fed: Fast Explicit Diffusion for C/C++

Sven Grewenig[1] Joachim Weickert[1] Andrés Bruhn[2]
Pascal Gwosdek[1] Oliver Demetz[1]

[1] Mathematical Image Analysis Group, Faculty of Mathematics and Computer Science,
Saarland University, Campus E 1.1, 66123 Saarbrücken, Germany
{grewenig,weickert,gwosdek,demetz}@mia.uni-saarland.de

[2] Vision and Image Processing Group,
Cluster of Excellence Multimodal Computing and Interaction,
Saarland University, Campus E 1.1, 66123 Saarbrücken, Germany
bruhn@mmci.uni-saarland.de

October 14, 2010

1 Introduction

This document provides a short reference to our sample implementation of the Fast Explicit Diffusion (FED) scheme presented in [GWB10]. To this end, our library allows to compute time steps in a stable ordering, which can then be used within a standard implementation of an explicit scheme for diffusion processes.

A simple example of how to use this library is provided in the next section. As a good way to get started with FED, we suggest to take an explicit scheme for an arbitrary diffusion problem, and to extend it by using time steps generated by this library. If you are instead interested in a complete function reference, you will find this in Section 3.

We would like to remind you that this piece of code is distributed in the hope that it will be useful, but without any warranty. Please understand that we can not help you with the implementation of a particular project, nor will we take responsibility for correctness or functionality of our code.

2 Example

Let us now show the usage of our library with a simple example. As a test setting, we choose linear diffusion, i.e. the solution of the 1-D heat equation

$$\partial_t u = \partial_{xx} u \,.$$

To this end, we create a discrete random 1-D signal of length 100, and apply 5 FED cycles with an overall stopping time of the process of 500. For a standard discretisation with central finite differences, the stability limit is 0.5, assuming a unit spatial grid size.

Since we know the stopping time of the process and the number of FED cycles *a priori*, the best function for this purpose is `fed_tau_by_process_time`. On input of the number of outer cycles `M` and the stopping time `T` of the process, it outputs the number `n` of inner steps, and allocates and fills the array `tau` with the respective step sizes. In addition, this function requires knowledge of the stability limit `tau_max` of the process. Instead of using a constant step size of 0.5 in a standard explicit solver, we substitute it by the entries of the array `tau`.

```c
#include <limits.h>
#include "fed.h"

int main()
{
  float u  [102]; /* Signal ([1..100] with 1-px bounds)            */
  float tmp[102]; /* Temporary helper array                        */
  float *tau;     /* Vector of FED time step sizes                 */
  int   N;        /* Number of steps                               */
  int   M;        /* Number of cycles                              */
  int   n, m, i;  /* Loop counters over steps, cycles, pixels      */

  /* Inititalise signal with uniform random numbers from [0,1]     */
  for(i = 1; i <= 100; i++)
  {
    u[i] = (float)rand()/(float)RAND_MAX;
    printf("%f\n", u[i]);
  }

  /* Initialise step sizes for process with                        */
  /* - overall stopping time T:           500                      */
  /* - number of cycles M:                5                        */
  /* - stability limit for 1-D diffusion: 0.5                      */
  M = 5;
  N = fed_tau_by_process_time(500.0f, M, 0.5f, 1, &tau);

  /* Perform M outer cycles                                        */
  for(m = 0; m < M; m++)
  {
    /* Each cycle performs N steps with varying step size          */
    for(n = 0; n < N; n++)
    {
      /* Reflecting boundary conditions                            */
      u[0]   = u[1];
      u[101] = u[100];

      /* One explict step                                          */
      for(i = 1; i <= 100; i++)
        tmp[i] = u[i] + tau[n] * (u[i-1] - 2.0f * u[i] + u[i+1]);

      /* Copy back for next iteration                              */
      for(i = 1; i <= 100; i++)
        u[i] = tmp[i];
    }
  }

  /* Output filtered array                                         */
  printf("\n\nFiltered:\n\n");
  for(i = 1; i <= 100; i++)
    printf("%f\n", u[i]);

  /* Free time step vector                                         */
  free(tau);

  return 0;
}
```

You can also find this example in the file `example.c` in the `fed` package. It can be compiled by

```
$> gcc -lm -o example example.c fed.c
```

We would like to remind the reader that this particular piece of code is not optimised for speed. In fact, a five-fold application of a box filter will yield equivalent results in a shorter time. However, this code still perfectly demonstrates how to use FED for arbitrary diffusion-like processes. While the actual implementation of the explicit scheme will change if we consider more complex or higher-dimensional problems, the call to the FED routine will remain simple.

3 Function Reference

The core of FED is given by three functions that take different arguments, but all output an ordered array of time steps. Please use

- `fed_tau_by_steps`
  if you want to obtain the maximal stopping time that can be achieved with a certain number of steps,
- `fed_tau_by_cycle_time`
  if you aim for a certain stopping time for one cycle, and
- `fed_tau_by_process_time`
  if you aim for a certain stopping time for the entire process.

