $signum(rr)$ seems to work better than *rr* itself.

rs determines the Krylov subspace $rr0, Arr0, A^2rr0, ...$ [saad96]. For an element with zero initial residual the algorithm would never try to modify the solution. This may sound useful for the grounding node, but it also means that if the algorithm arrives at a solution that disagrees with zero potential at the grounding node, the answer vector would not reflect this. This means that the initial residual for the grounding node should be of considerable size.

Anyway, it's better not to work with a grounding node when the model size is very large [potse:bidofex].

```
⟨ calculate residual rs and rr0 14 ⟩ ≡
  {
    double small = 0.1 * final_tol[5]/sqrt(n);
    atimes(n, x, rs);      /* rs serves as temporary array */
#pragma omp parallel␣for␣schedule(static)␣private(j)
    for (j = 0; j < n; j++) {
      rs[j] = b[j] - rs[j];
#if 0     /* old attempt to ignore small fluctuations, but it bugged exp247 */
      if (fabs(rs[j]) < small) rs[j] = 0;
#endif
    }
    constrain(n, rs, bicgstab.clamp_r0);
    inrm = 0.0;
#pragma omp parallel␣for␣schedule(static)␣private(j)␣reduction(+:inrm)
    for (j = 0; j < n; j++) {
      rr0[j] = rs[j];      /* copy_vector(n, rr0, rs) */
      inrm += rs[j] * rs[j];
    }
    if (bicgstab.save_rr0) save_data_d(rr0, "rr0", SAVE_NODES, restarts, Λ);
    inrm = sqrt(inrm);      /* norm of the initial residual */
    rnrm = inrm;      /* used in test for bknum */
  }
```

This code is used in section 8.

15. This saves time in easy situations, and avoids a failure in case the initial residual is zero. Arguments to *printf* are typecast to correspond to their conversions, so we don't have to worry what **solver_t** is today.

In case restarts are used, this can also be used to monitor the true residual, since the initial residual is a true explicitly computed residual, while r and s are updated residuals and can deviate importantly.

FIXME: If *bnrm* is very small, inaccuracies in *atimes* may make it impossible for *rnrm* to reach the criterion. This will lead to an unending series of restarts. It should be possible to estimate these inaccuracies. Using $itol = 5$ or $itol = 6$ can prevent this problem.

```
⟨ check residual; return if small enough 15 ⟩ ≡
  {
    rnrm = norm_dt(n, rs, itol);
    if (isnan(rnrm)) {
      int i, cnt = 0;
      printf("\ntrouble: residual is not-a-number");
      for (i = 0; i < n ∧ cnt < 10; i++) {
        if (isnan(x[i])) printf("\nx[%d] = NaN", i), cnt++;
      }
      if (cnt ≡ 0) printf("\nno NaN found in x");
      exit(1);
    }
    *err = relative_error(rnrm, bnrm, *iter, itol, start_tol);
    if (bicgstab.verbose ≥ 2)
      logline("BCS2", "rr0 iter=%d  rnrm=%.2e  rerr=%5.3lf", *iter, (double) rnrm, (double) *err);
    if (*err < 1) {
      if (bicgstab.verbose) logline("BCS1", "residual %.2e is small enough", *err);
      return;      /* no further check needed */
    }
  }
```

This code is used in section 8.

