```
/****
*
* GPU accelerated pattern formation analysis
*
* Copyright (C) 2009 Tobias Preis (http://www.tobiaspreis.de)
*
* This program is free software; you can redistribute it and/or
* modify it under the terms of the GNU General Public License
* as published by the Free Software Foundation; either version
* 3 of the License, or (at your option) any later version.
*
* This program is distributed in the hope that it will be useful,
* but WITHOUT ANY WARRANTY; without even the implied warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
* GNU General Public License for more details.
*
* You should have received a copy of the GNU General Public
* License along with this program; if not, see
* http://www.gnu.org/licenses/.
*
* Related publication:
*
* T. Preis, P. Virnau, W. Paul, and J. J. Schneider,
* Accelerated fluctuation analysis by graphic cards and complex
* pattern formation in financial markets,
* New Journal of Physics 11, 093024 (2009)
*
*/

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include <cutil.h>

#define FLAG_INPUT_DATA 0
#define FLAG_INPUT_TRIVIAL_LINE 1
#define FLAG_CPU_CALC 1
#define FLAG_GPU_CALC 1
#define FLAG_ERROR_DETAILS 0
#define FLAG_PRINT_CPU_RESULTS 1
#define FLAG_PRINT_GPU_RESULTS 1
#define FLAG_VERSION_CUT_OFF 0
#define FLAG_SYSTEM 0
#define FLAG_RANDOM_DATA 1

#define BLOCK_SIZE 512
#define GAMMA 0.5

const int T=20000;
const float CHI=100;
const float OMEGA=0.01;
const int SCAN_INTERVAL=30*BLOCK_SIZE;
const int DELTA_T_MINUS_MAX=10;
const int DELTA_T_PLUS_MAX=DELTA_T_MINUS_MAX;

/****
 *
 *  Function declaration
 *
 */
void calc(int argc,char** argv);
void init_array(float*,int);
```

```
void load_array(float*,int);
void cpu_function(float*,double*,double*);
void print_result(float*);
void print_result(double*);
__global__ void device_function(float*,float*,float*,int,int,int);
__global__ void device_function_postprocessing(float*,int);
__global__ void
device_function_finalprocessing(float*,float*,float*,float*,int);
__global__ void device_function_init(float*);

/****
 *
 *  Main function
 *
 */
int main(int argc,char** argv) {
  calc(argc,argv);
}

/****
 *
 *  Calc
 *
 */
void calc(int argc,char** argv) {

  printf("
--------------------------------------------------------------------------------
---------\n");
  printf(" *\n");
  printf(" * GPU accelerated pattern formation analysis\n");
  printf(" *\n");
  printf(" * Copyright (C) 2009 Tobias Preis (http://www.tobiaspreis.de)\n");
  printf(" *\n");
  printf(" * This program is free software; you can redistribute it and/or\n");
  printf(" * modify it under the terms of the GNU General Public License\n");
  printf(" * as published by the Free Software Foundation; either version\n");
  printf(" * 3 of the License, or (at your option) any later version.\n");
  printf(" *\n");
  printf(" * This program is distributed in the hope that it will be use-
ful,\n");
  printf(" * but WITHOUT ANY WARRANTY; without even the implied warranty of\n");
  printf(" * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the\n");
  printf(" * GNU General Public License for more details.\n");
  printf(" *\n");
  printf(" * You should have received a copy of the GNU General Public\n");
