1.723 - COMPUTATIONAL METHODS FOR FLOW IN POROUS MEDIA
Spring 2009

FINITE DIFFERENCE METHODS (II): 1D EXAMPLES IN MATLAB

Luis Cueto-Felgueroso

1. COMPUTING FINITE DIFFERENCE WEIGHTS

The function `fdcoefs` computes the finite difference weights using Fornberg's algorithm (based on polynomial interpolation). The syntax is

```
>> [coefs]= fdcoefs(m,n,x,xi);
```

Given a set of $n+1$ nodes, with coordinates $x = [x_0, \ldots, x_n]$ (remember the basic idea of the point-wise finite difference discretization, figure 1), `fdcoefs(m,n,x,xi)` computes the FD weights associated to each nodal point for the approximation of the m-th derivative at point xi (xi may or may not be a grid point).

1.1. *A first example*

We may use `fdcoefs` to derive general finite difference formulas. Let's compute, for example, the weights of the 5-point, centered formula for the first derivative

$$u'_j = \frac{-u_{j+2} + 8u_{j+1} - 8u_{j-1} + u_{j-2}}{12\Delta x} + O(\Delta x^4) \tag{1}$$

Here we are interested in the first derivative ($m = 1$) at point x_j. The stencil is, in principle, $\{x_{j-2}, x_{j-1}, x_j, x_{j+1}, x_{j+2}\}$. In order to use `fdcoefs`, we could either generate a vector $\boldsymbol{x}$ with the coordinates of the particular points we are considering or, specially for uniformly spaced nodes, compute the general weights assuming $\Delta x = 1$. In the latter approach, the Δx will enter separately in the FD formula, as in the denominator of (1). Thus, the function `fdcoefs` will provide the coefficients $\{\alpha_{j+2}, \alpha_{j+1}, \alpha_j, \alpha_{j-1}, \alpha_{j-2}\}$ such that

$$u'_j = \frac{\alpha_{j+2}u_{j+2} + \alpha_{j+2}u_{j+1} + \alpha_j u_j + \alpha_{j-1}u_{j-1} + \alpha_{j-2}u_{j-2}}{\Delta x} + O(\Delta x^4) \tag{2}$$

How is this done? After setting

```
>> m= 1;             % First order derivative
>> n= 4;             % The stencil comprises n+1 points...
>> x= [2 1 0 -1 -2]; % Generic coordinates of the points (dx= 1)
>> xi= 0;            % Formula for the central point
```

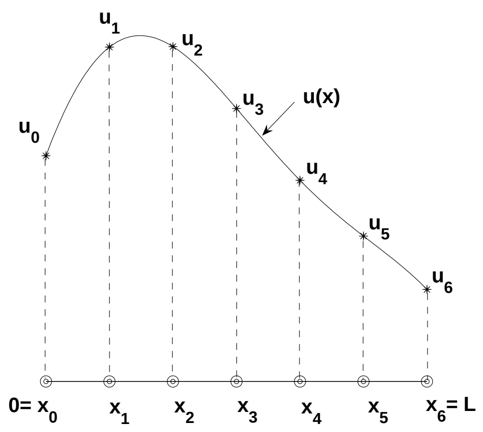

Figure 1. Point-wise discretization used by finite differences.

we run `fdcoefs`, to obtain

```
>> coefs= fdcoefs(m,n,x,xi)'

coefs =

  -0.08333333333333
   0.66666666666667
                  0
  -0.66666666666667
   0.08333333333333
```

The output is therefore the vector `coefs= [-1 8 0 -8 1]/12=` $[\alpha_{j+2}\,\alpha_{j+1}\,\alpha_j, \alpha_{j-1}, \alpha_{j-2}]$, whose components are precisely the coefficients in (1). Analogously, we could obtain the coefficients to compute the second derivative using a 7-point centered formula, through the sequence

```
>> m= 2;                  % Second order derivative
>> n= 6;                  % The stencil comprises n+1 points...
>> x= [3 2 1 0 -1 -2 -3]; % Generic coordinates of the points (dx= 1)
>> xi= 0;                 % Formula for the central point
>> coefs= fdcoefs(m,n,x,xi);
```

and now `coefs= [2 -27 270 -490 270 -27 2]/180`.

