# Constant-Aware Sparse Newton

info@stce.rwth-aachen.de

**Given** We consider the computation of solutions $\mathbf{x} = \mathbf{x}(\mathbf{p})$ of differentiable parameterized sparse systems of nonlinear equations

$$F(\mathbf{x}, \mathbf{p}) = 0$$

with residual

$$F : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^n : \begin{pmatrix} \mathbf{x} \\ \mathbf{p} \end{pmatrix} \mapsto \mathbf{y} = F(\mathbf{x}, \mathbf{p})$$

using Newton's method [1]. The Jacobian

$$\frac{dF}{d\mathbf{x}} \equiv F'$$

can be computed by algorithmic differentiation (AD) using dco/c++ [4]. It decomposes into variable

$$F'_v = F'_v(\mathbf{x}, \mathbf{p}) \in \mathbb{R}^{n \times n}$$

and constant

$$F'_c = F'_c(\mathbf{p}) \in \mathbb{R}^{n \times n}$$

submatrices with disjoint sparsity patterns $S(F'_v)$ and $S(F'_c)$ as $F' = F'_v + F'_c$. Either one of them can potentially vanish identically.

Corresponding convex unconstrained nonlinear minimization problems

$$\min_{\mathbf{x}} f(\mathbf{x}, \mathbf{p})$$

with objective

$$f : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R} : \begin{pmatrix} \mathbf{x} \\ \mathbf{p} \end{pmatrix} \mapsto y = f(\mathbf{x}, \mathbf{p})$$

take their minimum at the solution of the necessary optimality condition

$$f'(\mathbf{x}, \mathbf{p}) = 0$$

which can be computed using Newton's method. A sequence of linear systems with system matrix (Hessian)

$$\frac{d^2 f}{d\mathbf{x}^2} \equiv f'' = f''(\mathbf{x}, \mathbf{p})$$

needs to be solved. A minimum requires satisfaction of the sufficient optimality condition

$$\hat{\mathbf{x}}^T \cdot f''(\mathbf{x}, \mathbf{p}) \cdot \hat{\mathbf{x}} > 0$$

for all $0 \neq \hat{\mathbf{x}} \in \mathbb{R}^n$. The Hessian is assumed to decompose into variable

$$f''_v = f''_v(\mathbf{x}, \mathbf{p}) \in \mathbb{R}^{n \times n}$$

and constant

$$f_c'' = f_c''(\mathbf{p}) \in \mathbb{R}^{n \times n}$$

submatrices with disjoint sparsity patterns $S(f_v'')$ and $S(f_c'')$ as $f'' = f_v'' + f_c''$. Again, either one of them can potentially vanish identically.

Several state of the art methods for solving constrained nonlinear optimization problems rely on the solution of systems of nonlinear equations representing necessary optimality conditions. For example, solving a convex equality constrained nonlinear optimization problem amounts to the solution of the so-called KKT system, see, e.g, [5] for further information.

**Wanted** Software infrastructure for Newton's method for differentiable parameterized sparse systems of nonlinear equations $F(\mathbf{x}, \mathbf{p}) = 0$ as well as for corresponding convex unconstrained nonlinear objectives $f(\mathbf{x}, \mathbf{p})$ exploiting sparsity and partial constantness including

1. computation of gradients using adjoint AD with dco/c++ ($\Rightarrow f'$)
2. linearity analysis ($\Rightarrow S(F_c')$ or $S(f_c'')$)
3. Jacobian compression using ColPack and dco/c++ ($\Rightarrow S(F_c')$ or $S(f_c'')$)
4. solution of sparse Newton system using Eigen[1]

A compressed Jacobian $U \in \mathbb{R}^{n \times k}$ can be computed using dco/c++ [4] as

$$U = F' \cdot V$$

with the seed matrix $V \in \mathbb{R}^{n \times k}$ computed, for example, by coloring of the column incidence graph [2] using ColPack [3]. With $F' = F_v' + F_c'$ we get

$$U = (F_v' + F_c') \cdot V_v$$

yielding the linear system

$$F_v' \cdot V_v = U - F_c' \cdot V_v \tag{1}$$

where $V_v \in \mathbb{R}^{n \times k_v}$, $k_v \leq k$ is the seed matrix of $F_v'$.

In the context of Newton's algorithm the Jacobian $F'$ is computed once (first iteration) yielding $F_c'$ and followed by potentially less costly iterations based on Equation (1). Coloring can also be applied to during the first iteration provided that the overhead in computational cost can be amortized by the compressed computation of $F'$. Alternatively, compression is applied to $F_v'$ only. The overhead in computational cost is more likely to be amortized due to reapplication of the compression during the second and subsequent Newton iterations.