  printf(" * License along with this program; if not, see\n");
  printf(" * http://www.gnu.org/licenses/\n");
  printf(" *\n");
  printf(" * Related publication:\n");
  printf(" *\n");
  printf(" * T. Preis, P. Virnau, W. Paul, and J. J. Schneider,\n");
  printf(" * Accelerated fluctuation analysis by graphic cards and complex\n");
  printf(" * pattern formation in financial markets,\n");
  printf(" * New Journal of Physics 11, 093024 (2009)\n");
  printf(" *\n");

  if(!FLAG_SYSTEM) {
    printf("\n ----------------------------------- Pattern Conformity
----------------------------------\n");
  }

  //Init device
  CUT_DEVICE_INIT(argc,argv);
```

```
  unsigned int timer=0;

  //Init rand()
  srand(23);

  //Allocate and init host memory
  float* p=(float*) malloc(sizeof(float)*T);
  if(FLAG_RANDOM_DATA) init_array(p,T);
  else load_array(p,T);
  double* pc=(double*)
malloc(sizeof(double)*DELTA_T_MINUS_MAX*DELTA_T_PLUS_MAX);
  double* pc_c=(double*)
malloc(sizeof(double)*DELTA_T_MINUS_MAX*DELTA_T_PLUS_MAX);
  for(int i=0;i<DELTA_T_MINUS_MAX*DELTA_T_PLUS_MAX;++i) {
      pc[i]=0;
      pc_c[i]=0;
  }

  //Allocate memory for device transfers of results
  float* d_host_pc=(float*)
malloc(sizeof(float)*DELTA_T_MINUS_MAX*DELTA_T_PLUS_MAX);
  float* d_host_pc_c=(float*)
malloc(sizeof(float)*DELTA_T_MINUS_MAX*DELTA_T_PLUS_MAX);
  for(int i=0;i<DELTA_T_MINUS_MAX*DELTA_T_PLUS_MAX;++i) {
      d_host_pc[i]=0;
      d_host_pc_c[i]=0;
  }

  //Create and start timer
  float gpu_sum=0;
  timer=0;
  CUDA_SAFE_CALL(cudaThreadSynchronize());
  CUT_SAFE_CALL(cutCreateTimer(&timer));
  CUT_SAFE_CALL(cutStartTimer(timer));

  //Allocate device memory for arrays
  float* d_p_data;
  float* d_buffer_data;
  float* d_buffer_data_abs;
  float* d_buffer_data_out;
  float* d_buffer_data_abs_out;
  CUDA_SAFE_CALL(cudaMalloc((void**) &d_p_data,sizeof(float)*T));
  CUDA_SAFE_CALL(cudaMalloc((void**)
&d_buffer_data,sizeof(float)*SCAN_INTERVAL*DELTA_T_PLUS_MAX));
  CUDA_SAFE_CALL(cudaMalloc((void**)
&d_buffer_data_abs,sizeof(float)*SCAN_INTERVAL*DELTA_T_PLUS_MAX));
  CUDA_SAFE_CALL(cudaMalloc((void**)
&d_buffer_data_out,sizeof(float)*DELTA_T_MINUS_MAX*DELTA_T_PLUS_MAX));
  CUDA_SAFE_CALL(cudaMalloc((void**)
&d_buffer_data_abs_out,sizeof(float)*DELTA_T_MINUS_MAX*DELTA_T_PLUS_MAX));

  //Init device memory
  dim3 threads_init(DELTA_T_PLUS_MAX);
  dim3 grid_init(DELTA_T_MINUS_MAX);
  device_function_init<<<grid_init,threads_init>>>(d_buffer_data_out);
  device_function_init<<<grid_init,threads_init>>>(d_buffer_data_abs_out);

  //Stop and destroy timer
  CUDA_SAFE_CALL(cudaThreadSynchronize());
  CUT_SAFE_CALL(cutStopTimer(timer));
  float gpu_dt_malloc=cutGetTimerValue(timer);
  float gpu_dt_mem=0;
  float gpu_dt_main=0;
```

```
  //GPU version
  if(FLAG_GPU_CALC) {
    gpu_sum+=gpu_dt_malloc;
    if(!FLAG_SYSTEM) {
      printf("\n ------------------------------------------ GPU
------------------------------------------ \n");
      printf(" Processing time on GPU for allocating and init: %f (ms)
\n",gpu_dt_malloc);
    }
    CUT_SAFE_CALL(cutDeleteTimer(timer));

    //Create and start timer
    timer=0;
    CUDA_SAFE_CALL(cudaThreadSynchronize());
    CUT_SAFE_CALL(cutCreateTimer(&timer));
    CUT_SAFE_CALL(cutStartTimer(timer));

