Leibniz Universität Hannover

Leibniz Universität Hannover

Bachelor degree project

Search for new radio pulsars via coincidence in the Einstein@Home and Cornell result databases

Author:
Tjark Miener 2860000

Supervisor:
Prof. Dr. Bruce Allen

November 24, 2014

Declaration of autonomy

I hereby declare to have written the present work independently and only with help of specified sources and resources. The passages in the text, which, in their wording or meaning, were taken from other sources are clearly identified through their stated origin. The present work has not been submitted for another degree or diploma at any university or other institute of higher education.

Hanover, the 24th of November, 2014

Abstract

The task of this bachelor degree project is to find new radio pulsars via coincidence in the Einstein@Home and Cornell result databases. The result databases contain millions of pulsar candidates, which were processed by Einstein@Home and the Cornell University.

The idea of the coincident search is to compare these candidates with each other. Finding candidates with similar frequencies and coordinates, which both databases contained at different observation times, is the goal of this search.

Within the project the described procedure delivered five matches of already known pulsars (B1855+02, B1913+10, B1914+13, B1929+20, J1935+2025). This project seems to be a good preparatory work for further investigations, which may follow, since new pulsars haven't been discovered in this short amount of time.

Contents

1 Introduction **1**

2 Processing the data **5**

2.1 Einstein@Home 5

2.2 Data structures 5

2.3 General approach of the algorithm 8

2.4 Explanation of the code 9

3 Results **25**

3.1 Further processing of the output 25

3.2 The identification of previously discovered pulsars 26

3.3 The remain clusters with "best" candidates 28

4 References **33**

1 Introduction

Call me crazy but, I imagine a world where we smile when we have low batteries.
Cuz that will mean we'll be one bar closer, to Humanity.

***(Richard Williams**, musician)*

Beforehand, the most relevant characteristics of a neutron star and a pulsar are described. The remains of a supernova (stellar explosion) evolve to three different objects, depending on the mass of the core of the original star. In the following the solar mass of $1.989 \cdot 10^{30}\,\text{kg}$ is written as $M_\odot$ [1]. A **neutron star** is formed, when the mass of the core is located between $1.5M_\odot$ (Chandrasekhar limit) and $3M_\odot$ (Tolman–Oppenheimer–Volkoff limit). The Chandrasekhar limit is the highest possible mass of a white dwarf. When the core mass of the original star is greater then the Tolman–Oppenheimer-Volkoff limit, a black hole is formed [2].

The neutron star possesses nearly the same mass as the core of the original star, although the size of the star reduces to a diameter of roughly 20 kilometers after the supernova. This implies an enormous rise in the density of the resulting neutron star. The average density of neutron stars is $2.0 \cdot 10^{26} \frac{kg}{km^3}$ [3].

Radio pulsars (<u>pulsa</u>ting source of <u>r</u>adio emission) are rapidly rotating, highly magnetized neutron stars. About 2400 pulsars have been found in the past 50 years [4]. The basic model for a pulsar is shown in figure 1 [5]. The magnetic axis differs from the rotation axis. The radiation beam, in the form of radio waves, is emitted along the dipole axis. When the radio beam hits the earth, earthbound telescopes can measure a signal. Because of the rotation of the neutron star, this signal is nearly periodic (*lighthouse effect*) [6].

Pulsars can be used to study an extremely wide range of physical and astrophysical problems. These unique and versatile objects give important information about the gravitational potential and magnetic fields of the galaxy and the interstellar medium, stars, binary systems and their evolution. However, unanswered questions remain (e.g. *What is the structure of neutron stars or of their magnetic and electric fields?*, *What is the composition of neutron star atmosphere?*, *What is the radio emission mechanism?*). Finding new radio pulsars could help us to answer these questions [6].

A milestone of radio pulsar astronomy was the discovery of the first pulsar (B1919+21) in 1967 by Jocelyn Bell and Antony Hewish, which was recognized with the Nobel Prize for Physics in 1974 [7]. The second breakthrough was the discovery (1974) and analysis of the pulsar B1913+16, better known as **Hulse-Taylor binary pulsar** [8]. This provides indirect evidence for gravitational waves according to the general theory of relativity. In recognition of their achievement, Hulse and Taylor also received the Nobel Prize for Physics in 1993 [9].

