NUMERICAL RECIPES

Webnote No. 12, Rev. 2

Routine Implementing the Simplex Method

Our routine implementing the simplex method is based on the algorithm given in [1]. The constraint matrix $\mathbf{A}$ is input as an $m \times n$ sparse matrix `sa` of type `NRsparseMat` (§2.7). Only the coefficients in the equations like (10.10.3) and (10.10.4) are input; the unit vectors corresponding to the m slack and artificial variables are generated internally. In addition to nonnegative and zero variables, the routine can also handle *free* variables. These are variables that are unrestricted, taking on any value in $(-\infty, \infty)$. A free variable should always be a basic variable in the optimal solution, since you can improve the value of the objective function by exchanging some restricted variable for an unrestricted one. The routine moves all the free variables into the basis as part of a phase zero. The type of the variables is input in the array `initvars[1..n+m]`. A value of 0 signifies a zero variable (typically an artificial variable for an equality constraint, but zero variables among the original independent variables are also allowed); a value of 1 denotes a nonnegative variable (independent variable or slack variable for an inequality); and -1 is used for free variables. The coefficients of the objective function are input in `obj[0..n+m]`. A constant c_0 can be included in (10.10.16), and this is stored in `obj[0]`. In most problems, `obj[n+1..n+m]` will be zero. The solution is output in `u[0..n+m]`: `u[0]` contains the optimal value of the objective function, `u[1..n]` the values of $x_1, \ldots, x_n$ at the minimum, and `u[n+1..n+m]` the values of the slack variables. Intermediate results are printed when the variable `verbose` is set to `true`, which can be helpful for diagnosing problems. Set it to `false` to suppress this printing.

For efficient access, the routine initially copies the constraint matrix `sa` into an array `a` of its n columns. Again, only the nonzeros are stored, together with the corresponding row numbers. Every `NREFAC` steps the LU decomposition of the basis is carried out from scratch. This is because the Bartels-Golub updates are actually stored as a sequence of transformations to the original LU decomposition. Applying the accumulated updates becomes expensive after too many steps, and also the accuracy can deteriorate. The default is `NREFAC` $= 50$, which is satisfactory for most problems.

Note that the Bartels-Golub update requires not the incoming column $\mathbf{a}_k$, but $\mathbf{L}^{-1} \cdot \mathbf{a}_k$. Accordingly, the routine computes the quantity $\mathbf{w} = \mathbf{A}_B^{-1} \cdot \mathbf{a}_k = \mathbf{U}^{-1} \cdot \mathbf{L}^{-1} \cdot \mathbf{a}_k$ of equation (10.10.25) in two steps: first $\mathbf{L}^{-1} \cdot \mathbf{a}_k$ is computed and stored for the update, and then $\mathbf{w}$ is computed by multiplying by $\mathbf{U}^{-1}$.

The routine incorporates implicit scaling. Column scale factors `scale[1..n]` and row scale factors `scale[n+1..n+m]` are first determined so that the maximum norm is one for each row and each column of the matrix

$$\bar{a}_{ik} = a_{ik}\frac{\texttt{scale}[i]}{\texttt{scale}[n+i]} \qquad i = 1,\ldots,m \qquad k = 1,\ldots,n \tag{1}$$

Then the pivot selection rules are modified so that applying the modified rules to the unscaled matrix gives the same results as the unmodified rules to the scaled matrix. Most of the time, performance is improved with scaling. However, since no perfect scaling algorithm is known, occasionally no scaling works better. To turn off scaling, just set `scale[1..n+m]` $= 1$ in function `scaleit`.

For coding ease, the routine implements the minimum ratio test (10.10.25) by computing the inverse of the step length, i.e., the quantity $(\mathbf{A}_B^{-1}\cdot\mathbf{a}_k)^i/x_B^i$. The various tests for zero quantities in the pivot selection and in other parts of the code are replaced by tests with appropriate tolerances that depend on the machine precision.