16. Testing a new criterion for smallness of *bknum*. Note that *bknum* is the inner product of *rs* and *rr0*. This corresponds approximately with the interval of 20 iterations that worked quite well in exp213g7.

```
⟨ check for bknum 16 ⟩ ≡
  {
    long double crit, eps = 1 · 10^-18, d;
    static double rate, lasterr;
    if (bknum ≡ 0.) {
      Warning(23, "bicgstab fails: (r,rr0)=%Lf", (long double) bknum);
      break;
    }
    d = (inrm * rnrm * n);
    if (d ≡ 0) crit = 1.0;      /* paranoid */
    else crit = bknum/d;
    if (bicgstab.verbose ≥ 2) {
      if (*iter ≡ 1) rate = 0;
      else rate = *err/lasterr;
      lasterr = *err;
      logline("BCS", "crit=%+9.2e rnrm=%+9.2e inrm=%+9.2e iter=%-2d rerr=%-8.3f rate=%-.2f",
          (double) crit, (double) rnrm, (double) inrm, (int) *iter, (double) *err, rate);
    }
    if (fabsl(crit) < 1 · 10^3 * 2 * eps) {
      if (bicgstab.verbose) logline("BCS1", "bknum unreliable, calling for restart");
      if (bicgstab.restart) break;      /* a restart may help */
    }
  }
```

This code is used in section 9.

17. ⟨ compute and check *akden* 17 ⟩ ≡

```
{
  long double a, vnrm, d, crit, eps = 1.0 · 10^-18;
#if 0
  akden = inner_product(n, multpp, rr0);
#else
  akden = vnrm = 0;
#pragma omp parallel␣for␣schedule(static)␣private(a)␣reduction(+:vnrm,akden)
  for (j = 0; j < n; j++) {
    a = multpp[j];
    vnrm += a * a;
    akden += a * rr0[j];
  }
  vnrm = sqrt(vnrm);
  d = inrm * vnrm * n;
  if (d ≡ 0) crit = 1.0;
  else crit = akden/d;
  if (bicgstab.verbose ≥ 2) logline("BCS2", "akden=%.2Le␣crit=%.2Le", akden, crit);
  if (fabsl(crit) < 1 · 10^3 * 2 * eps) {
    if (bicgstab.verbose) logline("BCS1", "akden␣unreliable,␣calling␣for␣restart");
    if (bicgstab.restart) break;
  }
#endif
}
```

This code is used in section 9.

18. With ω_i computed this way, this is actually a BiCGStab-P variant [vorst92], but this is what everybody calls Preconditioned BiCGStab [saad96,barrett:templates]. Setting the *varp* parameter to zero makes it an original preconditioned BiCGStab, which takes three extra arrays and an extra left preconditioner step. There's also a variant, enabled with the *varw* parameter, that turns QMRCGStab into QMRCGStab-2. The same smallness criterion as for *bknum* is used for the inner product (t, s).

```
⟨ compute wk = ω_i = (t · s)/(t · t) 18 ⟩ ≡
  {
    long double wkden = 0.0, wknum = 0.0, crit, tj, sj, eps = 1 · 10^-18;
    if (¬bicgstab.varp) precondition(multss, wa, wc);      /* flush result, actually only left preco needed */
    if (bicgstab.varp) {      /* the default */
#pragma omp parallel␣for␣schedule(static)␣private(j,tj,sj)␣reduction(+:wknum,wkden)
      for (j = 0; j < n; j++) {
        tj = multss[j];      /* t itself */
        sj = rs[j];      /* s itself */
        wknum += tj * sj;
        wkden += tj * tj;
      }
    }
    else {
#pragma omp parallel␣for␣schedule(static)␣private(j,tj,sj)␣reduction(+:wknum,wkden)
      for (j = 0; j < n; j++) {
        tj = wa[j];      /* left-preconditioned t */
        sj = wb[j];      /* left-preconditioned s */
        wknum += tj * sj;
        wkden += tj * tj;
      }
    }
    if (bicgstab.varw) wk = snrm * snrm/wknum;      /* ω_i, QMRCGStab-2 variant */
    else wk = wknum/wkden;      /* ω_i */
    crit = wknum/(snrm * sqrt(wkden) * n);
    if (fabsl(crit) < 1 · 10^3 * 4 * eps) {
      if (bicgstab.verbose) logline("BCS1","wknum␣unreliable,␣calling␣for␣restart");
      if (bicgstab.restart) break;      /* a restart may help */
    }
  }
```

This code is used in section 9.

