SPARTAN: A Sparse Trust-region Algorithm for Nonlinear Equations*

Christopher Maes

August 17, 2009

Abstract

SPARTAN solves square systems of nonlinear equations, $F(x) = 0$, with sparse derivatives. It is written in ANSI/ISO C, with a Scilab and Matlab interface, and released under the GPL. SPARTAN relies on Tim Davis's UMFPACK routines to efficiently solve a sequence of sparse linear systems, and Coleman, Garbow, and Moré's DSM subroutine to compute a column coloring of the Jacobian matrix.

Overview

SPARTAN is a library for solving systems of nonlinear equations

$$F(x) = 0, \tag{1}$$

where F is a function from $\mathbb{R}^n \to \mathbb{R}^n$ that has at least one continuous derivative. The Jacobian matrix $J = F'(x)$ is defined by

$$J_{ij} = \frac{\partial F_i(x)}{\partial x_j}, \quad i, j = 1, \ldots, n.$$

SPARTAN is designed to solve nonlinear systems with sparse Jacobians (those having few nonzero entries).

When not to use SPARTAN

SPARTAN should not be used to compute solutions to $F(x) = 0$ when there are number of equations is not equal to the number of unknowns (*i.e.* $F : \mathbb{R}^m \to \mathbb{R}^n$ with $m \neq n$). That problem should be solved with nonlinear least-squares solvers such as LMDER [10] or levmar [9]. SPARTAN should not be used if there are bounds on the variables (*e.g.* $\ell \leq x \leq u$); in this case, more general sparse nonlinear optimization solvers such as Ipopt[14] or SNOPT [8] are required.

Routines like MINPACK's [10] HYBRD and HYDRJ outperform SPARTAN when n is fairly small (*e.g.* $n \leq 100$) or the Jacobian is dense. The TENSOLVE [2] package can also be used to solve systems with dense derivatives.

Solving a small nonlinear system with SPARTAN

The easiest way to use SPARTAN is within Scilab or Matlab. Suppose we wish to solve the small system of equations [12]

$$\begin{aligned} F_1(x) &= 10(x_2 - x_1^2) &= 0 \\ F_2(x) &= 1 - x_1 &= 0 \end{aligned} \tag{2}$$

with Jacobian matrix J given by

$$J = \begin{pmatrix} -20x_1 & 10 \\ -1 & 0 \end{pmatrix}. \tag{3}$$

In Scilab version x.xx, the system is solved with the following code:

*This work was supported by Google's Summer of Code program and the Scilab Consortium

```
function [f,J] = powell(x)
   f(1) = 10*(x(2) - x(1)^2); f(2) = 1 - x(1);
   J = sparse([ -20*x(1) 10; -1 0]);
endfunction
options = init_param(); options = add_param(options,'Jacobian','on');
x0 = [-3; 4];
x = fsolve(powell,x0,options);
```

For large systems the Jacobian should be constructed in a more efficient manner; see the following section on forming sparse Jacobians. The MATLAB code to solve (2) is similar (with `fsolve` replaced by the MEX function `spartan_solve`). Calling SPARTAN from C is discussed in the section on the C interface.

Singular Jacobians

SPARTAN employees a trust-region algorithm to minimize the merit function $f(x) = \| F(x) \|_2^2$. If f has a global minimum of zero then (1) has a solution. SPARTAN computes a vector x_s that is a local minimum of f, namely, x_s satisfies $J(x_s)^T F(x_s) = 0$. Unfortunately, if $J(x_s)$ is singular and $f(x_s) > 0$ then $F(x_s) \neq 0$ and so x_s is not a solution. In this case, SPARTAN reports convergence to a local minimum and it is suggested that the user resolve with a different initial estimate of the solution.

The trust-region algorithm is capable of handling singular Jacobians that arise during the normal course of algorithm.

Forming a Sparse Jacobian

For large problems it is crucial that the user construct the sparse Jacobian efficiently. When using SCILAB or MATLAB the user should *not* form `J` via `J = sparse(Jdense)`, as is done for the small system above. This method involves creating a $n \times n$ *dense* matrix that requires n^2 elements of storage. This can be prohibitively expensive when n is large. For instance, when $n = 10,000$ the matrix `Jdense` will require 763 megabytes of memory to store.