You invoke the routine with a fragment like the following:

```
Simplex s(m,n,initvars,b,obj,A,false);
s.solve();
```

Setting the boolean argument to `true` prints intermediate diagnostics. The constraint matrix `A` is of type `NRsparseMat`. The output can be accessed with a fragment like

```
switch (s.ierr) {
    case 0:
        for (j=1;j<=n;j++)
            cout << j << " " << s.u[j] << endl;
        cout << "iterations " << s.nsteps << endl;
        cout << "minimum " << s.u[0] << endl;
        break;
    case 1: cout << "phase 0 failed" << endl;
        break;
    case 2: cout << "no feasible solution (phase 1)" << endl;
        break
     ...
```

(The rest of the error returns are listed in the function header comment.)

The routine in `simplex` is reasonably robust: it solves all the Netlib problems [2] that do not involve explicit bounds on the variables other than the usual nonnegativity constraints. Typical "medium-size" problems involving $\lesssim 10^4$ constraints and variables are solved in a few seconds on a 2006 workstation. Very degenerate or very dense problems take longer, of course. However, the routine is nowhere close to state-of-the-art, either in performance or in features. The main text discusses alternatives that may be better for your application.

Note that simplex codes almost universally number rows and columns starting at 1, not 0. Often the coefficients of the objective function are stored in row 0 (although we do not do so).

simplex.h

```
struct Simplex {
```

Minimizes an objective function by the revised simplex method for linear programming.

```
    Int m,n,ierr;
    VecInt initvars,ord,ad;
```

If `ord[i]=k`, `i` $=$ `1...m`, the `i`th column in the basis is column `k` of the original matrix, `k` $=$ `1...n+m`. However, if `k` corresponds to a free variable, we make `ord` < 0 by subtracting `n+m+1`. `ad[k]`, `k` $=$ `1...n+m` is -1 if column `k` is in the basis, 1 if not, and 0 if it corresponds to a zero variable. Thus only columns with `ad` > 0 are eligible to enter the basis, only those with `ord` > 0 are eligible to leave.

```
    VecDoub b,obj,u,x,xb,v,w,scale;
```

`xb` contains the current basic solution $\mathbf{A}_B^{-1}\cdot\mathbf{b}$.

```
    NRsparseMat &sa;
    Int NMAXFAC,NREFAC;
```

```
    Doub EPS,EPSSMALL,EPSARTIF,EPSFEAS,EPSOPT,EPSINFEAS,EPSROW1,EPSROW2,EPSROW3;
```
Parameters: NMAXFAC specifies the maximum number of iterations as NMAXFAC*max(m,n); NREFAC is the number of iterations between refactorizations. The various EPS tolerances are individually adjustable. In particular, EPSSMALL governs the size of terms that are assumed to be zero in the sparse matrix procedures. Difficult problems may require it to be either increased or decreased.
```
    Int nm1,nmax,nsteps;
    Bool verbose;
    NRvector<NRsparseCol *> a;
```
Input matrix sa is stored as a vector of sparse columns in a.
```
    NRlusol *lu;
```
Interface to LUSOL routines.

```
    Simplex(Int mm,Int nn,VecInt_I &initv,VecDoub_I &bb,VecDoub_I &objj,
        NRsparseMat &ssa,Bool verb);
```
The constructor initializes the problem with the following variables: m is the number of constraints (rows) and n is the number of variables (columns). initvars[1..n+m] is 0 for a zero variable, -1 for a free variable, and 1 for a nonnegative variable. In particular, initvars[n+1..n+m] is 0 for an equality constraint and 1 for an inequality, while initvars[1..n] is generally 1 unless there are free or zero variables. The right-hand side is input as b[1..m], while the coefficients of the objective function are in obj[0..n+m]. The coefficients of the constraint matrix are input in the sparse matrix sa. Diagnostic output is generated if verbose is input as true.
```
    void solve();
```
After solve() has been called, u[0] is the value of the objective function, u[1..n] contains the values of the variables at the minimum, while u[n+1..n+m] gives the values of the slack variables (i.e., the amount by which the corresponding inequality is satisfied). The error returns are:

ierr=1: phase 0 failed
ierr=2: no feasible solution (phase 1)
ierr=3: steps exceeded in phase 1
ierr=4: steps exceeded in phase 2
ierr=5: unbounded objective function
ierr=6: rowpiv failed in phase 1

```
    void initialize();
    void scaleit();
    void phase0();
    void phase1();
    void phase2();
    Int colpiv(VecDoub &v,Int phase,Doub &piv);
    Int rowpiv(VecDoub_I &w,Int phase,Int kp,Doub xbmax);
    void transform(VecDoub &x,Int ip,Int kp);
    Doub maxnorm(VecDoub_I &xb);
    VecDoub getcol(Int k);
    VecDoub lx(Int kp);
    Doub xdotcol(VecDoub_I &x, Int k);
    void refactorize();
    void prepare_output();
};

Simplex::Simplex(Int mm,Int nn,VecInt_I &initv,VecDoub_I &bb,VecDoub_I &objj,
        NRsparseMat &ssa,Bool verb) :
    m(mm),n(nn),initvars(initv),b(bb),obj(objj),sa(ssa),verbose(verb),ord(m+1),
    ad(m+n+1),u(m+n+1),x(m+1),xb(m+1),v(m+1),w(m+1),scale(n+m+1),a(n+1) {
```
Constructor accepts input data, allocates vectors, and sets parameters.
```
    NMAXFAC=40;
    NREFAC=50;
    EPS=numeric_limits<Doub>::epsilon();
    EPSSMALL=1.0e5*EPS;
```
Determines negligible elements in LU (see refactorize).
```
    EPSARTIF=1.0e5*EPS;
    EPSFEAS=1.0e8*EPS;
    EPSOPT=1.0e8*EPS;
    EPSINFEAS=1.0e4*EPS;
    EPSROW1=1.0e-5;
    EPSROW2=EPS;
```

```
    EPSROW3=EPS;
}

void Simplex::solve()
```
Solves problem after constructor has been called.
```
{
    initialize();
    scaleit();
    phase0();
    if (verbose)
        cout << "    at end of phase0,iter= " << nsteps << endl;
    if (ierr != 0) {
        delete lu;
        for (Int i=0;i<n;i++)
            delete a[i+1];
        return;
    }
    phase1();
    if (verbose)
        cout << "    at end of phase1,iter= " << nsteps << endl;
    if (ierr != 0) {
        delete lu;
        for (Int i=0;i<n;i++)
            delete a[i+1];
        return;
    }
    phase2();
    prepare_output();
    delete lu;
    for (Int i=0;i<n;i++)
        delete a[i+1];
}

void Simplex::initialize() {
    VecInt irow(2);
```
Used to input unit vector to basis. Element 0 is ignored.
```
    VecDoub value(2);
    nsteps=0;
```
Number of iterations.
```
    ierr=0;
    nm1=n+m+1;
    nmax=NMAXFAC*MAX(m,n);
    for (Int i=1;i<=n;i++)
        if (initvars[i] == 0)
```
A zero variable is never eligible to enter the basis.
```
            ad[i]=0;
        else
            ad[i]=1;
    for (Int i=n+1;i<=n+m;i++)
        ad[i]=-1;
    for (Int i=1;i<=m;i++)
```
Start with the basis of logical variables, i.e., slack and artificial variables.
```
        if (initvars[n+i] >= 0)
            ord[i]=n+i;
        else
```
Keep track of free logical variables (if any) by subtracting $n+m+1$.
```
            ord[i]=-m+i-1;
    for (Int i=0;i<n;i++) {
```
Copy input into columns. Neither column 0 nor row index 0 in each column is used.
```
        Int nvals=sa.col_ptr[i+1]-sa.col_ptr[i];
        a[i+1]=new NRsparseCol(m+1,nvals+1);
        Int count=1;
        for (Int j=sa.col_ptr[i]; j<sa.col_ptr[i+1]; j++) {
            Int k=sa.row_ind[j];
            a[i+1]->row_ind[count]=k+1;
            a[i+1]->val[count]=sa.val[j];
            count++;
        }
    }
    lu=new NRlusol(m,sa.col_ptr[n]);
```
Initialize as $m \times m$. Number of nonzeros is `sa.col_ptr[n]`.

```
    value[0]=0.0;
    value[1]=1.0;
    irow[0]=0;
    for (Int i=1;i<=m;i++) {          Load unit vectors.
        irow[1]=i;
        lu->load_col(i,irow,value);
    }
    lu->factorize();                  Initial factorization of basis (unit matrix).
}

void Simplex::scaleit()
Store scale factors in scale[1:n+m].
{
    for (Int i=1;i<=m;i++)
        scale[n+i]=0.0;
    for (Int k=1;k<=n;k++) {
        x=getcol(k);
        Doub h=0.0;
        for (Int i=1;i<=m;i++)
            if (abs(x[i]) > h)
                h=abs(x[i]);
        if (h == 0.0)
            scale[k]=0.0;
        else
            scale[k]=1.0/h;
        for (Int i=1;i<=m;i++)
            scale[n+i]=MAX(scale[n+i],abs(x[i])*scale[k]);
    }
    for (Int i=1;i<=m;i++)
        if (scale[n+i] == 0.0)
            scale[n+i]=1.0;
}

void Simplex::phase0()
Phase 0 ends when all free variables are in the basis, and all zero variables are out.
{
    Int ind,ip,kp;
    Doub piv;
    for (kp=1;kp<=n;kp++) {           Loop over all columns.
        if (initvars[kp] < 0) {       Found a free variable.
            x=lx(kp);                 x = L^{-1} · a_k .
            w=lu->uinv(x);            w = A_B^{-1} · a_k .
            ip=rowpiv(w,0,kp,0.0);    Find column ip to leave.
            ind=ord[ip];
            transform(x,ip,kp);       Interchange columns.
            ord[ip] -= nm1;           Mark as a free variable.
            if (initvars[ind] == 0)   If a zero variable went out, keep it out.
                ad[ind]=0;
        }
    }
    for (ip=1;ip<=m;ip++) {           Loop over basis.
        ind=ord[ip];
        if (ind < 0 || initvars[ind] != 0)      Skip free variables and nonzero variables.
            continue;
        for (Int i=1;i<=m;i++) v[i]=0.0;
        v[ip]=1.0;                    Column ip will leave.
        kp=colpiv(v,0,piv);           Find entering column kp.
        if (abs(piv) < EPSSMALL) {
            xb=lu->solve(b);          x_B = A_B^{-1} · b.
            if (abs(xb[ip]) > EPSARTIF*maxnorm(xb)) {
                ierr=1;               Zero artificial variable left in basis with nonzero rhs.
                return;
            } else {
```

```
                if (verbose)
                    cout << "    artificial variable remains: ip " << ip << endl;
                continue;
            }
        }
        x=lx(kp);                   x = L^{-1} · a_k.
        transform(x,ip,kp);         Interchange columns.
        if (ad[ind] == 1)
            ad[ind]=0;              Zero variable removed and no longer eligible.
    }
}

void Simplex::phase1()
Phase 1: find a feasible basis.
{
    Int ip,kp;
    Doub piv;
    for (;;) {
        xb=lu->solve(b);            x_B = A_B^{-1} · b.
        Doub xbmax=maxnorm(xb);
        Bool done=true;
        for (Int i=1;i<=m;i++) {
            if (ord[i] > 0 && xb[i] < -EPSFEAS*xbmax) {
                v[i]=1.0/scale[ord[i]];     The auxiliary objective function (scaled).
                done=false;
            }
            else
                v[i]=0.0;
        }
        if (done)                   Done: go to phase 2.
            break;
        kp=colpiv(v,1,piv);         Find entering column kp.
        if (ierr != 0)
            return;
        Bool first=true;
        for (;;) {
            x=lx(kp);               x = L^{-1} · a_k.
            w=lu->uinv(x);          w = A_B^{-1} · a_k.
            ip=rowpiv(w,1,kp,xbmax);
            if (ierr == 0)          Found column ip to leave.
                break;
            if (!first) {           Failure on second attempt: error.
                ierr=6;
                return;
            }
            Maybe accumulated error of updates is too large. Try refactorizing.
            if (verbose)
                cout << "    attempt to recover" << endl;
            ierr=0;
            first=false;
            refactorize();
        }
        transform(x,ip,kp);         Interchange columns.
        if (nsteps >= nmax) {
            ierr=3;
            return;
        }
    }
}

void Simplex::phase2()
Find optimal feasible basis.
{
```

```
    Int ip,kp;
    Doub piv;
    for (;;) {
        for (Int i=1;i<=m;i++) {
            if (ord[i] > 0)          Load minus coefficients of objective function into v.
                v[i]=-obj[ord[i]];
            else
                v[i]=-obj[ord[i]+nm1];
        }
        kp=colpiv(v,2,piv);          Find entering column kp.
        if (piv > -EPSOPT)           The feasible solution has been optimized successfully.
            break;
        Bool first=true;
        for (;;) {
            x=lx(kp);                x = L^{-1} · a_k.
            w=lu->uinv(x);           w = A_B^{-1} · a_k.
            xb=lu->solve(b);         x_B = A_B^{-1} · b.
            Doub xbmax=maxnorm(xb);
            ip=rowpiv(w,2,kp,xbmax);
            if (ierr == 0)           Found column ip to leave.
                break;
            if (!first)
                return;              Unbounded objective function.
            if (verbose)             Maybe accumulated error of updates is too large.
                cout << "    attempt to recover" << endl;    Try refactorizing.
            ierr=0;
            first=false;
            refactorize();
        }
        transform(x,ip,kp);          Interchange columns.
        if (verbose) {
            prepare_output();
            cout << "    in phase2,iter,obj. fn. " << nsteps << " " << u[0] << endl;
        }
        if (nsteps >= nmax) {
            prepare_output();
            cout << "    in phase2,iter,obj. fn. " << nsteps << " " << u[0] << endl;
            ierr=4;
            return;
        }
    }
}

Int Simplex::colpiv(VecDoub &v,Int phase,Doub &piv)
```