19. If the norm of s is small enough, do a half update of x before returning. In the current implementation this half-update was already done, to allow monitoring of the true residual.

```
⟨ check for norm of s 19 ⟩ ≡
  {
    *err = relative_error(snrm, bnrm, *iter − 0.5, itol, final_tol);
    if (bicgstab.verbose ≥ 2)
      logline("BCS2","step1␣iter=%d␣snorm=%.2e␣rerr=%5.3lf␣ak=%.2Le␣bk=%.2Le␣bknum=%.2Le",
          (int) *iter,(double) snrm,(double) *err,(long double) ak,(long double) bk,(long
          double) bknum);
    if (*err < 1.0) {
      if (bicgstab.verbose) logline("BCS1","Norm␣of␣s␣small␣enough");
      quits++;
      break;
    }
  }
```

This code is used in section 9.

20. When we arrive here, x was already updated.

```
⟨ check for norm of r 20 ⟩ ≡
  {
    *err = relative_error(rnrm, bnrm, *iter, itol, final_tol);
    if (bicgstab.verbose ≥ 2) logline("BCS2", "step2␣iter=%d␣rnorm=%.2e␣␣rerr=%5.3f␣wk=%.2Le",
          *iter, (double) rnrm, (double) *err, (long double) wk);
    if (*err < 1.0) {
      if (bicgstab.verbose) logline("BCS1", "Norm␣of␣r␣small␣enough");
      quits++;
      break;
    }
  }
```

This code is used in sections 9, 27, and 28.

21. The explicit or "true" residual can be larger than the updated residuals r and s [vorst92]. In large problems it can also be smaller. This section just reports what is going on.

```
⟨ report on the true residual 21 ⟩ ≡
  {
    solver_t tnrm, rsnrm, ip;
    static solver_t *ax = Λ;      /* storage for Ax */
    if (ax ≡ Λ) ax = MALLOC(n + 1, solver_t, "BiCGStab:ax") + 1;
    atimes(n, x, ax);      /* BiCGStab version */
    true_norm(n, &tnrm, &rsnrm, &ip, ax, b, rs);
    logline("BCS", "ttest␣rsnrm=%.2e␣␣tnrm=%.2e␣␣nip=%.3f", (double) rsnrm, (double)
        tnrm, (double) ip);
    if (bicgstab.qmr) {
      atimes(n, xx, ax);      /* QMRCGStab version */
      true_norm(n, &tnrm, &rsnrm, &ip, ax, b, rs);
      logline("BCS", "qtest␣rsnrm=%.2e␣␣tnrm=%.2e␣␣nip=%.3f", (double) rsnrm, (double)
          tnrm, (double) ip);
    }
  }
```

This code is used in sections 9 and 27.

22. ⟨ helper functions 22 ⟩ ≡

```
  static void true_norm(long n, solver_t *tnrm, solver_t *rnrm, solver_t *ip, solver_t *ax, double
        *b, solver_t *rs)
  {
    long j;
    solver_t rj, tsum = 0.0, rsum = 0.0, psum = 0.0;
#pragma omp parallel␣for␣schedule(static)␣private(j,rj)␣reduction(+:tsum,rsum,psum)
    for (j = 0; j < n; j++) {
      rj = b[j] − ax[j];
      tsum += rj * rj;      /* true norm */
      rsum += rs[j] * rs[j];      /* current residual norm */
      psum += rj * rs[j];      /* inner product */
    }
    *tnrm = sqrt((double) tsum);
    *rnrm = sqrt((double) rsum);
    *ip = psum/(*tnrm * *rnrm);
  }
```

See also sections 29, 30, 31, 32, 33, and 34.

This code is used in section 6.

23. QMR Variant. Testing the QMRCGStab variant [chan94], adapted for preconditioning. With some shuffling a more efficient implementation of QMRCGStab-2 is possible [xpliu05], but this is mostly of interest for massively parallel machines.

```
⟨ initialize QMR variables 23 ⟩ ≡
  {
    tau = inrm;      /* ||r^0|| */
    tet1 = tet2 = 0.0;
    eta1 = eta2 = 0.0;
#pragma omp parallel␣for␣schedule(static)␣private(j)
    for (j = 0; j < n; j++) {
      dd[j] = 0;
      xx[j] = x[j];      /* temporarily keep a second x array */
    }
  }
```

This code is used in section 8.