Note that, for uniformly spaced nodes, we may write a general approximation for the m-th derivative at point i, using a stencil of $n+1$ neighbor nodes, as

$$u_i^{(m)} = \frac{1}{\Delta x^m}\sum_{j=0}^{n}\alpha_j u_j \tag{3}$$

where the coefficients $\{\alpha_j\}$ are computed on a standard grid with $\Delta x = 1$, as we did before.

We may also use `fdcoefs` to obtain one-sided formulas, use irregularly spaced points, etc. Let us compute the coefficients of an approximation for the first derivative at point x_1, using the stencil $\{x_0, x_1, x_2, x_3, x_4\}$, as

$$u_1' = \frac{\alpha_0 u_0 + \alpha_1 u_1 + \alpha_2 u_2 + \alpha_3 u_3 + \alpha_4 u_4}{\Delta x} + O(\Delta x^4) \tag{4}$$

The use of `fdcoefs` in this case reads

```
>> m= 1;                  % First order derivative
>> n= 4;                  % The stencil comprises n+1 points...
>> x= [0 1 2 3 4];        % Generic coordinates of the points (dx= 1)
>> xi= 1;                 % Formula for the second point
>> coefs= fdcoefs(m,n,x,xi);
```

which produces `coefs= [-3 -10 18 -6 1]/12=` $[\alpha_0\,\alpha_1\,\alpha_2\,\alpha_3\,\alpha_4]$

The grid points need not be uniformly spaced, nor the evaluation point has to be one of the grid points. Assume for example that we are given the values of a function $u(x)$ at 5 points with coordinates $\boldsymbol{x} = [x_0\,x_1\,x_2\,x_3\,x_4] =$ `[-0.12 0.03 0.205 0.34 0.5]`. We could approximate the second derivative of $u(x)$ at $x_i =$ `0.103`, for example, with an expression of the form

$$u_i'' = \sum_{j=0}^{4} \alpha_j u_j \tag{5}$$

and determine the weights $\{\alpha_j\}$ using `fdcoefs`, as

```
>> m= 2;
>> n= 4;
>> x= [-0.12 0.03 0.205 0.34 0.5];
>> xi= 0.103;
>> alpha= fdcoefs(m,n,x,xi)'

alpha =

   9.55277974400200
  14.32199405938392
 -78.42383543004449
  63.08923101917553
  -8.54016939251694
```

2. CREATING DIFFERENTIATION MATRICES. CONVERGENCE

Consider now a full grid of $N+1$ points, $\{x_0, x_1, \ldots, x_N\}$. Once we have determined the weights that we will use to approximate the m-th derivatives at each grid point, we can evaluate them as

$$u_i^{(m)} = \sum_{j=0}^{N} d_{ij}^{(m)} u_j \tag{6}$$

Note that, although the sum extends in principle over all grid points, in practice only a few coefficients are different from zero (for each node i, those $n+1$ nodes that belong to its *stencil*). The right hand side of equation (6) is just a matrix-vector product, which can be written as

$$\boldsymbol{u}_i^{(m)} = \boldsymbol{D}^{(m)} \boldsymbol{u} \tag{7}$$

where $\boldsymbol{D}^{(m)}$ is the *differentiation matrix*. For general, irregular grids, this matrix can be constructed by generating the FD weights for each grid point i (using `fdcoefs`, for example), and then introducing these weights in row i. Of course `fdcoefs` only computes the non-zero weights, so the other components of the row have to be set to zero.