Return index kp of variable to enter basis. Also return value of the pivot piv. On input, v must contain minus the coefficients of the objective function. The pivot rule is determined by whether phase is 0, 1, or 2.

```
{
    Int kp;
    Doub h1;
    x=lu->solvet(v);             −(A_B^{-1})^T · c_B.
    piv=0.0;
    for (Int k=1;k<=n+m;k++) {
        if (ad[k] > 0) {         Candidate to enter.
            if (k > n)
                h1=x[k-n];       x dotted with a unit vector.
            else
                h1=xdotcol(x,k); h1 = −a_k · (A_B^{-1})^T · c_B.
            if (phase == 2)
                h1=h1+obj[k];
            h1=h1*scale[k];      Scaled reduced cost.
            if ((phase == 0 && abs(h1) > abs(piv)) ||
                    (phase > 0 && h1 < piv)) {
```

```
                piv=h1;
                kp=k;
            }
        }
    }
    if (phase == 1) {                If no pivot is found in phase 1, the problem is infeasible.
        h1=0.0;
        for (Int k=1;k<=m;k++)
            h1 += abs(x[k])*scale[n+k];
        if (piv > -EPSINFEAS*h1)
            ierr=2;
    }
    return kp;
}

Int Simplex::rowpiv(VecDoub_I &w,Int phase,Int kp,Doub xbmax)
```

