DIFFOP - Differential operators in MATLAB without the pain

Table of Contents

Introduction and Quickstart

`DIFFOP` is a library to quickly generate sparse matrices from linear `MATLAB` expressions.

The following simple example computes the second derivatives of a function using the usual `[1 -2 1]` stencil with Neumann boundary conditions, and verifies that it works:

```
function diffop_demo

function result = second_deriv(f)
    f_ex = [f(1); f; f(end)]; % expand using ghost cells
    ii = 2:(numel(f_ex)-1);
    result = f_ex(ii+1) + f_ex(ii-1) - 2 * f_ex(ii);
end

f = [2 5 -4 1]';
fxx1 = second_deriv(f);
fxx1'
```

```
        ans =

             3   -12    14    -5



        ans =

            -1     1     0     0
             1    -2     1     0
             0     1    -2     1
             0     0     1    -1
```

While this works for simply applying the linear operator, there are many cases where it is actually preferable to represent the *function* as a *matrix* instead:

- Writing `fxx = A * f(:)` makes immediately clear that we are evaluating a linear operator.

- It is easy to pass the operator including information about its dimensions to other functions, and to *combine* operators just by concatenating or multiplying their associated matrices.
- It is trivial to compute the *adjoint* (transpose) of the operator, which is often required in numerical optimization methods.
- The operator can also be easily (pseudo-) *inverted* using standard `MATLAB` functions for solving linear equation systems.

For `second_deriv()`, the associated matrix is tridiagonal and can be constructed as follows:

```
[1 2 3]

n = numel(f);
A = spdiags([ones(n,1), -2*ones(n,1), ones(n,1)],[-1 0 1],n,n);
A(1,1) = -1;   % Adjust for Neumann boundary conditions
A(end,end) = -1;
full(A)        % convert to dense matrix for display
```

```
        ans =

             1     2     3


        ans =

            -1     1     0     0
             1    -2     1     0
             0     1    -2     1
             0     0     1    -1
```

This creates a sparse $n \times n$ matrix with `[-1, 2, -1]` on the diagonals. It computes the same as `second_deriv()`:

```
fxx2 = A * f(:);
fxx2'
norm(fxx2 - fxx1, +inf) % zero up to numerical precision
```

```
        ans =

             3   -12    14    -5


        ans =

             0
```

For this simple example the code doesn't look too bad, but once we step into multiple dimensions, different boundary conditions, local weights etc., constructing the associated matrix manually becomes not only annoying but also a reliable source of discretization errors.

This is what computing `A` looks like using `DIFFOP`:

```
f_vars = spvar([n,1]); % create 4 x 1 vector of variables
op = second_deriv(f_vars);

full(op.A())  % op.A() returns the matrix form of second_deriv()
```

This automatically generates the matrix form of `second_deriv()` and makes it accessible through `op.A()`. As long as only linear operations are used, the same works for almost any other code instead of `second_deriv()` as well, see below for some more advanced examples.

This means we can implement the operator in the most natural way, but still have all the benefits of having access to a matrix representation.

How it works

`DIFFOP` relies on a custom class for the variables and heavy operator overloading. When creating a new set of variables as in

```
x = spvar([4 1]);
```

`spexpr()` actually returns an **object** of class `spexpr`:

```
class(x)
```

```

ans =

spexpr

```

The object knows its size as passed to `spvar`

```
x
```

```

x =

variables, dimensions = 4  1
```

but internally stores the matrix representation of all linear operations that have been applied to it. For the plain set of variables, A is thus the identity matrix:

```
full(x.A())
```

```

ans =

     1     0     0     0
     0     1     0     0
     0     0     1     0
     0     0     0     1

```

Other linear operations behave in a similar way:

```
x1 = 5 * x;
full(x1.A()) % all entries of A are multiplied by 5
```

```
        ans =

             5     0     0     0
             0     5     0     0
             0     0     5     0
             0     0     0     5


x2 = x(1:2);
full(x2.A()) % a subset of rows of A is extracted


        ans =

             1     0     0     0
             0     1     0     0


x3 = x(1) + x(2);
full(x3.A()) % rows of A are added


        ans =

             1     1     0     0
```

and so on. This is a simplified version of the concept used by tools for automatic differentiation and libraries such as CVX, specialized to the linear case.