    //Copy host memory to device

CUDA_SAFE_CALL(cudaMemcpy(d_p_data,p,sizeof(float)*T,cudaMemcpyHostToDevice));

    //Stop and destroy timer
    CUDA_SAFE_CALL(cudaThreadSynchronize());
    CUT_SAFE_CALL(cutStopTimer(timer));
    gpu_dt_mem=cutGetTimerValue(timer);
    gpu_sum+=gpu_dt_mem;
    if(!FLAG_SYSTEM) {
      printf(" Processing time on GPU for memory transfer: %f (ms)
\n",gpu_dt_mem);
    }
    CUT_SAFE_CALL(cutDeleteTimer(timer));

    //Create and start timer
    timer=0;
    CUDA_SAFE_CALL(cudaThreadSynchronize());
    CUT_SAFE_CALL(cutCreateTimer(&timer));
    CUT_SAFE_CALL(cutStartTimer(timer));

    //Setup execution parameters and kernel execution
    int grids=SCAN_INTERVAL/BLOCK_SIZE;
    dim3 threads(BLOCK_SIZE);
    dim3 grid(grids);
    int mem_buffer_out_size=sizeof(float)*DELTA_T_MINUS_MAX*DELTA_T_PLUS_MAX;

    for(int delta_t_minus=1;delta_t_minus<=DELTA_T_MINUS_MAX;delta_t_minus++) {
      for(int
t=2*delta_t_minus+DELTA_T_PLUS_MAX+SCAN_INTERVAL;t<T-DELTA_T_PLUS_MAX;t++) {


device_function<<<grid,threads>>>(d_p_data,d_buffer_data,d_buffer_data_abs,delta
_t_minus,t-DELTA_T_PLUS_MAX-SCAN_INTERVAL-delta_t_minus,t);

     for(int factor=1;factor<SCAN_INTERVAL;factor=factor*2) {
       dim3 grid_reduce(SCAN_INTERVAL/(2*factor));
       dim3 threads_reduce(DELTA_T_PLUS_MAX);

device_function_postprocessing<<<grid_reduce,threads_reduce>>>(d_buffer_data,fac
tor*DELTA_T_PLUS_MAX);

device_function_postprocessing<<<grid_reduce,threads_reduce>>>(d_buffer_data_abs
,factor*DELTA_T_PLUS_MAX);
     }

     //Move partial results in result arrays
```

```
     dim3 threads_final(DELTA_T_PLUS_MAX);
     dim3 grid_final(2);

device_function_finalprocessing<<<grid_final,threads_final>>>(d_buffer_data,d_bu
ffer_data_out,d_buffer_data_abs,d_buffer_data_abs_out,delta_t_minus);
      }
    }

    //Check if kernel execution generated and error
    CUT_CHECK_ERROR("Kernel execution failed");

    //Copy results

CUDA_SAFE_CALL(cudaMemcpy(d_host_pc,d_buffer_data_out,mem_buffer_out_size,cudaMe
mcpyDeviceToHost));

CUDA_SAFE_CALL(cudaMemcpy(d_host_pc_c,d_buffer_data_abs_out,mem_buffer_out_size,
cudaMemcpyDeviceToHost));

    //Normalize
    for(int i=0;i<DELTA_T_MINUS_MAX*DELTA_T_PLUS_MAX;++i) {
      d_host_pc[i]=d_host_pc[i]/d_host_pc_c[i];
    }

    //Stop and destroy timer
    CUDA_SAFE_CALL(cudaThreadSynchronize());
    CUT_SAFE_CALL(cutStopTimer(timer));
    gpu_dt_main=cutGetTimerValue(timer);
    gpu_sum+=gpu_dt_main;
    if(!FLAG_SYSTEM) {
      printf(" Processing time on GPU for main function: %f (ms)
\n",gpu_dt_main);
    }
    CUT_SAFE_CALL(cutDeleteTimer(timer));

    if(FLAG_PRINT_GPU_RESULTS) print_result(d_host_pc);
  }

  //CPU version
  float cpu_sum=0;
  if(FLAG_CPU_CALC) {

    //Create and start timer
    timer=0;
    CUDA_SAFE_CALL(cudaThreadSynchronize());
    CUT_SAFE_CALL(cutCreateTimer(&timer));
    CUT_SAFE_CALL(cutStartTimer(timer));

    cpu_function(p,pc,pc_c);

    //Normalize
    for(int i=0;i<DELTA_T_MINUS_MAX*DELTA_T_PLUS_MAX;++i) {
      pc[i]=pc[i]/pc_c[i];
    }

    //Stop and destroy timer
    CUDA_SAFE_CALL(cudaThreadSynchronize());
    CUT_SAFE_CALL(cutStopTimer(timer));
    cpu_sum=cutGetTimerValue(timer);
    if(!FLAG_SYSTEM) {