When `J` is sparse it is best construct `J` in triplet form. For example, the Jacobian in (3) could be constructed with:

```
n = 2;
i = [ 1; 1; 2];
j = [ 1; 2; 1];
v = [ -20*x(1); 10; -1];
```

and the command `sparse([i j], v, [n, n])` in SCILAB, or `sparse(i,j,v,n,n)` in MATLAB.

Checking Derivatives

The most common bug (and cause of failure) is an error in the code that constructs derivatives. SPARTAN contains a routine to check the derivatives provided by the user to ensure they are consistent with the definition of the function.

Users of SCILAB can enable this routine by setting the option `DerivativeCheck` to `on`. For example:

```
options = init_param();  options = add_param(options,'Jacobian','on');
options = add_param(options,'DerivativeCheck','on');
x0 = [-3 4];
x = fsolve(powell,x0,options);
```

Computing Derivatives via Sparse Finite-Differences

Often, it is difficult, or time-consuming to compute the analytical Jacobian. Fortunately, when the Jacobian is sparse it can often be quickly estimated with function values alone via finite-differences.

To estimate derivatives via finite-differences the user must provide the Jacobian's sparsity pattern. Computing the sparsity pattern is much easier then computing the analytical Jacobian. The sparsity pattern $\hat{J}$ is defined as

$$\hat{J}_{ij} = \begin{cases} 1 & \text{if } \frac{\partial F_i(x)}{\partial x_j} \neq 0 \text{ for some } x \in \mathbb{R}^n \\ 0 & \text{otherwise.} \end{cases} \tag{4}$$

A simple rule to follow is that $\hat{J}_{ij} = 1$ if F_i is a function of x_j.

SPARTAN uses the DSM subroutine of Coleman, Garbow, and Moré [3, 4] to compute a coloring of the column graph of $\hat{J}$. The number of colors in this graph coloring indicates the number of function evaluations required to estimate $J(x)$ through finite-differences in the form:

$$\frac{F(x+\epsilon p) - F(x)}{\epsilon} \simeq J(x)p_k, \tag{5}$$

where $p = \sum_{j \in \mathcal{C}_k} e_j$, with e_j the jth column of the identity matrix, and $\mathcal{C}_k$ a set of columns of the same color.

Users of Scilab can enable computing the Jacobian through sparse finite-differences by providing the sparsity pattern $\hat{J}$ via the option `JacobPattern`:

```
function f = powell_nd(x)
   f(1) = 10*(x(2) - x(1)^2); f(2) = 1 - x(1);
endfunction
Jhat = sparse([ 1 1; 1 0]);
options = init_param();  options = add_param(options,'JacobPattern',Jhat);
x0 = [-3 4];
x = fsolve(powell_nd,x0,options);
```

Algorithm

A trust-region algorithm is used to solve the optimization problem

$$\underset{x \in \mathbb{R}^n}{\text{minimize}} \; f(x) \equiv \| F(x) \|_2^2 \tag{6}$$

The gradient $g(x)$ and Hessian $H(x)$ of the merit-function $f(x)$ are given by

$$g(x) \equiv \nabla f(x) = J^T(x)F(x), \quad H(x) \equiv \nabla^2 f(x) = J^T(x)J(x) + \sum_{i=1}^{n} F_i(x)\nabla^2 F_i(x) \tag{7}$$

An approximation to the Hessian $\hat{H}(x) = J(x)^T J(x)$ is used that does not contain the nonconvex (and second derivative) term $\sum_{i=1}^n F_i(x)\nabla^2 F_i(x)$.

At each iteration an approximate solution to the trust-region subproblem

$$\begin{array}{ll} \underset{p}{\text{minimize}} & \| J(x)p + F(x) \| \\ \text{subject to} & \| Dp \| \leq \Delta \end{array} \tag{8}$$

is computed using Powell's dogleg method [13, 12]. The approximate solution $\tilde{p}$ is in the form $\tilde{p} = p_C + \alpha(p_N - p_C)$ for some $\alpha \in [0,1]$. Here p_C is the Cauchy direction $p_C = -\lambda g$ (for some $\lambda \geq 0$) and p_N is the Newton direction defined via $J(x)p_N = -F(x)$. UMFPACK [5] is used to compute a sparse LU factorization of $J(x)$ to solve for p_N. If UMFPACK declares that $J(x)$ is singular then $\tilde{p}$ is taken to be p_C.