Higher-dimensional data

DIFFOP can also easily be used for two- and higher-dimensional data. First we implement the two-dimensional forward difference operator:

```
function result = forward_diff(u)
    result = {u(2:end,1:end-1) - u(1:end-1,1:end-1),...
              u(1:end-1,2:end) - u(1:end-1,1:end-1)};
end
```

This operator returns **two** results as a cell array, one for the gradient of u in the x direction, and one for the gradient in the y direction:

```
n = 3; m = 4; % small 3x4 domain for demonstration
```

forward_diff can be used in exactly the same way as in the simple one-dimensional example, except that we create a two-dimensional $n \times m$ array of variables instead of a vector:

```
u = spvar([n m]);
u_neu = [u        u(:,end);...
         u(end,:) 0        ]; % extend to Neumann boundary conditions

G_neu = forward_diff(u_neu)


        G_neu =
          [3x4 spexpr]    [3x4 spexpr]
```

`G_neu` is now a cell array as well, and contains the matrices for evaluating the x and y derivatives:

```
full(G_neu{1}.A())
full(G_neu{2}.A())
```

```
        ans =

            -1     1     0     0     0     0     0     0     0     0     0     0
             0    -1     1     0     0     0     0     0     0     0     0     0
             0     0     0     0     0     0     0     0     0     0     0     0
             0     0     0    -1     1     0     0     0     0     0     0     0
             0     0     0     0    -1     1     0     0     0     0     0     0
             0     0     0     0     0     0     0     0     0     0     0     0
             0     0     0     0     0     0    -1     1     0     0     0     0
             0     0     0     0     0     0     0    -1     1     0     0     0
             0     0     0     0     0     0     0     0     0     0     0     0
             0     0     0     0     0     0     0     0     0    -1     1     0
             0     0     0     0     0     0     0     0     0     0    -1     1
             0     0     0     0     0     0     0     0     0     0     0     0


        ans =

            -1     0     0     1     0     0     0     0     0     0     0     0
             0    -1     0     0     1     0     0     0     0     0     0     0
             0     0    -1     0     0     1     0     0     0     0     0     0
             0     0     0    -1     0     0     1     0     0     0     0     0
             0     0     0     0    -1     0     0     1     0     0     0     0
             0     0     0     0     0    -1     0     0     1     0     0     0
             0     0     0     0     0     0    -1     0     0     1     0     0
             0     0     0     0     0     0     0    -1     0     0     1     0
             0     0     0     0     0     0     0     0    -1     0     0     1
             0     0     0     0     0     0     0     0     0     0     0     0
             0     0     0     0     0     0     0     0     0     0     0     0
             0     0     0     0     0     0     0     0     0     0     0     0
```

To apply the new operator to numeric data, we can use `apply()`:

```
x = rand(n,m);
g1 = G_neu{1}.apply(x)
```

```
        g1 =

           -0.4233   -0.7597    0.6214   -0.2162
            0.0547    0.4789   -0.2467    0.2588
                 0         0         0         0
```

Alternatively, we can use the manual form

```
g2 = reshape(G_neu{1}.A() * x(:), size(G_neu{1}))
```

```
g2 =

   -0.4233   -0.7597    0.6214   -0.2162
    0.0547    0.4789   -0.2467    0.2588
         0         0         0         0
```

Affine operators

`DIFFOP` is not restricted to linear operators, but also handles constant terms. The code below shows an example with inhomogeneous Dirichlet boundary conditions, i.e., u is assumed to be 1 on the boundary:

```
u_inhom = [u         ones(n,1);...
           ones(1,m) 0        ];
G_inhom = forward_diff(u_inhom);
```