Return index ip of variable to leave basis. On input, $\mathbf{w} = \mathbf{A}_B^{-1} \cdot \mathbf{a}_k$, where $\mathbf{a}_k$ is the incoming column. xbmax and xb must have been calculated before calling this routine in phase 1 or 2.

```
{
    Int j=0,ip=0;
    Doub h1,h2,min=0.0,piv=0.0;
    for (Int i=1;i<=m;i++) {         Inspect each current basis vector.
        Int ind=ord[i];
        if (ind > 0) {               Free variables don't leave.
            if (abs(w[i])*scale[kp] <= EPSROW1*scale[ind])
                continue;            Skip zero pivots.
            if (phase == 0) {
                h1=abs(w[i]);
                h2=h1;
            } else {
                Doub hmin=EPSROW2*xbmax*scale[ind]*m;
                if (abs(xb[i]) < hmin)      Degenerate case.
                    h2=hmin;
                else
                    h2=xb[i];
                h1=w[i]/h2;
            }
            if (h1 > 0.0) {
                if (h2 > 0.0 && h1 > piv) {
                    piv=h1;
                    ip=i;
                } else if ((h2*scale[kp] < -EPSROW3*scale[ind]) &&
                    (j == 0 || h1 < min)) {        Only applies in phase one.
                        min=h1;
                        j=i;
                }
            }
        }
    }
    if (min > piv) {
        piv=min;
        ip=j;
    }
    if (ip == 0)
        ierr=5;                      No more pivots in phase 2 signals unbounded objective function.
    return ip;
}

void Simplex::transform(VecDoub &x,Int ip,Int kp)
```