If $f(x+p) \leq \eta f(x)$ (for some $\eta \in [0, 1/4)$) the solution estimate x is updated via $x = x + p$. The trust-region radius Δ is adjusted via the algorithm described in [11, pp. 291].

C Interface

The following is a list of SPARTAN's primary routines and data structures:

1. `spartan_index`: the basic integer type used in SPARTAN for indices and dimensions of sparse matrices. If the `LP64` symbol is defined `sizeof(spartan_index) = 8` otherwise `sizeof(spartan_index) = 4`. The 64-bit version of `spartan_index` is useful for large sparse matrices, and compatibility with MATLAB's `-largeArrayDimensions` MEX option.

2. `spartan_sparse`: a sparse matrix stored in either compressed-column format or triplet (coordinate form). The sparse matrix `A`, in compressed column form, contains:

   - `A->p`, a `spartan_index` array of column pointers of size `A->n+1`.
   - `A->i`, a `spartan_index` array of row indices of size `A->nzmax`.
   - `A->x`, a `double` array of size `A->nzmax`.

   The row indices of the jth column of A are stored in `A->i[A->p[j]]` thorough `A->i[A->p[j+1]-1]`. The values of the jth column of A are stored in `A->x[A->p[j]]` through `A->x[A->p[j+1]-1]`.

   Compressed column format is an efficient data structure for storing and factorizing a sparse matrix. However, it can be difficult for users to construct matrices in this form. So, in addition, a sparse matrix can be stored in triplet format. The sparse matrix `A`, in triplet format, contains:

   - `A->p`, a `spartan_index` array of column indices of size `A->nzmax`.
   - `A->i`, a `spartan_index` array of row indices of size `A->nzmax`.
   - `A->x`, a `double` array of size `A->nzmax`.

   The index `A->i[k]` contains the row index i, and the index `A->p[k]` contains the column index j of the `k`th element A_{ij}. The value of the `k`th nonzero element is stored in `A->x[k]`.

   The total number of nonzero elements in the sparse matrix is `A->nzmax`. The value of `A->nz` indicates whether a `spartan_sparse` matrix is stored in compressed sparse column or triplet format.

3. `spartan_spalloc`: Allocates memory for a `spartan_sparse` matrix.

4. `spartan_spfree`: Frees memory used by a `spartan_sparse` matrix.

5. `spartan_entry`: Inserts an entry into a sparse matrix stored in triplet format.

6. `spartan_compress`: Converts a `spartan_sparse` matrix from triplet format to compressed-column format.

7. `spartan_function`: The user provided routine that computes $F(x)$ and stores the result in `f`, a `double` array of size `n`. The routine should `return` zero on success. If the routine returns a nonzero value the solve is stopped.

8. `spartan_jacobian`: The user provided routine that computes $J(x)$ and stores the result in `**J` an `n` by `n` `spartan_sparse` matrix. A `spartan_sparse` matrix can be allocated inside this routine via `*J = spartan_spalloc(...)`. In this case the option `SPARTAN_MEMMGMT` should be set so SPARTAN can free the matrix with `spartan_spfree`.

9. `spartan_solve`: SPARTAN's main routine to compute a solution x such that $F(x) = 0$.

C Example

The following code shows how to use SPARTAN to solve the system:

$$\begin{aligned} F_1(x) &= (3 - hx_1)x_1 - 2x_2 + 1 \\ F_i(x) &= (3 - hx_i)x_i - x_{i-1} - 2x_{i+1} + 1, \quad i = 2, \ldots, n-1 \quad \text{with} \quad J(x)_{ij} = \begin{cases} -1 & j = i-1 \\ 3 - 2hx_i & j = i \\ -2 & j = i+1 \end{cases} \\ F_n(x) &= (3 - hx_{n-1})x_{n-1} - x_{n-1} + 1 \end{aligned} \tag{9}$$

```
#include <stdio.h>
#include "spartan.h"
double h = 0.5;
int broydenfunc(double *f, const double *x, spartan_index n){
  spartan_index i;
  f[0] = (3 - h*x[0])*x[0] - 2*x[1] + 1;
  for(i=1; i<=n-2; i++)
    f[i] = (3 - h*x[i])*x[i] - x[i-1] - 2*x[i+1] + 1;
  f[n-1] = (3 - h*x[n-1])*x[n-1] - x[n-1] + 1;
  return(0);
}

int broydenjacob(spartan_sparse **J, const double *x, spartan_index n){
  spartan_index nnz, i; spartan_sparse *T;
  nnz = 2 + 3*(n-2) + 2;    /* The number of nonzeros in the tridiagonal jacobian */
  T = spartan_spalloc(n,n,nnz,1,1);   /* Allocate Jacobian in triplet form*/
  if (T == NULL) return(-1);   /* Check for out of memory error */
  for (i = 0; i < n; i++){
    if (i > 0) spartan_entry(T,i,i-1,-1);      /* sub diagonal entries */
    spartan_entry(T, i, i, 3 - 2*h*x[i]);      /* Diagonal entries */
    if (i < n-1) spartan_entry(T,i, i+1, -2);  /* superdiagonal entries */
  }
  *J = spartan_compress(T); /* convert from triplet to compressed-col */
  T = spartan_spfree(T);   /* free matrix triplet matrix */
  return(0);
}

void printstring(const char *s){ printf("%s\n",s); }

int main(int argc, char **argv){
  spartan_index j, n = 1024;
  double *f, *x, opts[SPARTAN_OPTIONS];
  int status, i;
  f = malloc(sizeof(double)*n);   /* Allocate space for f and x */
  x = malloc(sizeof(double)*n);
  if (!x || !f) return(-1);       /* Check for out of memory errors */
  for(i=0; i<SPARTAN_OPTIONS; i++) opts[i] =0;    /* Set spartan's default options */
  opts[SPARTAN_MEMMGMT] = 1;   /* Tell spartan to free the Jacobian after use */
  spartan_setprintfunction(printstring);   /* Set the print function for spartan output */
  for (j=0;j<n;j++) x[j] = -1.;   /* Set x = -1 */
  status = spartan_solve(broydenfunc, broydenjacob, n, f, x, opts); /* Solve the problem */
  free(x); free(f); /* Clean up*/
  return(0);
}
```

Spartan C Routines

This section describes the routines in SPARTAN. Routines marked with (†) are from CSPARSE [6]. The parameters and return values are labelled as one of the following:

- **in:** The parameter must be defined on input. It is not modified.
- **in/out:** The parameter must be defined on input. It is modified on output.
- **returns:** The return value of each function. Functions that return a pointer return `NULL` if an error occurs.

Primary Routines

spartan_index: **sparse matrix index**

All matrix indices and dimensions in SPARTAN are of type `spartan_index`. The size of `spartan_index` can be controlled at compile time. By default `spartan_index` is a 4-byte integer. Defining the symbol `LP64` (or `MATLAB_MEX`) forces `spartan_index` to be an 8-byte integer.

spartan_sparse: **sparse matrix in compressed-column or triplet form†**

```
typedef struct spartan_sp {
  spartan_index nzmax; /* maximum number of entries */
  spartan_index  m;    /* number of rows */
  spartan_index  n;    /* number of cols */
  spartan_index *p;    /* column pointers (size n+1) or col indices (size nzmax)*/
  spartan_index *i;    /* row indices, size nzmax */
  double        *x;    /* numerical values, size nzmax */
  spartan_index nz;    /* # of entries in triplet form, -1 for compressed-col*/
} spartan_sparse ;
```

spartan_solve: **solve a system of nonlinear equations**

```
int spartan_solve(spartan_function usrfun, spartan_jacobian usrjac, spartan_index n, double *fval,
                  double *x, double *opts);
```

Solves a system of nonlinear equations with sparse derivatives.