For a grid of $N+1$ uniformly spaced points, the function `diffmatrix` computes the differentiation matrices for the first and second derivatives. It generates formulas with centered stencils for interior nodes, and one-sided approximations near the boundaries. You can specify the size of the stencils. It works with formulas of arbitrary number of points n, but it assumes that n is odd (i.e. it will only work for $n = 3, 5, 7, 9, 11, ...$). The syntax is

```
>> [D1,D2]= diffmatrix(x,n);
```

where `x` is a vector containing the coordinates of the grid points (now this means the coordinates of the $N+1$ points of the grid, not just those of the nodes in a particular stencil like in `fdcoefs`), and n is the stencil of the formulas ($n = 5$ for 5-point formulas...). Consider, for example, the function $u(x) = x^2$, defined on the interval $I = [0, 3]$, and discretized into a grid of 4 evenly spaced points, with coordinates $\boldsymbol{x} = (0\,1\,2\,3)$. In order to compute the first and second order derivatives at the grid points, we may use `diffmatrix` to construct the 3-point differentiation matrices as

```
>> n= 3;
>> x= 0:3;
>> [D1,D2]= diffmatrix(x,n)

D1 =

    -1.5   2    -0.5  0
    -0.5   0     0.5  0
     0    -0.5   0    0.5
     0     0.5  -2    1.5


D2 =

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

We can thus approximate the derivatives at the grid points as

$$\boldsymbol{u}_i' = \boldsymbol{D}^{(1)}\boldsymbol{u} = \begin{pmatrix} -1.5 & 2 & -0.5 & 0 \\ -0.5 & 0 & 0.5 & 0 \\ 0 & -0.5 & 0 & 0.5 \\ 0 & 0.5 & -2 & 1.5 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 4 \\ 9 \end{pmatrix} = \begin{pmatrix} 0 \\ 2 \\ 4 \\ 6 \end{pmatrix} \tag{8}$$

and

$$\boldsymbol{u}_i'' = \boldsymbol{D}^{(2)}\boldsymbol{u} = \begin{pmatrix} 1 & -2 & 1 & 0 \\ 1 & -2 & 1 & 0 \\ 0 & 1 & -2 & 1 \\ 0 & 1 & -2 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 4 \\ 9 \end{pmatrix} = \begin{pmatrix} 2 \\ 2 \\ 2 \\ 2 \end{pmatrix} \tag{9}$$

In this simple case, as $u(x)$ is just a quadratic function, the computed derivatives are exact at the nodes.

2.1. *Another example of numerical differentiation*

Consider a more general differentiation example. We will now be using the function `diff_test.m`, which also calls `diffmatrix.m`. Consider the function

$$u(x) = \frac{3}{5 - 4cos^2(2x)} \qquad x \in [0, 2\pi] \tag{10}$$

This function is periodic in $[0, 2\pi]$, but we will not make use of this property in this example. The function `diff_test` computes the first and second order derivatives of $u(x)$ using finite differences on a grid of $N + 1$ points. The syntax is

```
>> diff_test(n,N);
```

For example,

```
>> diff_test(7,134);
```

would produce the plots shown in figure (2). Let's take a look a the code... it starts by creating "inline" functions that will be used later to evaluate the function at the grid points and to compute the exact derivatives for the error analysis

```
syms x
u= 3*(5-4*cos(2*x)ˆ2)ˆ-1;
du = diff(u,1);
ddu= diff(u,2);
Fu  = inline(vectorize(simplify(u)));
Fdu = inline(vectorize(simplify(du)));
Fddu= inline(vectorize(simplify(ddu)));
```

then we generate the grid and differentiation matrices

```
h= 2*pi/N;
x= (0:h:2*pi)';
[D1,D2]= diffmatrix(x,n);
```

We can now compute the derivatives of u... but first we need its values at the nodes...

```
u= Fu(x);
```

and then we are just a matrix-vector product away...