Move column ip out of the basis, column kp in. The vector x must be input as the result of the call x=lx(kp).

```
{
    ad[ord[ip]]=1;                   Column is out but eligible to reenter.
```

```
    ad[kp]=-1;                          Column is in the basis.
    Int oldord=ord[ip];                 Save in case we need to undo the transformation.
    ord[ip]=kp;
    nsteps++;
    if ((nsteps % NREFAC) != 0) {
        Int ok;
        lu->update(x,ip,ok);            Sparse Bartels-Golub update.
        if (ok != 0) {                  Singular update may be due to accumulated round-
            if (verbose)                    off; try to recover by refactorizing.
                cout << "    singular update, refactorize" << endl;
            ad[oldord]=-1;              Restore previous settings.
            ad[kp]=1;
            ord[ip]=oldord;
            refactorize();
        }
    }
    else                                Refactorize every NREFAC iterations.
        refactorize();
}

Doub Simplex::maxnorm(VecDoub_I &xb)
Returns scaled ||x_B||.
{
    Int indv;
    Doub maxn=0.0;
    for (Int i=1;i<=m;i++) {
        if (ord[i] > 0)
            indv=ord[i];
        else
            indv=ord[i]+nm1;
        Doub test=abs(xb[i])/scale[indv];
        if (test > maxn)
            maxn=test;
    }
    if (maxn != 0.0)
        return maxn;
    else
        return 1.0;
}

VecDoub Simplex::getcol(Int k)
Returns column k of A.
{
    VecDoub temp(m+1,0.0);
    for (Int i=1;i<a[k]->nvals;i++) {
        temp[a[k]->row_ind[i]]=a[k]->val[i];
    }
    return temp;
}

VecDoub Simplex::lx(Int kp)
Compute L^{-1} · x, where x is column number kp.
{
    if (kp <= n)
        x=getcol(kp);
    else {                              x is a unit vector.
        for (Int i=1;i<=m;i++) x[i]=0.0;
        x[kp-n]=1.0;
    }
    return lu->linv(x);
}

Doub Simplex::xdotcol(VecDoub_I &x,Int k)
```