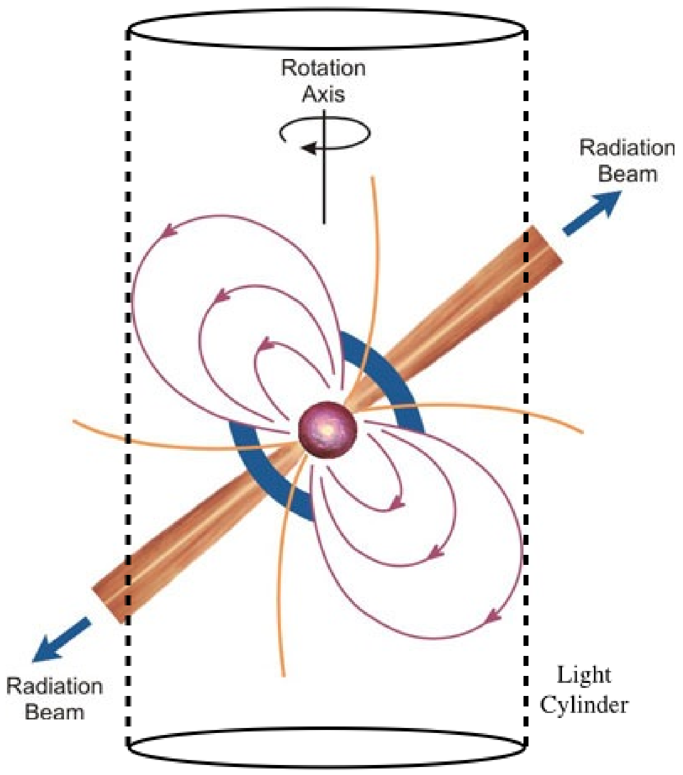

Figure 1: This basic model for a pulsar and its magnetosphere (not drawn to scale!) is originated from the McGill Pulsar Group [5]. The purple ball in the middle represents the pulsar and the closed magnetic field lines are purple. Afterwards a light cylinder, which contains all closed magnetic field lines, was added. The light cylinder represents the maximum size of the magnetosphere. The radius of the light cylinder is defined by $R \cdot \omega = c$, where $\omega = 2\pi \cdot f$ and f is the pulsar rotation frequency.

Further important discoveries:

- The first millisecond pulsar B1937+21 by Kulkarni, Backer and collaborators [10]
- The first millisecond pulsar in a globular cluster M28 by Lyne and collaborators [11]
- The first pulsar planetary system B1257+12 by Wolszczan and Frail [12]
- The first triple system B1620-26: a pulsar, a white dwarf and a Jupiter-mass planet by Thorsett and collaborators in globular cluster M4 [13]
- The first ”double pulsar” system by Burgay and collaborators [14]

The data in this project originates from the Arecibo radio telescope, which is located in Puerto Rico. Most of the raw data was reprocessed by the **Cornell University** and the **Max Planck Institute for Gravitational Physics** (Albert Einstein Institute) (see figure 2). The Albert Einstein Institute is using the distributed computing project **Einstein@Home** for creating the result database. The method of the Cornell University is mentioned in section 4.9.2. *(II) Whitening/RFI Zapping* of [15]. The structure of the pulsar candidates are explained in section 2.2 *Data structures.*

The task of the program, shown in the appendix, is to detect pairs of "related" candidates in the two result databases (Einstein@Home and Cornell). Candidates are related, when they have similar coordinates and angular frequencies. In addition, related candidates are more significant, if they are identified in different data sets in both databases, meaning that the candidates were recorded at different points of time. This is a good way to sort out candidates, which emerged from **temporary** radio noises and radio frequency interference (RFI). Some of these interfering signals emanate from intended transmissions, such as radio and TV stations or cell phones. However, there are **persistent** RFI as well. These undesired signals appear in many different sky positions at different observation times. In contrast to RFI signals of all sorts, a specific pulsar signal is time-independent and can only be measured in one sky region. A more detailed explanation of the filter process is shown in section 3.1 *Further processing of the output.*

2 Processing the data

2.1 Einstein@Home

Einstein@Home is a volunteer distributed computing project, which uses the computer idle time of hundreds of thousands volunteers from 193 countries, to search for weak astrophysical signals from pulsars using raw data from the LIGO gravitational-wave detectors, the Arecibo radio telescope, and the Fermi gamma-ray satellite [16]. This project was formally launched at the American Association for the Advancement of Science meeting on 2005 February 19, as one of the cornerstone activities of the World Year of Physics 2005. The search for new radio pulsars with Einstein@Home is accomplished by using **Pulsar ALFA survey data** from the Arecibo 305-meter telescope and the ALFA multibeam receivers. Approximately 176 million results have been generated and completed in the Einstein@Home search of the Pulsar ALFA dataset [15].

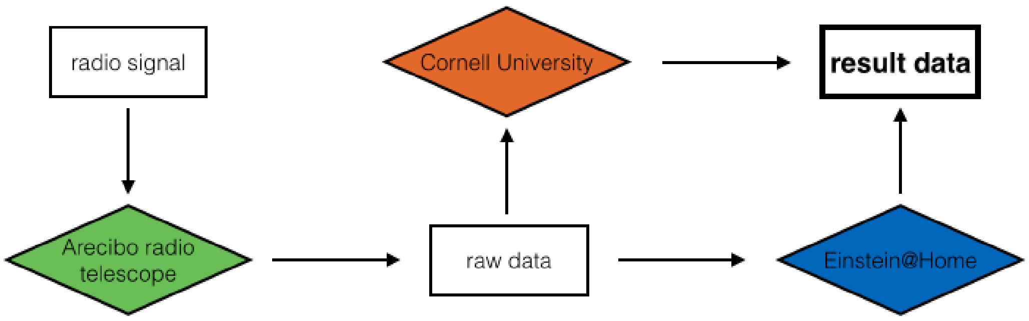

Figure 2: The flow diagram of the processing of the radio signal measured by the Arecibo radio telescope in Puerto Rico.