24. ⟨ first quasi-minimization and update 24 ⟩ ≡

```
  {
    long double f, c;
    tet2 = snrm/tau;      /* θ̃ = ||s_i||/τ */
    c = 1.0/sqrt(1 + tet2 * tet2);      /* c = 1/√(1+θ̃_i^2) */
    tau = tau * tet2 * c;      /* τ̃ = τ θ̃_i c */
    eta2 = c * c * ak;      /* η̃ = c^2 α_i */
    f = tet1 * tet1 * eta1 / ak;      /* θ^2_{i-1} η_{i-1} / α_i */
#pragma omp parallel␣for␣schedule(static)␣private(j)
    for (j = 0; j < n; j++) {
#if 0
      dd[j] = p[j] + f * dd[j];      /* d̃_i = p + f d_{i-1} */
#else
      dd[j] = pp[j] + f * dd[j];      /* d̃_i = p̂ + f d_{i-1} for preconditioned */
#endif
      xx[j] += eta2 * dd[j];      /* x̃_i = x_{i-1} + η̃_i d̃_i */
    }
  }
```

This code is used in section 9.

25. ⟨ second quasi-minimization and update 25 ⟩ ≡

```
  {
    long double f, c;
    tet1 = rnrm/tau;      /* θ_i = ||r_i||/τ̃ */
    c = 1.0/sqrt(1 + tet1 * tet1);      /* c = 1/√(1+θ_i^2) */
    tau = tau * tet1 * c;      /* τ = τ̃ θ_i */
    eta1 = c * c * wk;      /* η = c^2 ω_i */
    f = tet2 * tet2 * eta2 / wk;      /* θ̃^2_i η̃_i / ω_i */
#pragma omp parallel␣for␣schedule(static)␣private(j)
    for (j = 0; j < n; j++) {
#if 0
      dd[j] = rs[j] + f * dd[j];      /* d_i = s + f d̃_i */
#else
      dd[j] = ss[j] + f * dd[j];      /* d_i = ŝ + f d̃_i for preconditioned */
#endif
      xx[j] += eta1 * dd[j];      /* x_i = x̃_i + η_i d_i */
    }
  }
```

This code is used in section 9.

26. FIXME: Instead of QMRCGStab we may try Minimal-Residual Smoothing as described by Zhou and Walker [zhou94].

Residual smoothing was implemented as an attempt to remove the oscillations that occur in the signals especially in monodomain simulations. But although it leads to a smoothly converging solution it does not reduce the oscillations at all.

Other methods to reduce these oscillations are to compute ϕ_e more often, as in a bidomain simulation, or with a lower tolerance, which reduces the amplitude of the oscillations. These methods actually worked, at least in experiments with a small heart.

Furthermore, any measure that extends the fast and smooth convergence regime would be very helpful. In case of monodomain simulations, an interesting alternative would be to compute ϕ_e on a much coarser grid. This would of course be much faster and merits a study by itself.

27. Conjugate Gradient iterations. NOTE: The matrix must be symmetric, so the *nground* parameter in `bidofex` must be zero if using the constraint method, otherwise CG does not converge!