```
duFD = D1*u;
dduFD= D2*u;
```

The remainder of the function is just computing errors and plotting...

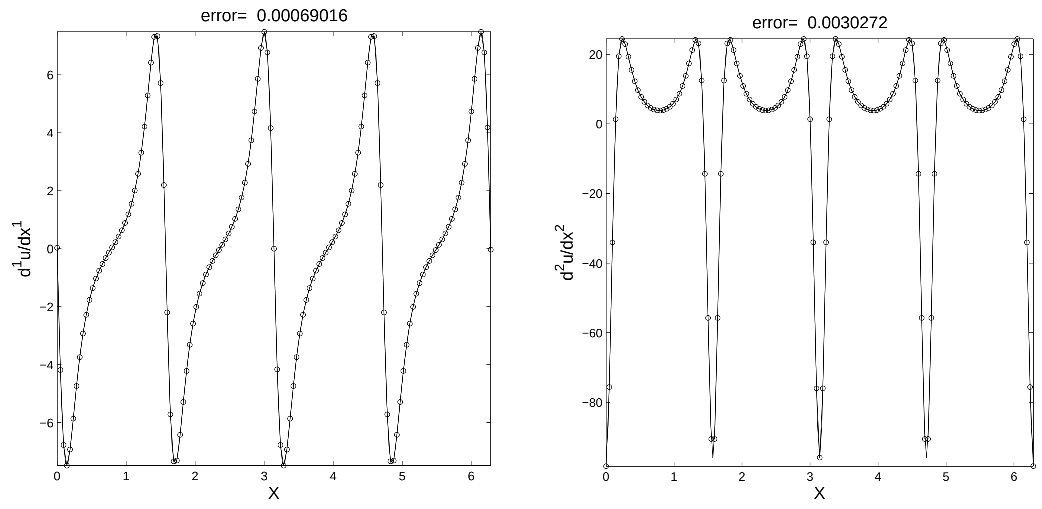

Figure 2. What pops up when we run *diff_test(7,134)*

2.2. *Periodic functions*

The function `diff_test_per.m` is similar to `diff_test.m`, but it exploits the fact that $u(x)$ is periodic. It also goes a step further in that it "formally" allows to compute derivatives of arbitrarily high order m. I say "formally" because in practice the propagation of roundoff errors would end up ruining our finite difference computations for sufficiently large m's. Differentiation is numerically unstable...

The syntax for `diff_test_per` is

```
>> diff_test_per(m,n,N);
```

which means that we want to compute the m-th derivative of $u(x)$ (the same function as in the previous case (10)), with an n-point formula, on a periodic grid of N points in $[0, 2\pi]$. It also assumes that n is odd.

For example,

```
>> diff_test_per(3,9,147);
```

would produce the plot shown in figure (3).

The main difference with respect to `diff_test` is that `diff_test_per` computes the weights using `fdcoefs` for a generic point of grid, and then just constructs the differentiation matrix as a Toeplitz one.

Thus, the coefficients of the formula are first computed with

```
%Coefficients of the difference scheme
x= -(n-1)/2:(n-1)/2;
[FDcoefs]= fdcoefs(m,n-1,x,0);
```

after which the grid and differentiation matrices are

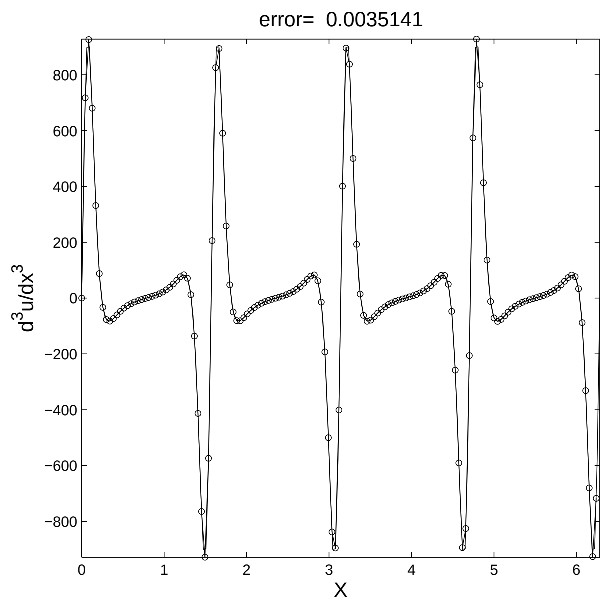

Figure 3. What pops up when we run *diff_test_per(3,9,147)*