2.2 Data structures

The result data originates from the **Einstein@Home database** of the Albert Einstein Institute and the **Cornell database** of the Cornell University. The file of the Cornell database (five columns) is merged with the file of the Einstein@Home database (six columns), so that there is only one input file with the following characterizing information:

1. Candidate numeration (Nr)
   Negative integers decreasing from -1 to -1265589 for Einstein@Home candidates and for Cornell candidates positive integers increasing from 1 to 4804805.

2. Candidate ID (ID)
   Example: A candidate was recorded on 2009 March 17 at sky location G35.41-02.95.N with the beam number b0s0g0 and the candidate number 00000_14944. Altogether, the candidate ID is composited to 20090317.G35.41-02.95.N.b0s0g0.0000 0_14944 (*date.galactic_coordinate.beam_number.candidate_number*).
   Caution: candidate numeration and candidate number are unequal!

3. Right ascension in J2000 epoch (RAJ)
   The Einstein@Home candidates are declared in hours and fractional hours and the Cornell's in degrees and fractional degrees.

4. Declination in J2000 epoch ($DecJ$)
   The Einstein@Home as well as the Cornell candidates are declared in degrees and fractional degrees.

5. Angular frequency (f)
   Both databases are declared the angular frequency of the rotating pulsar candidate in Hertz.

6. Significance (s)
   Only Einstein@Home candidates has a significance, which is defined as

   $$s \equiv -\log(p_{FA})$$

   with the false-alarm probability p_{FA}. As the equation above shows, the higher the significance, the higher the probability of the candidate being a pulsar. A detailed description of the derivation is shown in section 4.3 *Signal Model and Detection Statistic* of [15].

Table 1: The input file consists of the Einstein@Home and Cornell result databases. The file contains 6070393 candidates. Besides a frequency and specific coordinates, each candidate has a number and an ID. The Einstein@Home candidates have a significance as well.

Nr	**ID**	**RAJ**	**DecJ**	**f**	**s**
-1	20090317.G35.41-02.95.N.b0s0g0.00000_14944	19.106888888888875	0.917583333333335	59.849652377042	24.2328
-2	20090317.G35.41-02.95.N.b0s0g0.00000_10535	19.106888888888875	0.917583333333335	42.235461148349	21.126
-3	20090317.G35.41-02.95.N.b0s0g0.00000_22444	19.106888888888875	0.917583333333335	89.866117997603	18.2292
-4	20090317.G35.41-02.95.N.b0s0g0.00000_1564	19.106888888888875	0.917583333333335	6.289742209695	18.2275
⋮	⋮	⋮	⋮	⋮	⋮
-1265588	20140627.G61.50-01.85.C.b5s0g0.00000_5244	19.895663670563778	24.282206369513009	21.009098399769	13.5594
-1265589	20140627.G61.50-01.85.C.b5s0g0.00000_59185	19.895663670563778	24.282206369513009	237.335205078125	13.4458
1	20110112.G193.10-03.16.N.b3.00000_1	90.56429833	16.03619522	0.837937759123	
2	20110112.G193.10-03.16.N.b3.00000_2	90.56429833	16.03619522	59.9706663146	
3	20110112.G193.10-03.16.N.b3.00000_3	90.56429833	16.03619522	167.897109201	
4	20110112.G193.10-03.16.N.b3.00000_4	90.56429833	16.03619522	459.083163229	
⋮	⋮	⋮	⋮	⋮	
4804804	20120105.G176.91-03.77.S.b0.00000_4804804	80.8622075	29.44760894	403.690721299	
4804805	20120105.G176.91-03.77.S.b0.00000_4804805	80.8622075	29.44760894	718.691331543	

2.3 General approach of the algorithm

As the content of the databases is very large, the algorithm has to be as efficient as possible. Obviously, the worst case is, if all N candidates are compared with one another $\left(\frac{N\cdot(N-1)}{2} \text{ comparisons}\right)$. Since the desired output should consists of candidates with similar coordinates and frequencies, the following method is used:

Firstly, the surface of the celestial sphere is divided into horizontal stripes of the same angular width. The horizontal stripes are then uniformly subdivided vertically, with the result of cell-like division of the celestial sphere, shown in figure 3.