The CG routine must be completely initialized when $ii = 0$, using the p arry from the last iteration led to a divergence to infinity in an experiment in a small block.

```
⟨ perform one CG iteration; break if convergence occurs 27 ⟩ ≡
  {
    if (ii ≡ 0) precondition(rs, Λ, ss);      /* solve z^0 from Az^0 = r^0 */
    if (ii ≡ 0) copy_vector(n, p, ss);      /* p^0 = z^0, necessary at ii ≡ 0! */
    bknum = inner_product(n, rs, ss);      /* r · z */
    if (ii > 0) {
      bk = bknum/bkden;
#pragma omp parallel␣for␣schedule(static)␣private(j)
      for (j = 0; j < n; j++) p[j] = bk * p[j] + ss[j];      /*p ← βp + z*/
    }
    constrain(n, p, bicgstab.clamp_p);
    bkden = bknum;
    atimes(n, p, multpp);      /* v_i = Ap̂ */
    akden = inner_product(n, multpp, p);      /* v · p */
    ak = bknum/akden;      /* α_i = (r · z)/(pAp) */
    if (bicgstab.verbose) logline("BCS1","CG␣aknum=%.2e␣␣akden=%.2e␣␣ak=%.2e",(double)
          bknum,(double) akden,(double) ak);
#pragma omp parallel␣for␣schedule(static)␣private(j)
    for (j = 0; j < n; j++) {
      rs[j] -= ak * multpp[j];      /* r_i = r_{i−1} − α_i v_i */
      x[j] += ak * p[j];      /* x_i = x_{i−1} + αp */
    }
    constrain(n, rs, bicgstab.clamp_r);
    precondition(rs, Λ, ss);      /* solve z_i from Az_i = r_i */
    if (bicgstab.ttest ∧ (∗iter % bicgstab.ttest ≡ 0)) ⟨ report on the true residual 21 ⟩
    rnrm = norm_dt(n, rs, itol);      /* norm of rs = r */
    ⟨ check for norm of r 20 ⟩
  }
```

This code is used in section 8.

28. Point iterations. This is even simpler, actually the simplest possible iterative method:

$$x_i - x_{i-1} = \tilde{A}^{-1}(b - Ax_{i-1})$$

where $\tilde{A}$ is an approximation to A. Well-known such methods are the Jacobi iteration, which approximates A by its diagonal only, and the Gauss-Seidel iteration, but any approximation to A can be used [meij77]. We use the preconditioner to provide the approximate inverse. It is expected that point iterations converge terribly slow compared to BiCGStab, but perhaps more stable when the error is very small. For a small block and normal tolerance settings the number of iterations was 8 times larger with the point iterator than with BiCGStab. This ratio is the same when the tolerance is reduced by a factor 100. But in that case, using point iterations for the last factor 10 (ussing the *bicgstab.ftol* parameter) increases the total number of iterations by only a factor 2. A test on the large model, however, demonstrated an increase by more than a factor 10 for the same trick. Note that the optimal MILU parameter (`ddpreco.web`) may be different for a point-ILU algorithm than for an ILU-preconditioned BiCGStab.

```
⟨ perform a point iteration 28 ⟩ ≡
  {
    atimes(n, x, rs);      /* rs serves as temporary array */
#pragma omp parallel␣for␣schedule(static)␣private(j)
    for (j = 0; j < n; j++) {
      rs[j] = b[j] − rs[j];      /* r = b − Ax */
    }
    constrain(n, rs, bicgstab.clamp_r0);
    rnrm = norm_dt(n, rs, itol);      /* norm of rs = r */
    ⟨ check for norm of r 20 ⟩
    precondition(rs, Λ, ss);
#pragma omp parallel␣for␣schedule(static)␣private(j)
    for (j = 0; j < n; j++) {
      x[j] += ss[j];      /* x_i = x_{i−1} + Ã^{−1} r */
    }
  }
```

This code is used in section 8.