Returns $\mathbf{x} \cdot \mathbf{a}_k$, where $\mathbf{a}_k$ is the kth column of $\mathbf{A}$.

```
{
    Doub sum=0.0;
    for (Int i=1;i<a[k]->nvals;i++)
        sum += x[a[k]->row_ind[i]]*a[k]->val[i];
    return sum;
}

void Simplex::refactorize()
```

Refactorize basis by sparse LU decomposition.

```
{
    Int count=0;
    Doub sum=0.0;
    VecInt irow(2);
    VecDoub value(2);
    value[0]=0.0;
    value[1]=1.0;
    irow[0]=0;
    lu->clear();
    for (Int i=1;i<=m;i++) {
        Int ind=ord[i];
        if (ind < 0)
            ind += nm1;
        if (ind > n) {                  Basis vector is a unit vector.
            irow[1]=ind-n;
            lu->load_col(i,irow,value);
            count++;
            sum += 1.0;
        }
        else {                          Basis vector is a column of A.
            lu->load_col(i,a[ind]->row_ind,a[ind]->val);
            for (Int j=1;j<a[ind]->nvals;j++) {
                count++;
                sum += abs(a[ind]->val[j]);
            }
        }
    }
    Doub small=EPSSMALL*sum/count;  Set size of LU factors to treat as zero.
    lu->LUSOL->parmlu[LUSOL_RP_SMALLDIAG_U]    =
        lu->LUSOL->parmlu[LUSOL_RP_EPSDIAG_U]  = small;
    lu->factorize();
}

void Simplex::prepare_output()
```

Compute basic solution. On output, `u[0]` is the value of the objective function, `u[1..n]` contains the values of the variables at the minimum, while `u[n+1..n+m]` gives the values of the slack variables (i.e., the amount by which the corresponding inequality is satisfied).

```
{
    Int indv;
    Doub sum=obj[0];
    xb=lu->solve(b);                    x_B = A_B^{-1} · b.
    for (Int i=0;i<=m+n;i++) u[i]=0.0;
    for (Int i=1;i<=m;i++) {
        if (ord[i] > 0)
            indv=ord[i];
        else
            indv=ord[i]+nm1;
        u[indv]=xb[i];                  Nonzero components of solution.
        sum += xb[i]*obj[indv];
    }
    u[0]=sum;                           c_0 + c · x.
}
```

The `simplex` routine makes use of the external package LUSOL, and so requires the following interface, which is is `NRlusol.h`.