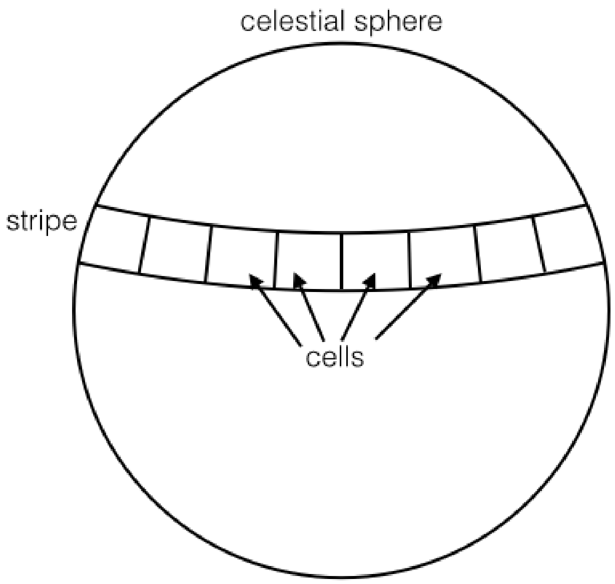

Figure 3: An extract of the cell-like division of the celestial sphere.

As a result of the sphere's arched surface, less cells are located on the top and bottom than around the equator of the sphere. Afterwards, the candidates are positioned in these cells and sorted according to their appropriate cell number. Figure 4 explains the features, how to compare candidates with each other.

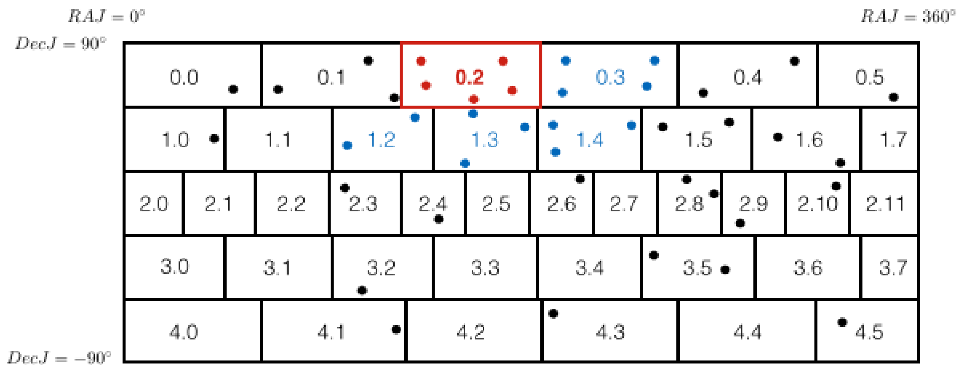

Figure 4: The basic procedure of comparing candidates. The candidates, represented as small colored points, are positioned in the different cells. Because the coordinates of the related candidates are very close to each other, it is only necessary to compare candidates to their neighboring cells. Certainly, candidates in the same cell are also compared to each other as well. Within the same cell, there is only one comparison between two candidates, because of the symmetry of the distance. The comparison between candidates, located at the beginning or end of a stripe, is much more complex. A detailed description is shown in figure 5 in section 2.4 *Explanation of the code.*
Example: The candidates of cell 0.2 are marked **red**. The **blue** points are candidates of neighboring cells, which **haven't already been compared** to the nearby candidates of cell 0.2. Therefore, all points of cell 0.2 have to be compared to the blue points and to each other.

2.4 Explanation of the code

The code is created in the programming language **C**. Instead of showing the whole code, only important parts of the code are explained in the following. It is helpful to point out the key aspects of the functioning of the program.

These six C standard libraries are used in the program.

```
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <math.h>
#include <stdbool.h>
#include <string.h>
```

In addition to that, the variables below are global running indices.

```
int i,j,k,l,m;
```

A structure called Skypoint, with three integers, three double values, one float value and one array of 64 characters, is utilized [17]. This structure includes the position (phi and theta coordinates) along with its cell number (p_cell and t_cell), as well as the frequency and the significance of the respectively located pulsar-candidate. Moreover, the candidates have an ID and are numbered consecutively. This is shown in the following code extract.

```
#define Maximum_scan 64


struct Skypoint{
    int number;
    char id[Maximum_scan];

    double phi;
    double theta;

    int p_cell;
    int t_cell;

    double frequency;
    float significance;
};
```

As mentioned in section 2.3 *General approach of the algorithm*, the surface of the celestial sphere is split up into uniform cells, so the cell number is included in the structure Skypoint. The number of cells in each stripe is calculated with the function *sin()* of <math.h>. The properties of *sin()* are equivalent to the consequences of the sphere's arched surface. The circumference of the sphere is maximal at the equator and zero at the poles. In terms of the angular size of the Arecibo beam (2 minutes of arc), the horizontal stripes (Stripestheta) set to 3001.

```
#define Stripestheta 3001
#define Stripesphi (2*Stripestheta)


int num_in_stripe[Stripestheta];


for (i=0; i<Stripestheta; i++) {
        num_in_stripe[i]=ceil(Stripesphi*sin(M_PI*(i+0.5)/Stripestheta));
}
```