29. Constraints. The first one orthogonalizes with respect to common mode; the second on the grounded cell (hardcoded!). The first works much better (exp208m51, 53 vs exp208m54). If *flag* $\equiv -1$, *mu* is only computed and reported; if *flag* $\equiv 1$, the constraint is applied.

```
#define GCELL 70512
⟨ helper functions 22 ⟩ +≡
  void constrain(long n, solver_t *v, int flag)
  {
#if 1
    long double mu;
    long j;
    if (flag ≠ 0) {
      mu = 0.0;
#pragma omp parallel␣for␣schedule(static)␣private(j)␣reduction(+:mu)
      for (j = 0; j < n; j++) mu += v[j];
      mu /= n;
      if (bicgstab.verbose) logline("BCS2","constrain:␣mu=%e,␣flag=%d",(double) mu,flag);
      if (flag ≡ 1) {
#pragma omp parallel␣for␣schedule(static)␣private(j)
        for (j = 0; j < n; j++) v[j] = (long double) v[j] − mu;
      }
    }
#else
    if (flag ≡ 1) v[GCELL] = 0.0;
#endif
  }
```

30. Helper functions. A macro rather than a function would be nice to make it work regardless of type, but a pragma inside a macro does not work. All vectors have the same length and all operations are supposed to be parallellized in the same way.

```
⟨ helper functions 22 ⟩ +≡
  void copy_vector(long n, solver_t *target, solver_t *source)
  {
    long j;
#pragma omp parallel␣for␣schedule(static)␣private(j)
    for (j = 0; j < n; j++) target[j] = source[j];
  }
```

31.
```
⟨ helper functions 22 ⟩ +≡
  long double inner_product(long n, solver_t *a, solver_t *b)
  {
    long j;
    long double p;
    p = 0.0;
#pragma omp parallel␣for␣schedule(static)␣private(j)␣reduction(+:p)
    for (j = 0; j < n; j++) p += a[j] * b[j];
    return p;
  }
```

32.
```
⟨ helper functions 22 ⟩ +≡
  solver_t norm_dt(long n, solver_t sx[], int itol)      /* compute norm of sx */
  {
    long i, isamax;
    double ans;
    if (itol ≤ 3 ∨ itol ≡ 5 ∨ itol ≡ 6) {
      ans = 0.0;
#pragma omp parallel␣for␣schedule(static)␣private(i)␣reduction(+:ans)
      for (i = 0; i < n; i++) ans += sx[i] * sx[i];
      return (solver_t) sqrt(ans);
    }
    else {
      isamax = 1;
      for (i = 0; i < n; i++) {
        if (fabs(sx[i]) > fabs(sx[isamax])) isamax = i;
      }
      return fabs(sx[isamax]);
    }
  }
```

33. ⟨ helper functions 22 ⟩ +≡

```
double norm_d(long n, double sx[], int itol)      /* compute norm of sx */
{
  long i, isamax;
  double ans;
  if (itol ≤ 3 ∨ itol ≡ 5 ∨ itol ≡ 6) {
    ans = 0.0;
#pragma omp parallel␣for␣schedule(static)␣private(i)␣reduction(+:ans)
    for (i = 0; i < n; i++) ans += sx[i] * sx[i];
    return sqrt(ans);
  }
  else {
    isamax = 1;
    for (i = 0; i < n; i++) {
      if (fabs(sx[i]) > fabs(sx[isamax])) isamax = i;
    }
    return fabs(sx[isamax]);
  }
}
```

34. The *iter* argument is float to allow representation of half-iterations.

⟨ helper functions 22 ⟩ +≡

```
double relative_error(solver_t norm, solver_t bnrm, float iter, int itol, double *threshold)
{
  double r1, r5;
  if (bnrm ≡ 0.0) {
    if (norm ≠ 0) Warning(25, "bnorm==0␣and␣norm=%.2e,␣cannot␣compute␣r1!", (double) norm);
    r1 = 0;
  }
  else {
    r1 = norm/(bnrm * threshold[1]);
  }
  r5 = norm/threshold[5];
  if (bicgstab.verbose)
    logline("BCS1", "rerr␣iter=%.1f␣norm=%.2e␣bnorm=%.2e␣itol=%d␣␣r1=%.2e␣␣r5=%.2e", iter,
        (double) norm, (double) bnrm, itol, (double) r1, (double) r5);
  switch (itol) {
  case 1: return r1; break;
  case 5: return r5; break;
  case 6:
    if (r1 < r5) return r1;
    else return r5;
    break;
  default: printf("\ninvalid␣itol␣=␣%d\n", itol);
    exit(1);
  }
}
```

35. Bibliography.

[barrett:templates] Richard Barrett, Michael Berry, Tony F. Chan, James Demmel, June M. Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, Philadelphia, 2nd edition, 1994. `http://www.netlib.org/templates/templates.ps`.