`usrfun`	in	pointer to a user-defined function that evaluates $F(x)$
`usrjac`	in	pointer to a user-defined function that evaluates $J = F'(x)$
`n`	in	the size of the system
`fval`	out	function value at the solution
`x`	in/out	on input an initial estimate of solution; on output the solution of the system
`opts`	in	options
	returns	`SPARTAN_OK` if sucessful, `SPARTAN_MEMORY_FAILURE` if unable to allocate sufficient memory, `SPARTAN_USER_ERROR` if `usrfun` returns nonzero, `SPARTAN_LOCAL_MIN` if converged to local minimum, `SPARTAN_ITER_LIMIT` if iteration limit reached

Utility Routines

spartan_spalloc: **allocate a sparse matrix†**

```
spartan_sparse *spartan_spalloc (spartan_index m, spartan_index n, spartan_index nzmax,
                                 int values, int triplet) ;
```

Allocates a sparse matrix in either compressed-column or triplet form.

`m`	in	number of rows
`n`	in	number of columns
`nzmax`	in	maximum number of entries
`values`	in	allocate pattern only if `0`, values and pattern otherwise
`triplet`	in	compressed-column if `0`, triplet form otherwise
	returns	`A` if successful; `NULL` on error

`spartan_spfree`: free a sparse matrix†

```
spartan_sparse *spartan_spfree (spartan_sparse *A) ;
```

Frees a sparse matrix, in either compressed-column or triplet form.

`A`	in/out	sparse matrix to free
	returns	`NULL`

`spartan_entry`: add an entry to a triplet form matrix†

```
int spartan_entry(spartan_sparse *T, spartan_index i, spartan_index j, double x) ;
```

Memory-space and dimension of `T` are increased if necessary.

`T`	in/out	triplet matrix; new entry added on output
`i`	in	row index of new entry
`j`	in	column index of new entry
`x`	in	numerical value of new entry
	returns	`1` if successful; `0` on error

`spartan_compress`: triplet form to compressed-column conversion†

```
spartan_sparse *spartan_compress (const spartan_sparse *T) ;
```

Converts a triplet-form matrix `T` into a compressed-column matrix `C`. The columns of `C` are not sorted, and duplicate entries may be present in `C`.

`T`	in	sparse matrix in triplet form
	returns	`C` if successful; `NULL` on error

`spartan_setprintfunction`: set routine to print output

Internal Routines

These routines are used internally by SPARTAN. They are documented here for completeness. The interface to these routines is subject to change; the user should not rely on these routines to persist between different version of SPARTAN.

Computational Routines

`spartan_dogleg`: computes an approximate solution to the trust-region subproblem

`spartan_gaxpy`: $y \leftarrow Ax + y$†

`spartan_gatxpy`: $y \leftarrow A^T x + y$

Vector operations (Level-1 BLAS)

`spartan_dnrm2`: $\| x \|_2$

`spartan_ddot`: $y^T x$

`spartan_dscal`: $x \leftarrow \alpha x$

`spartan_daxpy`: $y \leftarrow \alpha x + y$

`spartan_dcopy`: $y \leftarrow x$

`spartan_dmult`: $z \leftarrow x.*y$

`spartan_norminf`: $\| x \|_\infty$

`spartan_zero`: $x \leftarrow \mathbf{0}$

Memory routines

`spartan_calloc`: **allocate and clear memory†**

`spartan_malloc`: **allocate memory†**

`spartan_realloc`: **change size of a block of memory†**

`spartan_free`: **free memory†**

Output routines

`spartan_print`: **print strings**

Sparse Utility routines

`spartan_cumsum`: **computes the cumulative sum of an array†**

`spartan_norms`: **compute scaled column norms**

`spartan_done`: **free memory and return a sparse matrix †**

License

SPARTAN is released under the GPL. SPARTAN utilizes the GPL library UMFPACK [5], derives code from the Lesser GPL routines CSPARSE [6], and uses the DSM subroutine from ACM TOMS Algorithm 618 [3] released under the ACM Software License Agreement [1].

Further Work

SPARTAN is an experimental solver. The following additional work would greatly improve it:

- Currently, `spartan_linsolve` computes a symbolic factorization of the Jacobian at each iteration. This is unnecessary if the sparsity pattern of the Jacobian is fixed and known. This is the case when `spartan_sfdjac` is used to compute sparse finite-differences and when the user knows the sparsity pattern. But, not when numerically zeros entries are removed from the Jacobian matrix by Scilab or Matlab. Add a function `spartan_setpattern` that can be called by the user to provide a sparsity pattern, even when not computing the Jacobian via finite-differences, and modify `spartan_initsfd` to use this function. Update the Scilab gateway interface and Matlab MEX interface to call `spartan_setpattern` when the `JacobPattern` option is used.