The cells are computed, using the function *cellcomputing()*, which uses *num_in_stripe[]*:

```
void cellcomputing(struct Skypoint *a){

    if ((a->theta)<M_PI) {
        a->t_cell= (int)((a->theta)*Stripestheta/M_PI);
    } else {
        a->t_cell=Stripestheta-1;
    }
    a->p_cell=floor((a->phi)*(num_in_stripe[a->t_cell]-1)/(2.0*M_PI));
    return;
}
```

The function *fopen()* opens the candidate list, enabling the data to be processed [17].

```
FILE *input_candidate = fopen(argv[1],"r");
```

```
if (!input_candidate) {
        printf("Error! File %s not found\n", argv[1]);
        exit(4);
}
```

As a consequence of the unpredictable data volume, a dynamic memory allocation is useful. Whenever a new block of memory is needed, the program calls the function *my_Allocate()*, which calls the function *realloc()* of <stdlib.h>[17].

```
struct Skypoint *list=NULL;


void my_Allocate(int a) {
    list=realloc(list,a*sizeof(struct Skypoint));
    if (!list) {
        printf("Error! No more RAM available, skypoints_max = %d!\n", a);
        exit(1);
    }
}
```

By means of *fscanf()* of <stdio.h>, the input is inserted in the structure Skypoint [17]. The coordinates of the points are transformed from RAJD and DecJD into phi and theta (declared in radians).

$$\phi = \frac{2\pi}{360} \cdot RAJD$$

$$\theta = \frac{2\pi}{360} \cdot (90 - DecJD)$$

Afterwards, as a verification, it is checked if the coordinates are in the permitted range ($\phi \in [0, \ldots, 2\pi]$ and $\theta \in [0, \ldots, \pi]$). Subsequently, the function *cellcomputing()* is called to compute the cell along with its p_cell and t_cell.

```
#define Allocate_blocksize 1024
#define End_of_first_list 1265589


int skypoints_read=0;
int skypoints_allocated=0;


while (true) {

        //reallocating memory if necessary
        if (skypoints_read==skypoints_allocated) {
            skypoints_allocated += Allocate_blocksize;
            my_Allocate(skypoints_allocated);
        }

        if (skypoints_read < End_of_first_list) {

            //scan the elements of the first part of the file
            if (6==(i=fscanf(input_candidate, "%d %s %lf %lf %lf %f",&list[
                skypoints_read].number, list[skypoints_read].id,&list[
                skypoints_read].phi,&list[skypoints_read].theta,&list[
                skypoints_read].frequency,&list[skypoints_read].
                significance))) {

                list[skypoints_read].phi=(list[skypoints_read].phi*15.0)
                    *2.0*M_PI/360.0;
                list[skypoints_read].theta=(90.0-list[skypoints_read].theta
                    )*2.0*M_PI/360.0;

                //check the data ranges, exit if error
                if (list[skypoints_read].phi < 0 || list[skypoints_read].
                    phi > 2*M_PI || list[skypoints_read].theta < 0 || list[
                    skypoints_read].theta > M_PI) {
                    printf("Error! Problem reading %s at line %d. Point (%f
                        ,%f) isn't in the data range!\n", argv[1],
                        skypoints_read, list[skypoints_read].phi, list[
                        skypoints_read].theta);
                    exit(5);
                }

                cellcomputing(list+skypoints_read);
                skypoints_read++;

            } else break;

        } else {

            //scan the elements of the second part of the file
            if (5==(i=fscanf(input_candidate, "%d %s %lf %lf %lf",&list[
                skypoints_read].number, list[skypoints_read].id,&list[
                skypoints_read].phi,&list[skypoints_read].theta,&list[
                skypoints_read].frequency))) {

                list[skypoints_read].phi=list[skypoints_read].phi*2.0*M_PI
                    /360.0;
                list[skypoints_read].theta=(90.0-list[skypoints_read].theta
                    )*2.0*M_PI/360.0;
                list[skypoints_read].significance=0.0;

                //check the data ranges, exit if error
                if (list[skypoints_read].phi < 0 || list[skypoints_read].
                    phi > 2*M_PI || list[skypoints_read].theta < 0 || list[
                    skypoints_read].theta > M_PI) {
                    printf("Error! Problem reading %s at line %d. Point (%f
                        ,%f) isn't in the data range!\n", argv[1],
                        skypoints_read, list[skypoints_read].phi, list[
                        skypoints_read].theta);
                    exit(5);
                }

                cellcomputing(list+skypoints_read);
                skypoints_read++;
            } else break;

        }

}

if (i!=EOF) {
        fprintf(stderr, "Error! Problem reading %s at line %d (number:%d).
            fscanf() returned %d\n", argv[1], skypoints_read, list[
            skypoints_read].number, i);
        exit(6);
}
```

The special case, where candidates are located at the beginning or end of a stripe, is solved with the idea described in figure 5.