[chan94] T. F. Chan, E. Gallopoulos, V. Simoncini, T. Szeto, and C. H. Tong. A quasi-minimal residual variant of the Bi-CGStab algorithm for nonsymmetric systems. *SIAM J. Sci. Comput.*, 15(2):338–347, 1994.

[freund92] R. Freund, G. Golub, and N. Nachtigal. Iterative solution of linear systems. In A. Iserles, editor, *Acta Numerica 1992*, pages 57–100. Cambridge University Press, 1992.

[gatica02] Gabriel N. Gatica and Norbert Heuer. A preconditioned MINRES method for the coupling of mixed-FEM and BEM for some nonlinear problems. *SIAM J. Sci. Comput.*, 24(2):572–596, 2002.

[xpliu05] XingPing Liu, TongXiang Gu, ZhiQiang Sheng, and XuDeng Hang. Improved QMRCGSTAB method in distributed parallel environments. *Int. J. Computer Math.*, 82(9):1141–1148, 2005.

[meij77] J. A. Meijerink and H. A. van der Vorst. An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix. *Math. Comput.*, 31(137):148–162, 1977.

[murphy00] Malcolm F. Murphy, Gene H. Golub, and Andrew J. Wathen. A note on preconditioning for indefinite linear systems. *SIAM Journal on Scientific Computing*, 21(6):1969–1972, 2000.

[paige75] C. C. Paige and M. A. Saunders. Solution of sparse indefinite systems of linear equations. *SIAM J. Numer. Anal.*, 12(4):617–629, 1975.

[paige82] Christopher C. Paige and Michael A. Saunders. LSQR: An algorithm for sparse linear equations and sparse least squares. *ACM Trans. Math. Soft.*, 8(1):43–71, 1982.

[potse:bidofex] Mark Potse, Bruno Dubé, Jacques Richer, Alain Vinet, and Ramesh M. Gulrajani. A comparison of monodomain and bidomain reaction-diffusion models for action potential propagation in the human heart. *IEEE Trans. Biomed. Eng.*, 53(12):2425–2435, 2006.

[press92] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C; The Art of Scientific Computing*. Cambridge University Press, Cambridge, UK, second edition, 1992.

[saad89] Youcef Saad. Krylov subspace methods on supercomputers. *SIAM J. Sci. Stat. Comput.*, 10:1200–1232, 1989.

[saad96] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. PWS publishing, New York, 1996. Second edition published by SIAM, Philadelphia, 2003.

[sleijpen93] Gerard L. G. Sleijpen and Diederik R. Fokkema. BICGSTAB(L) for linear equations involving unsymmetric matrices with complex spectrum. *Elec. Trans. Numer. Anal.*, 1:11–32, 1993.

[sonne89] Peter Sonnenveld. CGS, a fast Lanczos-type solver for nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 10:35–52, 1989.

[vorst92] H. A. van der Vorst. Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 13:631–644, 1992.

[vorst89] Henk A. van der Vorst. High performance preconditioning. *SIAM J. Sci. Stat. Comput.*, 10:1174–1185, 1989.

[young89] David P. Young, Robin G. Melvin, Forrester T. Johnson, John E. Bussolett, Laurence B. Wigton, and Satish S. Samant. Application of sparse matrix solvers as effective preconditioners. *SIAM J. Sci. Stat. Comput.*, 10:1186–1199, 1989.

[zhou94] Lu Zhou and Homer F. Walker. Residual smoothing techniques for iterative methods. *SIAM J. Sci. Comput.*, 15(2):297–312, 1994.

36. Index.