```
%Grid and differentiation matrix
h= 2*pi/N;
x= 0:h:2*pi;x= x(1:N)';
FDcoefs= FDcoefs/(h^m);
R= [ FDcoefs((n-1)/2+1:n)   zeros(1,N-n)   FDcoefs(1:(n-1)/2) ];
C= [fliplr(FDcoefs(1:(n-1)/2+1))  zeros(1,N-n)  fliplr(FDcoefs((n-1)/2+2:n))];
D= toeplitz(sparse(C),sparse(R));
```

Note that we had to divide by Δx^m (and that I called $h = \Delta x$). Also note that, when we generate the grid, we *don't* include the node at the right boundary (periodicity implies that the point at the left boundary would be there again).

Once we have the differentiation matrix, computing the derivative is just

```
%Finite difference derivative
u= Fu(x);
duFD= D*u;
```

And then plotting...

2.3. Convergence check

In order to check the convergence of our finite difference approximations, we may generate increasingly refined grids, and track the evolution of the error as a function of the grid size. As we saw in the theoretical part, the plot is particularly insightful in logarithmic scale.

Take a look at the function `diff_conv`. It uses the same code as in `diff_test_per` to compute derivatives, but now it does it for several values of N, and then plots the convergence history. The syntax is

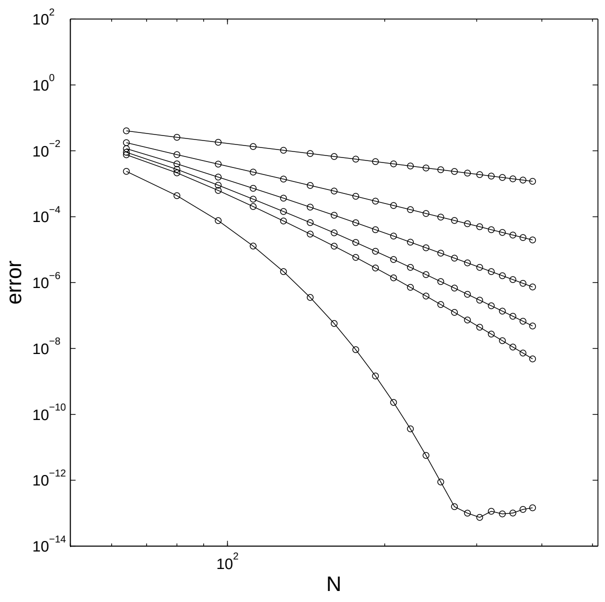

Figure 4. The convergence test.

```
>> diff_conv(m,n,N);
```

meaning that we are analyzing the convergence of an n-point formula for the m-th derivative. In this case N is a *vector* containing the number of grid points on each refinement level. We use again the same function $u(x)$ of the previous examples (10).

For this convergence test, in addition to difference formulas, the function includes the limit case of using all the points to compute the derivative ($n = N$). This may be specified by setting `n= NaN`, and in this case the derivatives are computed in Fourier space (we will talk about this later in the course). For example, the sequence