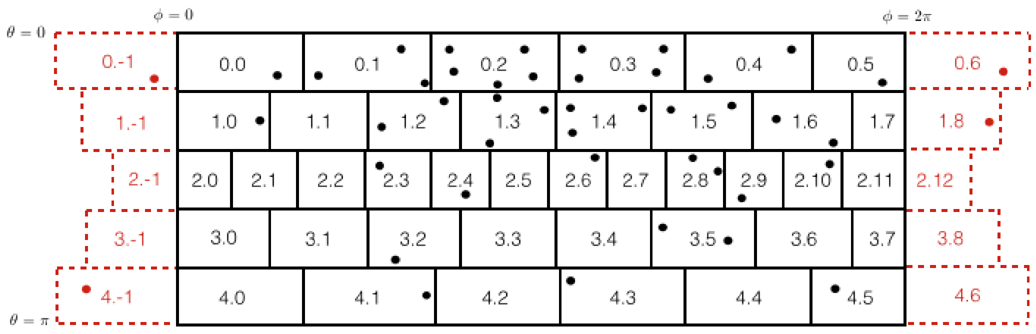

Figure 5: The procedure of creating shadow points. The black points in the different cells represent the candidates. If a candidate is located at the beginning or end of a stripe, a so-called shadow-point (**red**) of the original candidate is generated and added to the list of candidates. As a consequence, the algorithm's special case "border case" is neutralized and therefore the program becomes less complex.

```
k=j=0;

for (i=0; i<skypoints_read; i++) {

        //reallocating memory if necessary
        if ((skypoints_read+k)==skypoints_allocated && j==1) {
            skypoints_allocated += Allocate_blocksize;
            my_Allocate(skypoints_allocated);
        }

        //adding shadow points on the right
        if (list[i].p_cell < 2){

            list[skypoints_read+k].phi=((list[i].phi)+2.0*M_PI);
            list[skypoints_read+k].theta=list[i].theta;
            list[skypoints_read+k].t_cell=list[i].t_cell;
            list[skypoints_read+k].p_cell=list[i].p_cell+num_in_stripe[list
                [i].t_cell];
            list[skypoints_read+k].significance=list[i].significance;
            list[skypoints_read+k].frequency=list[i].frequency;
            list[skypoints_read+k].number=list[i].number;
            strncpy(list[skypoints_read+k].id, list[i].id, Maximum_scan);

            k++;
            j=1;

        //adding shadow points on the left
        } else if (list[i].p_cell > num_in_stripe[list[i].t_cell]-3) {

            list[skypoints_read+k].phi=((list[i].phi)-2.0*M_PI);
            list[skypoints_read+k].theta=list[i].theta;
            list[skypoints_read+k].t_cell=list[i].t_cell;
            list[skypoints_read+k].p_cell=list[i].p_cell-num_in_stripe[list
                [i].t_cell];
            list[skypoints_read+k].significance=list[i].significance;
            list[skypoints_read+k].frequency=list[i].frequency;
            list[skypoints_read+k].number=list[i].number;
            strncpy(list[skypoints_read+k].id, list[i].id, Maximum_scan);

            k++;
            j=1;
        } else {
            j=0;
        }

}

skypoints_read += k;
```

The list is sorted by the function *qsort()* of <stdlib.h>, which utilizes a quicksort algorithm [17].

```
qsort (list,skypoints_read,sizeof(struct Skypoint), compare);
```

The fourth parameter in *qsort()* is a function, which orders two elements of the list. In the first place, the horizontal cell numbers (t_cell) are compared with one another. The element with the smaller t_cell number is placed ahead of the other element in the list. If the two elements are in the same stripe, the same method is used with the vertical cell numbers (p_cell).

```
int compare(const void *a, const void *b){
    const struct Skypoint *elementa = a;
    const struct Skypoint *elementb = b;

    if (elementa->t_cell > elementb->t_cell) return 1;
    if (elementa->t_cell < elementb->t_cell) return -1;

    if (elementa->p_cell > elementb->p_cell) return 1;
    if (elementa->p_cell < elementb->p_cell) return -1;

    return 0;
}
```

The distance between two points on the celestial sphere ($r = 1$) is computed with the aid of the scalar product $|\vec{a}||\vec{b}| \cos \sphericalangle(\vec{a}, \vec{b}) = \vec{a} \cdot \vec{b}$ [18].

$$\cos\gamma = \begin{pmatrix} x_1 \\ y_1 \\ z_1 \end{pmatrix} \cdot \begin{pmatrix} x_2 \\ y_2 \\ z_2 \end{pmatrix} = \begin{pmatrix} \sin\theta_1 \cos\phi_1 \\ \sin\theta_1 \sin\phi_1 \\ \cos\theta_1 \end{pmatrix} \cdot \begin{pmatrix} \sin\theta_2 \cos\phi_2 \\ \sin\theta_2 \sin\phi_2 \\ \cos\theta_2 \end{pmatrix}$$