The *affine* parts of the operator are returned by the `b()` function:

```
full(G_inhom{1}.A()), G_inhom{1}.b()
full(G_inhom{2}.A()), G_inhom{2}.b()
```

```
ans =

    -1     1     0     0     0     0     0     0     0     0     0     0
     0    -1     1     0     0     0     0     0     0     0     0     0
     0     0    -1     0     0     0     0     0     0     0     0     0
     0     0     0    -1     1     0     0     0     0     0     0     0
     0     0     0     0    -1     1     0     0     0     0     0     0
     0     0     0     0     0    -1     0     0     0     0     0     0
     0     0     0     0     0     0    -1     1     0     0     0     0
     0     0     0     0     0     0     0    -1     1     0     0     0
     0     0     0     0     0     0     0     0    -1     0     0     0
     0     0     0     0     0     0     0     0     0    -1     1     0
     0     0     0     0     0     0     0     0     0     0    -1     1
     0     0     0     0     0     0     0     0     0     0     0    -1


ans =

     0
     0
     1
     0
     0
     1
     0
     0
     1
     0
     0
     1


ans =

    -1     0     0     1     0     0     0     0     0     0     0     0
     0    -1     0     0     1     0     0     0     0     0     0     0
     0     0    -1     0     0     1     0     0     0     0     0     0
     0     0     0    -1     0     0     1     0     0     0     0     0
     0     0     0     0    -1     0     0     1     0     0     0     0
     0     0     0     0     0    -1     0     0     1     0     0     0
     0     0     0     0     0     0    -1     0     0     1     0     0
     0     0     0     0     0     0     0    -1     0     0     1     0
     0     0     0     0     0     0     0     0    -1     0     0     1
     0     0     0     0     0     0     0     0     0    -1     0     0
     0     0     0     0     0     0     0     0     0     0    -1     0
     0     0     0     0     0     0     0     0     0     0     0    -1


ans =

     0
     0
     0
     0
     0
     0
     0
     0
     0
     1
     1
     1
```

To evaluate the derivatives in the first direction, we could again use

```
G_inhom{1}.apply(x)

% or the manual form

reshape(G_inhom{1}.A() * x(:) + G_inhom{1}.b, size(G_inhom{1}))
```

```
ans =

   -0.4233   -0.7597    0.6214   -0.2162
    0.0547    0.4789   -0.2467    0.2588
    0.8322    0.3109    0.4380    0.2331


ans =

   -0.4233   -0.7597    0.6214   -0.2162
    0.0547    0.4789   -0.2467    0.2588
```

```
      0.8322    0.3109    0.4380    0.2331
```

A real-world example

In the following example, we restore a missing part of an image by minimizing the integral of the squared Laplacian over the image:

$$\min_{u:\Omega\to R} \int_\Omega (\Delta u)^2 \quad s.t. \quad u = u_0 \; on \; C.$$

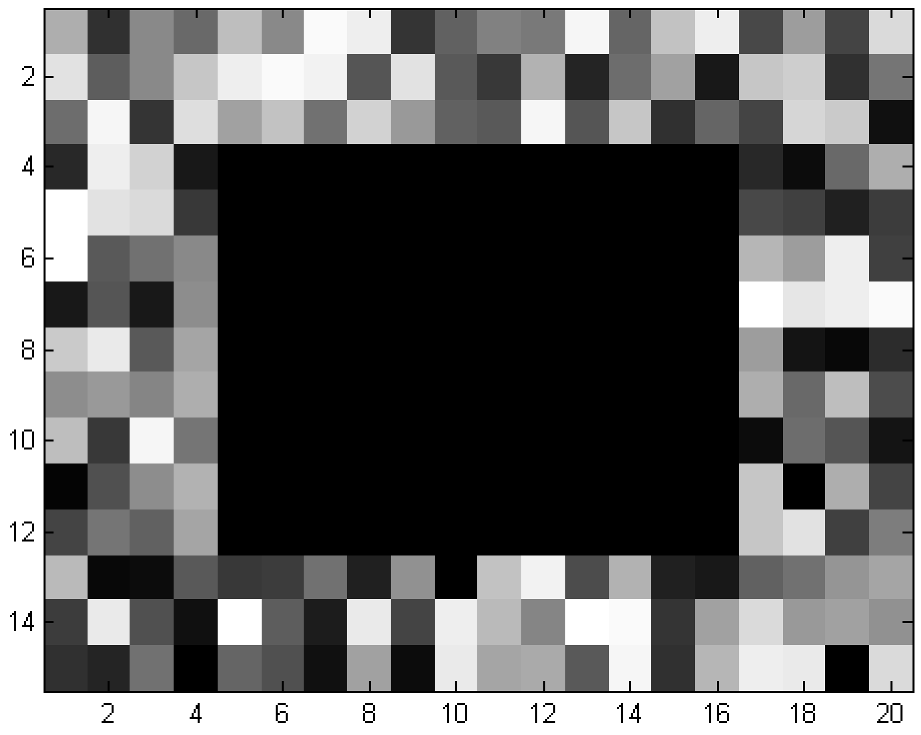
First we implement the two-dimensional Laplace operator:

```
function result = laplace(u)
    ii = 2:size(u,1)-1; jj = 2:size(u,2)-1;
    result = - 4 * u(ii,jj) + u(ii+1,jj) + u(ii-1,jj) + u(ii,jj+1) + u(ii,jj-1);
end
```