```
>> diff_conv(2,3,64:16:384);
>> diff_conv(2,5,64:16:384);
>> diff_conv(2,7,64:16:384);
>> diff_conv(2,9,64:16:384);
>> diff_conv(2,11,64:16:384);
>> diff_conv(2,NaN,64:16:384);
>> axis([50,512,10ˆ-14,10ˆ2]);
```

produces the plot shown in figure (4), which represents a comparison of the performance of several difference formulas for the second derivative ($n = 3 - 11$) with a Fourier spectral method (`n= NaN`). You may also plot the reference straight lines, which represent the formal order of the truncation error. This feature is by default commented in the code.

3. BOUNDARY-VALUE PROBLEMS

Consider the 1D Poisson problem

$$\frac{d^2u}{dx} = S(x) \qquad x \in [0, 2\pi] \tag{11}$$

with source term

$$S(x) = -\frac{9}{4}\cos\left(\frac{3}{2}x\right) \tag{12}$$

and boundary conditions

$$u(0) = 0 \qquad \left.\frac{du}{dx}\right|_{x=2\pi} = 0 \tag{13}$$

The exact solution of (11)–(13) is

$$u(x) = \cos\left(\frac{3}{2}x\right) - 1 \tag{14}$$

Consider a discretization of $[0, 2\pi]$ into a set of $N + 1$ grid points, $\{x_0, x_1, \ldots, x_N\}$. Once we have computed the differentiation matrix for the second derivative, $\boldsymbol{D}^{(2)}$, the discrete version of (11) is simply

$$\boldsymbol{D}^{(2)}\boldsymbol{u} = \boldsymbol{S} \tag{15}$$

where

$$\boldsymbol{u} = \begin{pmatrix} u_0 \\ u_1 \\ \vdots \\ u_N \end{pmatrix} \qquad \boldsymbol{S} = \begin{pmatrix} S(x_0) \\ S(x_1) \\ \vdots \\ S(x_N) \end{pmatrix} \tag{16}$$

and we still have to include the boundary conditions $u_0 = 0$ and $u'_N = 0$. The first one is trivial, while for the second one we may generate a finite difference approximation for the first derivative at x_N, and the impose that it vanishes. It will be more clear by taking a look at the function `poisson_test`, which solves this boundary value problem. The syntax is

```
>> poisson_test(n,N);
```

where n is again the number of points in the difference formula, and $N + 1$ the number of grid points. For example, the command

```
>> poisson_test(11,32);
```

produces the plot shown in figure 5

Looking into the code, the generation of the grid and differentiation matrices is carried out by

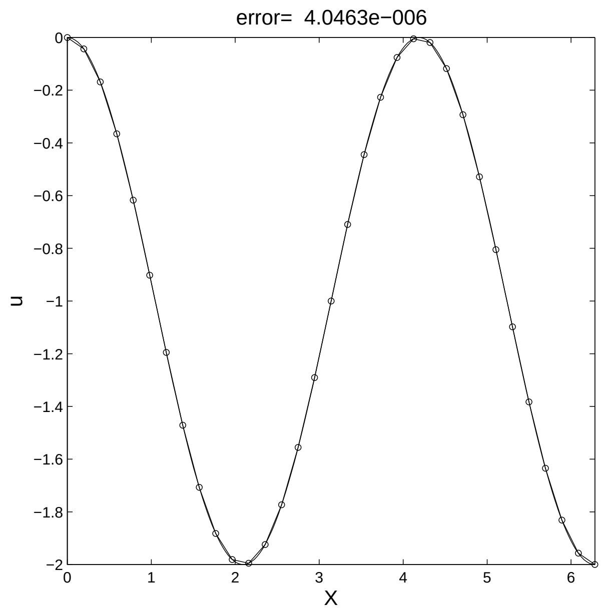

Figure 5. What pops up when we run *poisson_test(11,32)*

```
h= 2*pi/N;
x= (0:h:2*pi)';
[D1,D2]= diffmatrix(x,n);
L= D2;
```

where the discrete “Laplacian” is simply $\boldsymbol{L} = \boldsymbol{D}^{(2)}$. We also need the source term

```
%Source term
S= FS(x);
```

Once we have the differentiation matrices and source term, we just need to impose the boundary conditions

```
%Dirichlet
L(1,:)= 0;L(1,1)= 1;
S(1)= 0;
%Neumann
L(end,:)= D1(end,:);
S(end)= 0;
```

and then solve the linear system of equations (15)

```
%Finite difference solution
uFD= L\S;
```

and, of course, errors, plots and other herbs.