Thereafter, the equation is solved for γ.

$$\gamma = \arccos(\sin\theta_1 \cdot \sin\theta_2 \cdot \cos(\phi_1 - \phi_2) + \cos\theta_1 \cdot \cos\theta_2)$$

In the code, a function, called *angle()*, is used for computing γ, which is declared in degrees.

```
double angle(struct Skypoint *a,struct Skypoint *b) {
    double x = sin(a->theta)*sin(b->theta)*cos((a->phi)-(b->phi))+cos(a->
        theta)*cos(b->theta);
    if (x > 1.0) {
        x = 1.0;
    }
    return acos(x)*(180.0/M_PI);
}
```

Moreover, the output is created with the function *print_output()*. This function only operates if the two candidates originate from different databases and data sets. Furthermore, the difference between the frequencies must be smaller than 0.01 Hz. The output file possesses ten columns with the following characterizing information:

1. CandidateID (Einstein@home)
2. CandidateID (Cornell)
3. Frequency in *Hz* (Einstein@home)
4. Frequency in *Hz* (Cornell)
5. Azimuthal angle ϕ in *rad* (Einstein@home)
6. Polar angle θ in *rad* (Einstein@home)
7. Azimuthal angle ϕ in *rad* (Cornell)
8. Polar angle θ in *rad* (Cornell)
9. Significance (Einstein@home)
10. Distance γ in *degrees* between both candidates

```
#define Minimum_frequency 0.01


void print_output(int i,int j,double g){

    char string1[Maximum_scan];
    char string2[Maximum_scan];

    strncpy(string1, list[i].id, Maximum_scan);
    strncpy(string2, list[j].id, Maximum_scan);

    char *ptr1;
    char *ptr2;

    if (list[i].significance==0.0 && list[j].significance!=0.0) {

        if (fabs(list[i].frequency-list[j].frequency) < Minimum_frequency){

            ptr1 = strtok(string1,"b");
            ptr2 = strtok(string2,"b");

            int result = strcmp (ptr1,ptr2);
            if (result != 0) {
                printf("%s %s %f %f %f %f %f %f %f %f\n",
                       list[j].id,list[i].id,list[j].frequency,list[i].
                           frequency,
                       list[j].phi,list[j].theta,list[i].phi,list[i].theta,
                       list[j].significance,g);
            }
        }

    } else if (list[j].significance==0.0 && list[i].significance!=0.0){

        if (fabs(list[i].frequency-list[j].frequency) < Minimum_frequency){

            ptr1 = strtok(string1,"b");
            ptr2 = strtok(string2,"b");

            int result = strcmp (ptr1,ptr2);
            if (result != 0) {
                printf("%s %s %f %f %f %f %f %f %f %f\n",
                       list[i].id,list[j].id,list[i].frequency,list[j].
                           frequency,
                       list[i].phi,list[i].theta,list[j].phi,list[j].theta,
                       list[i].significance,g);
            }
        }
    }
    return;
}
```

In order to compare candidates of the same stripe, it is only necessary to have a closer look on the candidates of the same cell and the right neighboring cell. An example, how the code extract is working, is described in figure 6.

```
int target, runner;
double gamma;

for (target=0; target<skypoints_read; target++) {

        //only take the non shadow point
        if (list[target].phi>=0 && list[target].phi<=2.0*M_PI) {

            for (runner=target+1; runner<skypoints_read; runner++) {

                //target and runner are in the same cell
                if (list[target].p_cell==list[runner].p_cell && list[target
                    ].t_cell==list[runner].t_cell) {
                    gamma=angle(&list[target],&list[runner]);
                    print_output(target,runner,gamma);

                //runner are in the next cell
                } else if(list[target].p_cell==((list[runner].p_cell)-1) &&
                     list[target].t_cell==list[runner].t_cell) {
                    gamma=angle(&list[target],&list[runner]);
                    print_output(target,runner,gamma);

                //runner are too far away -> break
                } else break;
            }
        }
}
```

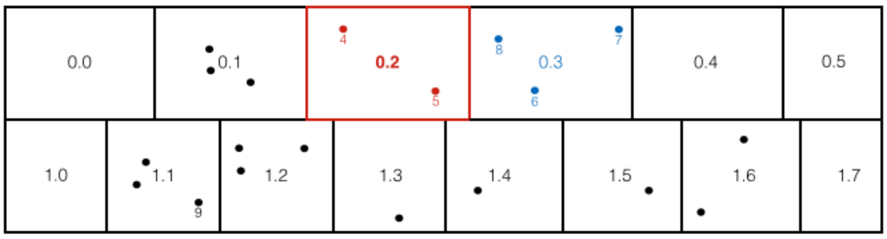

Figure 6: This example starts in the "target loop" at 4. The first iteration of the "runner loop" implies that the runner sets to 5. Because the target and the runner are in the same cell, the angle between them is computed. If these two points obey the condition of *print_out()*, the coincidence between them is printed out. At the next three iteration of the "runner loop" (runner sets to 6, 7 and 8) the runner is in the right neighboring cell and the angle is printed out, if it is necessary. The "runner loop" breaks, because runner equal 9 is not in the next cell to the targetcell. The equivalent procedure is implemented till the "target loop" ends.