simplex.h

```
extern "C" {
    #include "lusol.h"
}

struct NRlusol
Interface between Numerical Recipes routine Simplex and the required external package LUSOL.
{
    LUSOLrec *LUSOL;
    Int inform;

    NRlusol(Int m, Int nz);
    Constructor creates the LUSOL object and sets the user options. Input are m, the dimension
    of the basis matrix, and nz, the number of nonzeros.
    void load_col(const Int col, VecInt &row_ind, VecDoub &val);
    Loads column number col into basis matrix. The column is in sparse column storage mode,
    with row indices in row_ind and values in val.
    void factorize();                   Sparse LU factorization.
    VecDoub solve(VecDoub &rhs);        Solve A · x = b, with b in rhs.
    VecDoub solvet(VecDoub &rhs);       Solve A^T · x = b, with b in rhs.
    VecDoub linv(VecDoub &rhs);         Return L^{-1} · b, with b in rhs.
    VecDoub uinv(VecDoub &rhs);         Return U^{-1} · b, with b in rhs.
    VecDoub linvt(VecDoub &rhs);        Return (L^{-1})^T · b, with b in rhs.
    VecDoub uinvt(VecDoub &rhs);        Return (U^{-1})^T · b, with b in rhs.
    void update(VecDoub &x, Int i, Int &ok);
    Bartels-Golub update of sparse LU decomposition when column number i of original matrix
    is replaced with column vector x. A return value of ok ≠ 0 indicates that the replacement
    produced a singular update (possibly because of accumulated roundoff error).
    void clear();                       Clear LUSOL structure in preparation for reload-
    ~NRlusol();                             ing basis vectors and refactorizing.
};

NRlusol::NRlusol(Int m, Int nz) {
    LUSOL = LUSOL_create(stdout, 0, LUSOL_PIVMOD_TPP, 0);
    LUSOL->luparm[LUSOL_IP_SCALAR_NZA] = 10;
    LUSOL->parmlu[LUSOL_RP_FACTORMAX_Lij] = 5.0;      Set rather tight pivot tolerances.
    LUSOL->parmlu[LUSOL_RP_UPDATEMAX_Lij] = 5.0;
    LUSOL_sizeto(LUSOL, m, m, nz);
    LUSOL->m = m;
    LUSOL->n = m;
    LUSOL->nelem = nz;
}

void NRlusol::load_col(Int col, VecInt &row_ind, VecDoub &val)
{
    Int nz=row_ind.size()-1;
    Int status=LUSOL_loadColumn(LUSOL,&row_ind[0],col,&val[0],nz,0);
}

void NRlusol::factorize()
{
    LU1FAC(LUSOL, &inform );
    if (inform > LUSOL_INFORM_SERIOUS) {
        cout << "    Error:" << endl << LUSOL_informstr(LUSOL, inform) << endl;
        throw("LUSOL exiting");
    }
}

VecDoub NRlusol::solve(VecDoub &rhs)
{
    VecDoub x(rhs.size()),y=rhs;        Make copy of rhs since this call destroys it.
```

```
    LU6SOL(LUSOL,LUSOL_SOLVE_Aw_v,&y[0],&x[0], NULL, &inform);
    return x;
}

VecDoub NRlusol::solvet(VecDoub &rhs)
{
    VecDoub x(rhs.size()),y=rhs;        Make copy of rhs since this call destroys it.
    LU6SOL(LUSOL,LUSOL_SOLVE_Atv_w,&x[0],&y[0], NULL, &inform);
    return x;
}

VecDoub NRlusol::linv(VecDoub &rhs)
{
    VecDoub x=rhs;
    LU6SOL(LUSOL,LUSOL_SOLVE_Lv_v,&x[0],&x[0], NULL, &inform);
    return x;
}

VecDoub NRlusol::uinv(VecDoub &rhs)
{
    VecDoub x(rhs.size());
    LU6SOL(LUSOL,LUSOL_SOLVE_Uw_v,&rhs[0],&x[0], NULL, &inform);
    return x;
}

VecDoub NRlusol::linvt(VecDoub &rhs)
{
    VecDoub x=rhs;
    LU6SOL(LUSOL,LUSOL_SOLVE_Ltv_v,&x[0],&x[0], NULL, &inform);
    return x;
}

VecDoub NRlusol::uinvt(VecDoub &rhs)
{
    VecDoub x(rhs.size()),y=rhs;        Make copy of rhs since this call destroys it.
    LU6SOL(LUSOL,LUSOL_SOLVE_Utv_w,&x[0],&y[0], NULL, &inform);
    return x;
}

void NRlusol::update(VecDoub &x, Int i, Int &ok)
{
    Doub DIAG, VNORM;
    LU8RPC(LUSOL, LUSOL_UPDATE_OLDNONEMPTY, LUSOL_UPDATE_USEPREPARED,
        i, &x[0], NULL, &ok, &DIAG, &VNORM);
}

void NRlusol::clear()
{
    LUSOL_clear(LUSOL, TRUE);
}

NRlusol::~NRlusol()
{
    LUSOL_free(LUSOL);
}
```

CITED REFERENCES AND FURTHER READING:

Wilkinson, J.H., and Reinsch, C. 1971, *Linear Algebra*, vol. II of *Handbook for Automatic Computation* (New York: Springer).[1]

Netlib: `http://www.netlib.org/lp/`.[2]