- Add derivative checking to the C interface. See `checkderivative.sci` for a Scilab implementation.

- SPARTAN's main computational routines `spartan_solve` and `spartan_dogleg` use Level-1 BLAS vector operations. These BLAS operations are contained and implemented in `spartan_blas.c`. They do not check for overflows, compute sums in numerically stable manner, or take advantage of multiple processors. However, these routines are portable and do not require an external BLAS library. Add code to make the routines in `spartan_blas.c` call external BLAS routines. Add the symbols `BLAS` and `LBLAS` that, when defined at compile-time, cause these external BLAS routines to be used. Be sure to handle the case when `spartan_index` is a 64-bit integer (the `LBLAS` option).

- Develop a large set of sparse test problems and produce performance benchmarks.

- Develop the Matlab MEX interface.

- Investigate using COLPACK [7] instead of DSM to color the column graph of the Jacobian.

- Investigate the performance of different methods for solving the trust-region subproblem (8). Other methods include: searching in the two-dimensional subspace span$\{g, -J^{-1}g\}$, the double-dogleg method, or solving the problem exactly.

References

[1] ACM Software Copyright and License Agreement, August 2009. http://www.acm.org/publications/softwarecrnnotice.

[2] Bouaricha, A., and Schnabel, R. B. Algorithm 768: TENSOLVE: a software package for solving systems of nonlinear equations and nonlinear least-squares problems using tensor methods. *ACM Trans. Math. Softw. 23*, 2 (1997), 174–195.

[3] Coleman, T. F., Garbow, B. S., and Moré, J. J. Algorithm 618: FORTRAN subroutines for estimating sparse jacobian matrices. *ACM Trans. Math. Softw. 10*, 3 (1984), 346–347.

[4] Coleman, T. F., and Moré, J. J. Estimation of sparse jacobian matrices and graph coloring problems. *SIAM J. Numer. Anal. 20*, 1 (February 1983), 187–209.

[5] Davis, T. A. Algorithm 832: UMFPACK v4.3—an unsymmetric-pattern multifrontal method. *ACM Trans. Math. Softw. 30*, 2 (2004), 196–199. http://www.cise.ufl.edu/research/sparse/umfpack/.

[6] Davis, T. A. *Direct Methods for Sparse Linear Systems.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2006. http://www.cise.ufl.edu/research/sparse/CSparse/.

[7] Gebremedhin, A. H. Graph coloring for computing derivatives, August 2009. http://www.cscapes.org/coloringpage.

[8] Gill, P., Murray, W., and Saunders, M. SNOPT 7.0 description, August 2009. http://www.sbsi-sol-optimize.com/asp/sol_product_snopt.htm.

[9] Lourakis, M. levmar: Levenberg-Marquardt nonlinear least squares algorithms in C/C++, August 2009. http://www.ics.forth.gr/~lourakis/levmar/.

[10] Moré, J. J., Garbow, B. S., and Hillstrom, K. E. User guide for MINPACK-1. Tech. rep. http://www-unix.mcs.anl.gov/~more/ANL8074a.pdf.

[11] Nocedal, J., and Wright, S. J. *Numerical Optimization*, second ed. Springer Verlag, New York, 2006.

[12] Powell, M. J. D. A FORTRAN subroutine for solving systems of nonlinear algebraic equations. In *Numerical Methods for Nonlinear Algebraic Equations*, P. Rabinowitz, Ed. Gordon and Breach, Science Publishers Ltd., 1970, pp. 115–161.

[13] Powell, M. J. D. A hybrid method for nonlinear equations. In *Numerical Methods for Nonlinear Algebraic Equations*, P. Rabinowitz, Ed. Gordon and Breach, Science Publishers Ltd., 1970, pp. 87–114.

[14] Wächter, A., and Biegler, L. T. Ipopt, August 2009. https://projects.coin-or.org/Ipopt.