Then create a random (small) image, and select a region that should be restored:

```
n = 15; m = 20;
img = rand(n,m);

ni = 4:12; mi = 5:16;
img(ni,mi) = 0; % corrupt center region by setting it to zero
imagesc(img); colormap gray; axis tight;
```



From this data, we create a `DIFFOP` array of constants using the `spconst` function. `spconst` works just like `spvar`, but represents constants instead of variables. `spconst` requires a second parameter that specifies the number of variables that the final operator will have, as there is no way for `spconst` to infer this from the array of constants:

```
nvar = numel(ni) * numel(mi);
uext = spconst(img,nvar);
```

We now create a set of variables for the corrupted region replace the inner region with variables

```
u = spvar([numel(ni) numel(mi)]);
uext(ni,mi) = u;
```

We can now evaluate the Laplace operator on the whole image, which gives us an expression in terms of the variables `u`:

```
L = laplace(uext);
```

The boundary values determined by img appear in the affine part `b()`:

```
b = L.b();
b(1:10)
```

```
        ans =

            1.0960
           -1.9040
           -0.8989
           -0.4049
            1.2697
            0.0997
           -1.6016
           -0.1671
            1.6988
            0.0025
```

We can now find a solution with minimal squared Laplacian: in order to minimize

$$\frac{1}{2}\|Ax+b\|^2,$$

we can simply find a solution of

$$A^\top Ax = -A^\top b$$

with the following code:

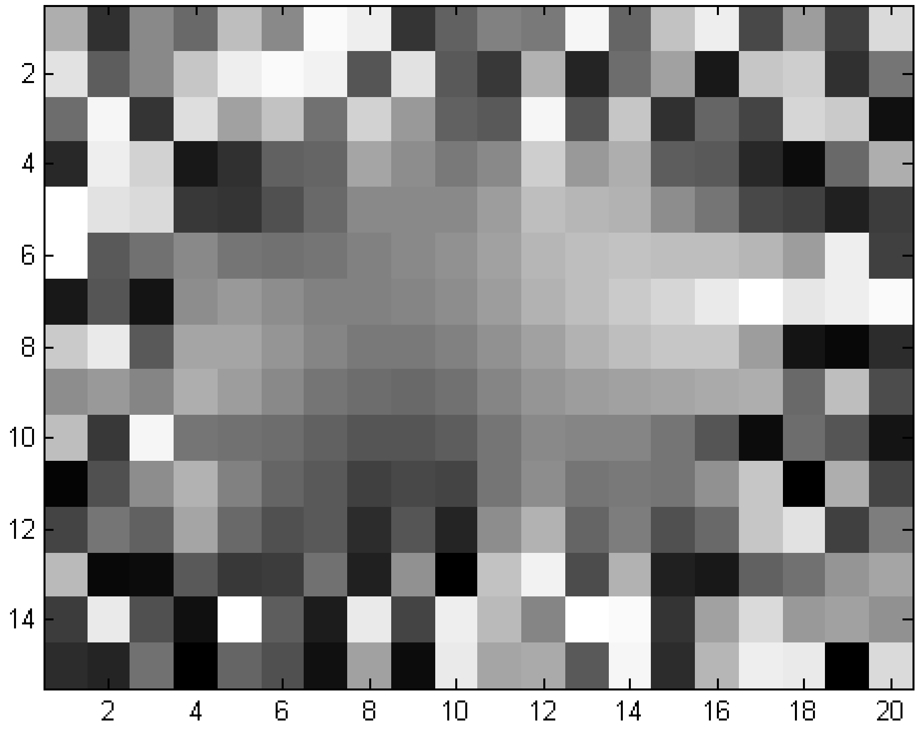
```
u_sol = (L.A()' * L.A()) \ (-L.A()' * L.b());
u_sol = reshape(u_sol, size(u));
```