      printf("\n ------------------------------------------ CPU
------------------------------------------\n");
      printf(" Total processing time on CPU: %f (ms) \n",cpu_sum);
    }
```

```
    if(FLAG_PRINT_CPU_RESULTS) {
     print_result(pc);
        printf("\n");
    }

    CUT_SAFE_CALL(cutDeleteTimer(timer));
    if(!FLAG_SYSTEM) {
      if(FLAG_GPU_CALC) printf("\n Speedup: %fX \n\n",(cpu_sum/gpu_sum));
    }
  }

  //Cleanup
  free(p);
  free(pc);
  free(pc_c);
  free(d_host_pc);
  free(d_host_pc_c);
  CUDA_SAFE_CALL(cudaFree(d_p_data));
  CUDA_SAFE_CALL(cudaFree(d_buffer_data));
  CUDA_SAFE_CALL(cudaFree(d_buffer_data_abs));
  CUDA_SAFE_CALL(cudaFree(d_buffer_data_out));
  CUDA_SAFE_CALL(cudaFree(d_buffer_data_abs_out));
}

/****
 *
 *  Init array with random walk data
 *
 */
void init_array(float* data,int size) {
  data[0]=(float)0;
  int value=0;
  for(int t=1;t<size;t++) {
    if(FLAG_INPUT_TRIVIAL_LINE) {
      data[t]=t;
    }
    else {
      float r=rand()/(float)RAND_MAX;
      if(r<GAMMA) value+=1;
      else if(r>(1-GAMMA)) value-=1;
      float s=rand()/(float)RAND_MAX;
      if(s>0.5) data[t]=value+1;
      else data[t]=value;
    }
    if(FLAG_INPUT_DATA) printf("%d %f\n",t,data[t]);
  }
}

/****
 *
 *  Load array
 *
 */
void load_array(float* data,int size) {

  FILE *fp;
  float offset=0;
  fp=fopen("p.filter","r");
  for(int t=0;t<size;t++) {
    fscanf(fp,"%f",&(data[t]));
    if(t==0) offset=data[0];
    data[t]=(float)(int)round((data[t]-offset)*100);
    if(FLAG_INPUT_DATA && t<FLAG_INPUT_DATA) printf("%d %f\n",t,data[t]);
```

```
  }
  fclose(fp);
}

/****
 *
 *  Device function main
 *
 */
__global__ void device_function(float* in,float* buffer,float* buffer_abs,int
delta_t_minus,int tau, int t) {

  //Allocate arrays
  __shared__ float r[DELTA_T_MINUS_MAX];
  __shared__ float c[DELTA_T_MINUS_MAX+BLOCK_SIZE+DELTA_T_PLUS_MAX];
  __shared__ float rf[DELTA_T_PLUS_MAX];

  //Load reference pattern
  if(threadIdx.x<DELTA_T_PLUS_MAX) {
    rf[threadIdx.x]=in[t+threadIdx.x];
    if(threadIdx.x<delta_t_minus) {
      r[threadIdx.x]=in[t-delta_t_minus+threadIdx.x];
    }
  }

  //Load comparison pattern sequence
  c[threadIdx.x+delta_t_minus]=in[tau+threadIdx.x+blockIdx.x*BLOCK_SIZE];
  if(threadIdx.x<DELTA_T_PLUS_MAX) {

c[delta_t_minus+BLOCK_SIZE+threadIdx.x]=in[tau+threadIdx.x+blockIdx.x*BLOCK_SIZE
+BLOCK_SIZE];
    if(threadIdx.x<delta_t_minus) {
      c[threadIdx.x]=in[tau-delta_t_minus+threadIdx.x+blockIdx.x*BLOCK_SIZE];
    }
  }
  __syncthreads();


  //Calculate high and low of the reference and comparison pattern
  float r_high=r[0];
  float r_low=r[0];
  for(int i=0;i<delta_t_minus;i++) {
    if(r[i]>r_high) r_high=r[i];
    if(r[i]<r_low) r_low=r[i];
  }
  float c_high=c[threadIdx.x];
  float c_low=c[threadIdx.x];
  for(int i=0;i<delta_t_minus;i++) {
    if(c[i+threadIdx.x]>c_high) c_high=c[i+threadIdx.x];
    if(c[i+threadIdx.x]<c_low) c_low=c[i+threadIdx.x];
  }

  //True ranges
  float r_true_range=r_high-r_low;
  float c_true_range=c_high-c_low;

  //Fit
  float fit_value=0;
  for(int i=0;i<delta_t_minus;i++) {
    float displacement=0;
    if(r_true_range>0 && c_true_range>0) {

displacement=(r[i]-r_low)/r_true_range-(c[i+threadIdx.x]-c_low)/c_true_range;
    }
```

```