[1] `https://eigen.tuxfamily.org/`

**Example** We illustrate the use of the sparsity patterns for linearity analysis of a simple example. Consider

$$y = f(\mathbf{x}) = p \cdot x_0 + x_1^2$$

The gradient is equal to

$$f' = \begin{pmatrix} p \\ 2 \cdot x_1 \end{pmatrix}$$

with sparsity pattern

$$S(f') = \begin{pmatrix} 1 \\ 1 \end{pmatrix} .$$

Both can be computed with dco/c++ as shown in the reference code listed in the appendix. Adjoint mode (e.g. `dco::ga1s<double>`) is typically used for the computation of gradients of multivariate scalar functions. dco/c++ provides a special pattern data type (e.g. `dco::p1f::type`) for the computation of sparsity patterns of Jacobians based on the given primal.

The Hessian is equal to

$$f'' = \begin{pmatrix} 0 & 0 \\ 0 & 2 \end{pmatrix}$$

with sparsity pattern

$$S(f'') = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}$$

and hence

$$S(f'') \cdot \mathbf{e} = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} .$$

Both $f''$ and $S(f'') \cdot \mathbf{e}$ can be computed with dco/c++ as shown in the reference code listed in the appendix. Second-order adjoint mode (e.g., tangent of adjoint mode using `dco::ga1s<dco::gt1s<double>::type>`) is typically used for the computation of Hessians of multivariate scalar functions. dco/c++'s pattern data type can be used to compute the sparsity patterns of the Hessian based on the given first-order adjoint gradient code.

Consequently,

$$C(f') = S(f') \ \texttt{XOR} \ S(f'') \cdot \mathbf{e} = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \ \texttt{XOR} \ \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

implying that the first gradient entry is constant.

**Todo** Design, implement, test, optimize and document the sparse Newton software infrastructure including

1. user interfaces for nonlinear systems

```
template<typename T, typename TP, size_t N, size_t NP>
void F(
  const std::array<T,N>& x,
  const std::array<TP,NP>& p,
  std::array<T,N>& y
);
```

and nonlinear objectives

```
template<typename T, typename TP, size_t N, size_t NP>
void f(
  const std::array<T,N>& x,
  const std::array<TP,NP>& p,
  T& y
);
```

2. use of dco/c++ for the computation of Jacobians, gradients, Hessians and of required sparsity patterns;
3. use of ColPack for Jacobian and Hessian compression based on graph coloring;
4. use of Eigen for sparse linear algebra (sparse matrix data format, sparse linear solvers);
5. validation of correctness of the software by implementing an appropriate test problem suite (e.g. discretizations of nonlinear partial differential equations);
6. study of runtimes of different approaches (dense vs. sparse vs. constant-aware sparse) based on the test problem suite;
7. documentation of the source code using doxygen;
8. presentation of a report including user and developer documentations.

**Further issues to explore (optional)**

1. Think about a custom coloring algorithm for $F'$ which minimizes the number of colors used for $F'_v$. A single coloring step would be suffice to obtain good compression for $F'$ as well as for $F'_v$.
2. Can constant-awareness be exploited by solvers of the linear Newton system?

# References

[1] W. Dahmen and A. Reusken. *Numerik fur Ingenieure und Naturwissenschaftler*. Springer-Verlag, Germany, Berlin, 2008.

[2] A. Gebremedhin, F. Manne, and A. Pothen. What color is your Jacobian? graph coloring for computing derivatives. *SIAM Review*, 47(4):629–705, 2005.

[3] A. Gebremedhin, D. Nguyen, M. Patwary, and A. Pothen. Colpack: Software for graph coloring and related problems in scientific computing. *ACM Trans. Math. Softw.*, 40(1):1:1–1:31, oct 2013.

[4] K. Leppkes, J. Lotz, and U. Naumann. Derivative Code by Overloading in C++ (dco/c++): Introduction and Summary of Features. Technical Report AIB-2016-08, RWTH Aachen University, September 2016. http://aib.informatik.rwth-aachen.de/2016/2016-08.pdf.

[5] J. Nocedal and S. Wright. *Numerical Optimization*. Springer, New York, NY, USA, second edition, 2006.