Finally we piece the solution together with the uncorrupted data:

```
img_sol = img;
```

```
img_sol(ni,mi) = u_sol;

imagesc(img_sol);
```



Large-scale data

`DIFFOP` works quite well even on large matrices with millions of variables, as long as they are sparse enough. The code below uses `DIFFOP` to compute a sparse matrix that computes the Laplace operator matrix for a 1000×2000 image with Dirichlet boundary conditions:

```
n = 1000; m = 2000;
u = spvar([n m]);

u_dir = [
               zeros(1,m+2);...
         zeros(n,1) u zeros(n,1);...
               zeros(1,m+2)...
        ];
```

This should take a few seconds at worst, depending on the `MATLAB` version and system performance.

```
tic; L2 = laplace(u_dir); toc

        Elapsed time is 2.425814 seconds.
```

The resulting matrix is a (very) sparse 2000000×2000000 matrix:

```
A = L2.A();
size(A)
full(A(1:10,1:10))
```

```
        ans =

             2000000     2000000


        ans =

            -4     1     0     0     0     0     0     0     0     0
             1    -4     1     0     0     0     0     0     0     0
             0     1    -4     1     0     0     0     0     0     0
             0     0     1    -4     1     0     0     0     0     0
             0     0     0     1    -4     1     0     0     0     0
             0     0     0     0     1    -4     1     0     0     0
             0     0     0     0     0     1    -4     1     0     0
             0     0     0     0     0     0     1    -4     1     0
             0     0     0     0     0     0     0     1    -4     1
             0     0     0     0     0     0     0     0     1    -4
```

In some cases this might still be too slow, in particular if the operator is implemented inefficiently (for example using loops instead of vectorization). If the matrix does not change between runs, it may be faster to compute the operator once, store it in a `.MAT` file, and just load it on all later runs. If that doesn't help or the matrix changes frequently, unfortunately there is not much hope other than resorting to a manual implementation using `sparse`, `spdiags`, `kron`, etc. `DIFFOP` can still be useful to verify that the manual implementation generates the correct matrix.

Using DIFFOP to speed up operator evaluation

In some cases it can actually be faster to evaluate linear operators using the `DIFFOP` and `apply()` than using the implementation directly. The reason is that sparse matrix operations are one of the most highly optimized parts of `MATLAB`, and once the matrix has been assembled, `MATLAB` does not have to parse/interpret any statements.

The code below evaluates the Laplace operator directly on a random 1000×2000 image:

```
q = rand(n,m);

tic; q1 = reshape(laplace([
                              zeros(1,m+2);...
                        zeros(n,1) q zeros(n,1);...
                              zeros(1,m+2)...
                     ]), [n m]); toc
```

```
        Elapsed time is 0.207650 seconds.
```

The same code using the previously computed operator `L2` and `apply()` is more than twice as fast:

```
tic; q2 = L2.apply(q); toc
```

```
        Elapsed time is 0.077442 seconds.
```

The results are in fact the same up to rounding error:

```
norm(q1(:) - q2(:), +inf)
```

```
ans =

   1.3323e-15
```

```
end

%
% DIFFOP - Demonstration Project
%
% Copyright (c) 2014 Jan Lellmann (J.Lellmann@damtp.cam.ac.uk) / CIA
%
% This file is part of the DIFFOP library. DIFFOP may be used for
% non-commercial purposes, and may be freely modified and distributed
% as long as this copyright notice is retained. Use of DIFFOP in a
% commercial environment requires explicit written permission.
%
```

Published with MATLAB® R2013a