    else displacement=1;
    fit_value+=displacement*displacement;
  }
  fit_value=fit_value/delta_t_minus;

  //Calculate job id
  int job=blockIdx.x*BLOCK_SIZE+threadIdx.x;

  //Calculate prediction quality
  for(int delta_t_plus=0;delta_t_plus<DELTA_T_PLUS_MAX;delta_t_plus++) {
    float rescaled_r_price=(r[delta_t_minus-1]-r_low);
    if(r_true_range>0) rescaled_r_price=rescaled_r_price/r_true_range;

    float rescaled_r_value=(rf[delta_t_plus]-r_low);
    if(r_true_range>0) rescaled_r_value=rescaled_r_value/r_true_range;
    rescaled_r_value=rescaled_r_value-rescaled_r_price;

    float rescaled_c_value=(c[delta_t_minus+delta_t_plus+threadIdx.x]-c_low);
    if(c_true_range>0) rescaled_c_value=rescaled_c_value/c_true_range;
    rescaled_c_value=rescaled_c_value-rescaled_r_price;

    float quality_factor=0;
    float alpha=rescaled_r_value*rescaled_c_value;
    float result=0;

    if(FLAG_VERSION_CUT_OFF) {
      if(fit_value<OMEGA) {
     if(alpha>0) quality_factor=1;
     else if(alpha<0) quality_factor=-1;
      }
      result=quality_factor;
    }
    else {
      if(alpha>0) quality_factor=1;
      else if(alpha<0) quality_factor=-1;

      //Exponentional weighting
      result=quality_factor*exp(-fit_value*CHI);
    }

    //Save
    buffer[job*DELTA_T_PLUS_MAX+delta_t_plus]=result;
    buffer_abs[job*DELTA_T_PLUS_MAX+delta_t_plus]=fabs(result);

  }
}

/****
 *
 *  Device function post processing
 *
 */
__global__ void device_function_postprocessing(float* in,int offset) {

  __shared__ float in_s[2*DELTA_T_PLUS_MAX];
  in_s[threadIdx.x]=in[2*blockIdx.x*offset+threadIdx.x];
  in_s[DELTA_T_PLUS_MAX+threadIdx.x]=in[2*blockIdx.x*offset+offset+threadIdx.x];
  __syncthreads();

  in_s[threadIdx.x]=in_s[threadIdx.x]+in_s[DELTA_T_PLUS_MAX+threadIdx.x];
  in[2*blockIdx.x*offset+threadIdx.x]=in_s[threadIdx.x];
}

/****
```

```
 *
 *  Device function final processing
 *
 */
__global__ void device_function_finalprocessing(float* in,float* out,float*
in_abs,float* out_abs,int delta_t_minus) {

  if(blockIdx.x==0) {
    out[(delta_t_minus-1)*DELTA_T_PLUS_MAX+threadIdx.x]+=in[threadIdx.x];
  }
  else {

out_abs[(delta_t_minus-1)*DELTA_T_PLUS_MAX+threadIdx.x]+=in_abs[threadIdx.x];
  }
}

/****
 *
 *  Device function init
 *
 */
__global__ void device_function_init(float* in) {

  in[blockIdx.x*DELTA_T_PLUS_MAX+threadIdx.x]=0;
}

/****
 *
 *  CPU function
 *
 */
void cpu_function(float* p,double* pattern_conformity,double* pattern_conformi-
ty_counter) {

  for(int delta_t_minus=1;delta_t_minus<=DELTA_T_MINUS_MAX;delta_t_minus++) {
    for(int
t=2*delta_t_minus+DELTA_T_PLUS_MAX+SCAN_INTERVAL;t<T-DELTA_T_PLUS_MAX;t++) {

      //Calculate high and low of the reference pattern
      double reference_pattern_high=p[t-1];
      double reference_pattern_low=p[t-1];
      for(int t_offset=1;t_offset<=delta_t_minus;t_offset++) {
     if(p[t-t_offset]>reference_pattern_high) reference_pattern_high=p[t-
t_offset];
     if(p[t-t_offset]<reference_pattern_low) reference_pattern_low=p[t-
t_offset];
      }

      //Loop over possible comparison pattern
      for(int
tau=t-delta_t_minus-DELTA_T_PLUS_MAX-SCAN_INTERVAL;tau<t-delta_t_minus-DELTA_T_P
LUS_MAX;tau++) {

     //Calculate high and low of the current comparison pattern