# A Reference Implementation with dco/c++

```
#include "dco.hpp"

#include <iostream>
#include <array>
#include <cmath>

template<typename T, typename TP, size_t N, size_t NP>
void f(
  const std::array<T,N>& x,
  const std::array<TP,NP>& p,
  T& y
) {
  using namespace std;
  y=p[0]*x[0]+sin(x[1]);
}

template<typename T, typename TP, size_t N, size_t NP>
void df(
  const std::array<T,N>& xv,
  const std::array<TP,NP>& p,
  T& yv,
  std::array<T,N>& dydx
) {
  typedef typename dco::ga1s<T> DCO_M;
  typedef typename DCO_M::type DCO_T;
  typedef typename DCO_M::tape_t DCO_TT;
  DCO_M::global_tape=DCO_TT::create();
  std::array<DCO_T,N> x; DCO_T y;
  for (size_t i=0;i<N;i++) x[i]=xv[i];
  for (auto& i:x) DCO_M::global_tape->register_variable(i);
```

```
31    f(x,p,y);
32    yv=dco::value(y);
33    DCO_M::global_tape->register_output_variable(y);
34    dco::derivative(y)=1;
35    DCO_M::global_tape->interpret_adjoint();
36    for (size_t i=0;i<N;i++) dydx[i]=dco::derivative(x[i]);
37    DCO_TT::remove(DCO_M::global_tape);
38  }
39
40  template<typename T, typename TP, size_t N, size_t NP>
41  void ddf(
42    const std::array<T,N>& xv,
43    const std::array<TP,NP>& p,
44    T& yv,
45    std::array<T,N>& dydx_v,
46    std::array<std::array<T,N>,N>& ddydxx
47  ) {
48    typedef typename dco::gt1v<T,N>::type DCO_T;
49    std::array<DCO_T,N> x,dydx; DCO_T y;
50    for (size_t i=0;i<N;i++) {
51      x[i]=xv[i];
52      dco::derivative(x[i])[i]=1;
53    }
54    df(x,p,y,dydx);
55    yv=dco::value(y);
56    for (size_t i=0;i<N;i++) {
57      dydx_v[i]=dco::value(dydx[i]);
58      for (size_t j=0;j<N;j++)
59        ddydxx[i][j]=dco::derivative(dydx[i])[j];
60    }
61  }
62
63  template<typename T, typename TP, size_t N, size_t NP>
64  void Sdf(
65    const std::array<T,N>& xv,
66    const std::array<TP,NP>& p,
67    T& yv,
68    std::array<bool,N> &sdf
69  ) {
70    using DCO_T=dco::p1f::type;
71    std::array<DCO_T,N> x; DCO_T y;
72    for (size_t i=0;i<N;i++) {
73      x[i]=xv[i];
74      dco::p1f::set(x[i],true,i);
75    }
76    f(x,p,y);
77    dco::p1f::get(y,yv);
78    for (size_t i=0;i<N;i++) dco::p1f::get(y,sdf[i],i);
79  }
80
```

```
template<typename T, typename TP, size_t N, size_t NP>
void dSddf(
  const std::array<T,N>& xv,
  const std::array<TP,NP>& p,
  T& yv,
  std::array<bool,N> &dsddf
) {
  using DCO_T=dco::p1f::type;
  std::array<DCO_T,N> x,dydx; DCO_T y;
  for (size_t i=0;i<N;i++) {
    x[i]=xv[i];
    dco::p1f::set(x[i],true,0);
  }
  df(x,p,y,dydx);
  dco::p1f::get(y,yv);
  for (size_t i=0;i<N;i++) dco::p1f::get(dydx[i],dsddf[i],0);
}

int main() {
  using T=double; using TP=float;
  const size_t N=2, NP=1;
  std::array<T,N> x={1,1};
  std::array<TP,NP> p={1.1};
  T y;

  std::cout << "f:" << std::endl;
  f(x,p,y);
  std::cout << y << std::endl;

  std::cout << "df:" << std::endl;
  std::array<T,N> dydx;
  df(x,p,y,dydx);
  for (const auto& i:dydx) std::cout << i << std::endl;

  std::cout << "ddf:" << std::endl;
  std::array<std::array<T,N>,N> ddydxx;
  ddf(x,p,y,dydx,ddydxx);
  for (const auto& i:ddydxx)
    for (const auto& j:i)
    std::cout << j << std::endl;

  std::cout << "Sdf:" << std::endl;
  std::array<bool,N> sdf;
  Sdf(x,p,y,sdf);
  for (const auto& i:sdf) std::cout << i << std::endl;

  std::cout << "dSddf:" << std::endl;
  std::array<bool,N> dsddf;
  dSddf(x,p,y,dsddf);
  for (const auto& i:dsddf) std::cout << i << std::endl;
```

```
131
132   std::cout << "Cdf:" << std::endl;
133   for (size_t i=0;i<N;i++)
134     std::cout << (sdf[i]!=dsddf[i]) << std::endl;
135   return 0;
136 }
```

The following output is generated.

```
1  f:
2  1.94147
3  df:
4  1.1
5  0.540302
6  ddf:
7  0
8  0
9  0
10 -0.841471
11 Sdf:
12 1
13 1
14 dSddf:
15 0
16 1
17 Cdf:
18 1
19 0
```