If you should consider the first function, you might also be interested in either of our functions `fed_max_cycle_time_by_steps` or `fed_max_process_time_by_steps` which compute the corresponding stopping time to the desired number of steps.

3.1 `int` **`fed_tau_by_steps`**

`(int n, float tau_max, int reordering, float **tau)`

This function assumes the number of cycles and steps are both given, and generates a series of time steps that yields the largest stopping time possible under these conditions.

Input Parameters

- **`int n`**
  The desired number of steps per cycle.
- **`float tau_max`**
  The stability limit for a standard explicit scheme designed for this purpose. For diffusion, $0.5^{\text{dimensions}}$ is usually a good choice (i.e. `0.5` for 1-D, `0.25` for 2-D, etc.), assuming unit spatial grid size.
- **`int reordering`**
  Flag indicating if time steps should be reordered using kappa cycles (`1`), or if they should be output in native order (`0`). `1` is highly recommended.

Output Parameters

- **`float **tau`**
  Address of uninitialised `float` pointer. The function will allocate and fill a sufficiently large array of time steps that can later be used to set up an FED cycle. `tau` must be freed after usage.

Returns (int)

n on success, or 0 else.

3.2 int fed_tau_by_cycle_time

(float t, float tau_max, int reordering, float **tau)

This function assumes the number of cycles and and the stopping time per cycle are given, and generates a series of time steps that yields this stopping time.

Input Parameters

- float **t**
  The desired stopping time per cycle.
- float **tau_max**
  The stability limit for a standard explicit scheme designed for this purpose. For diffusion, $0.5^{\text{dimensions}}$ is usually a good choice (i.e. 0.5 for 1-D, 0.25 for 2-D, etc.), assuming unit spatial grid size.
- int **reordering**
  Flag indicating if time steps should be reordered using kappa cycles (1), or if they should be output in native order (0). 1 is highly recommended.

Output Parameters

- float ****tau**
  Address of uninitialised float pointer. The function will allocate and fill a sufficiently large array of time steps that can later be used to set up an FED cycle. tau must be freed after usage.

Returns (int)

The number of steps per cycle n on success, or 0 else.

3.3 int fed_tau_by_process_time

(float T, int M, float tau_max, int reordering, float **tau)

This function assumes the number of cycles and the stopping time of the whole process consisting of several cycles are given. It generates a series of time steps that allow to construct a cycle, so that the concatenation of all such cycles yields the desired stopping time for the process.

Input Parameters

- float **T**
  The desired stopping time of the whole process.
- int **M**
  The number of cycles in the process.
- float **tau_max**
  The stability limit for a standard explicit scheme designed for this purpose. For diffusion, $0.5^{\text{dimensions}}$ is usually a good choice (i.e. 0.5 for 1-D, 0.25 for 2-D, etc.), assuming unit spatial grid size.

- `int` **reordering**
  Flag indicating if time steps should be reordered using kappa cycles (`1`), or if they should be output in native order (`0`). `1` is highly recommended.

Output Parameters

- `float **` **tau**
  Address of uninitialised `float` pointer. The function will allocate and fill a sufficiently large array of time steps that can later be used to set up an FED cycle. `tau` must be freed after usage.

Returns `(int)`

The number of steps per cycle `n` on success, or `0` else.

3.4 `float` fed_max_cycle_time_by_steps `(int n, float tau_max)`

Given a number of time steps per cycle and the stability criterion, this function outputs the maximal FED stopping time per cycle. Thus, this function is not required to generate a series of FED time steps, but serves informational purposes if the routine `fed_tau_by_steps` is used.

Input Parameters

- `int` **n**
  The number of steps per cycle.

- `float` **tau_max**
  The stability limit for a standard explicit scheme designed for this purpose. For diffusion, $0.5^{\text{dimensions}}$ is usually a good choice (i.e. `0.5` for 1-D, `0.25` for 2-D, etc.), assuming unit spatial grid size.

Returns `(float)`

The maximal stopping time `t` per cycle, as implied by `fed_tau_by_steps`.

3.5 `float` fed_max_process_time_by_steps `(int n, int M, float tau_max)`

As above, but the overall stopping time for the process is returned.

Input Parameters

- `int` **n**
  The number of steps per cycle.

- `int` **M**
  The number of cycles.

- `float` **tau_max**
  The stability limit for a standard explicit scheme designed for this purpose. For diffusion, $0.5^{\text{dimensions}}$ is usually a good choice (i.e. `0.5` for 1-D, `0.25` for 2-D, etc.), assuming unit spatial grid size.

Returns (`float`)

The maximal stopping time `T` of the process, as implied by `fed_tau_by_steps`.

4 License

If you intend to use this library for an own publication, please cite [GWB10] as a reference for FED.

References

[GWB10] S. Grewenig, J. Weickert, and A. Bruhn. From box filtering to fast explicit diffusion. In M. Goesele, S. Roth, Arjan Kuijper, Bernt Schiele, and Konrad Schindler, editors, *Pattern Recognition*, volume 6376 of *Lecture Notes in Computer Science*, pages 533–542. Springer, Berlin, 2010.