     double comparison_pattern_high=p[tau-1];
     double comparison_pattern_low=p[tau-1];
     for(int tau_offset=1;tau_offset<=delta_t_minus;tau_offset++) {
       if(p[tau-tau_offset]>comparison_pattern_high)
comparison_pattern_high=p[tau-tau_offset];
       if(p[tau-tau_offset]<comparison_pattern_low)
comparison_pattern_low=p[tau-tau_offset];
     }

     //True ranges
```

```
     double reference_true_range=reference_pattern_high-reference_pattern_low;
     double
comparison_true_range=comparison_pattern_high-comparison_pattern_low;

     //Fit
     double fit_value=0;
     for(int t_offset=1;t_offset<=delta_t_minus;t_offset++) {
       double displacement=0;
       if(reference_true_range>0 && comparison_true_range>0) {

displacement=(p[t-t_offset]-reference_pattern_low)/reference_true_range-(p[tau-t
_offset]-comparison_pattern_low)/comparison_true_range;
       }
       else displacement=1;
       fit_value+=displacement*displacement;
     }
     fit_value=fit_value/delta_t_minus;

     //Calculate prediction quality
     for(int delta_t_plus=0;delta_t_plus<DELTA_T_PLUS_MAX;delta_t_plus++) {
       double rescaled_reference_price=(p[t-1]-reference_pattern_low);
       if(reference_true_range>0)
rescaled_reference_price=rescaled_reference_price/reference_true_range;

       double
rescaled_reference_value=(p[t+delta_t_plus]-reference_pattern_low);
       if(reference_true_range>0)
rescaled_reference_value=rescaled_reference_value/reference_true_range;

rescaled_reference_value=rescaled_reference_value-rescaled_reference_price;

       double
rescaled_comparison_value=(p[tau+delta_t_plus]-comparison_pattern_low);
       if(comparison_true_range>0)
rescaled_comparison_value=rescaled_comparison_value/comparison_true_range;

rescaled_comparison_value=rescaled_comparison_value-rescaled_reference_price;

       double quality_factor=0;
       double alpha=rescaled_reference_value*rescaled_comparison_value;
       double result=0;

       if(FLAG_VERSION_CUT_OFF) {
         if(fit_value<OMEGA) {
           if(alpha>0) quality_factor=1;
           else if(alpha<0) quality_factor=-1;
         }
         result=quality_factor;
       }
       else {
         if(alpha>0) quality_factor=1;
         else if(alpha<0) quality_factor=-1;

         //Exponentional weighting
         result=quality_factor*exp(-fit_value*CHI);
       }

       //Save

pattern_conformity[(delta_t_minus-1)*DELTA_T_MINUS_MAX+delta_t_plus]+=result;

pattern_conformity_counter[(delta_t_minus-1)*DELTA_T_MINUS_MAX+delta_t_plus]+=fa
bs(result);
     }
```

```
      }
    }
  }
}

/****
 *
 *  Print result
 *
 */
void print_result(float* pc) {

  for(int delta_t_minus=1;delta_t_minus<DELTA_T_MINUS_MAX;++delta_t_minus) {
    for(int delta_t_plus=0;delta_t_plus<DELTA_T_PLUS_MAX;++delta_t_plus) {
      printf(" %f",(pc[delta_t_minus*DELTA_T_MINUS_MAX+delta_t_plus]));
    }
    printf("\n");
  }
}

/****
 *
 *  Print result
 *
 */
void print_result(double* pc) {

  for(int delta_t_minus=1;delta_t_minus<DELTA_T_MINUS_MAX;++delta_t_minus) {
    for(int delta_t_plus=0;delta_t_plus<DELTA_T_PLUS_MAX;++delta_t_plus) {
      printf(" %f",(pc[delta_t_minus*DELTA_T_MINUS_MAX+delta_t_plus]));
    }
    printf("\n");
  }
}
```