In other respects, it is a lot more complicated to compare the candidates, which are not located in the same stripe. Fortunately only the cells in the stripe below are relevant for the investigation, because candidates with a gap of two stripes are too far away for a comparison. Thus, the function *close_enough()* detects whether or not a comparison between the targetcell and the underlying runnercell is needed. The idea of the function is to look at the corners of those cells and to see if any two corner points of different cells are close to each other.

```
#define Gamma_max (2*M_PI/Stripesphi)


int close_enough(struct Skypoint *a, struct Skypoint *b){

    struct Skypoint targetcell_left_bottom;
    struct Skypoint targetcell_right_bottom;
    struct Skypoint runnercell_left_top;
    struct Skypoint runnercell_right_top;

    int p1=a->p_cell;
    int nextp1=(a->p_cell)+1;
    int nextt1=(a->t_cell)+1;

    int p2=b->p_cell;
    int t2=b->t_cell;
    int nextp2=(b->p_cell)+1;

    targetcell_right_bottom.theta=targetcell_left_bottom.theta=nextt1*M_PI/
        Stripestheta;
    targetcell_left_bottom.phi=2*M_PI*p1/num_in_stripe[a->t_cell];
    targetcell_right_bottom.phi=2*M_PI*nextp1/num_in_stripe[a->t_cell];

    runnercell_right_top.theta=runnercell_left_top.theta=t2*M_PI/
        Stripestheta;
    runnercell_left_top.phi=2*M_PI*p2/num_in_stripe[b->t_cell];
    runnercell_right_top.phi=2*M_PI*nextp2/num_in_stripe[b->t_cell];

    double gamma_1=angle(&targetcell_left_bottom,&runnercell_left_top);
    double gamma_2=angle(&targetcell_left_bottom,&runnercell_right_top);
    double gamma_3=angle(&targetcell_right_bottom,&runnercell_left_top);
    double gamma_4=angle(&targetcell_right_bottom,&runnercell_right_top);

    if (gamma_1<Gamma_max || gamma_2<Gamma_max || gamma_3<Gamma_max ||
        gamma_4<Gamma_max) {
        return 1;
    } else {
        return 0;
    }
}
```

Additionally, it is important to know which candidates of the sorted list are in the first position in each stripe. If the stripe is empty, it receives an unallowed and thus distinct number. This way, any empty stripe can be easily recognized in retrospect.

```
#define Empty -12345


int first_point_in_stripe[Stripestheta+1];
int last_stripe;

for (i=0; i<=Stripestheta; i++) {
        first_point_in_stripe[i]=Empty;
}

last_stripe=first_point_in_stripe[list[1].t_cell]=0;
for (i=1; i<skypoints_read; i++) {
        if (list[i].t_cell != last_stripe) {
            last_stripe=list[i].t_cell;
            first_point_in_stripe[list[i].t_cell]=i;
        }
}
```

In the next step, the list is run through until a point that has to be compared to a point in the stripe below, is encountered. This initialization is necessary for the following algorithm.

```
int runner_below_begin, runner_below_end;
int check_point;

for (i=0; i<skypoints_read; i++) {

        //only take the non shadow point
        if (list[i].phi>=0 && list[i].phi<=2.0*M_PI) {

            //if the stripe below is empty, take the next point in the For-
                loop and check!
            if (first_point_in_stripe[((list[i].t_cell)+1)]==Empty)
                continue;

            //check if any point in the stripe below is close_enough()
            for (j=first_point_in_stripe[list[i].t_cell+1]; list[j].t_cell
                ==(list[i].t_cell+1); j++) {
                 if (close_enough(&list[i],&list[j])) {
                     runner_below_begin=j;
                     check_point=j;
                     break;
                 } else check_point=Empty;
            }

            //if no point in the stripe below is close_enough(), take the
                next point in the For-loop and check
            if (check_point==Empty) continue;

            //check how much points in the stripe below are close_enough()
            for (k=runner_below_begin; list[k].t_cell==(list[i].t_cell+1);
                k++) {
                 if (close_enough(&list[i],&list[k])) {
                     runner_below_end=k;
                 } else break;
            }

            break;
        }
}

//compute gamma of the points in the stripe below which are close_enough()
   to the target cell
for (runner=runner_below_begin; runner<=runner_below_end; runner++) {
        gamma=angle(&list[i],&list[runner]);
        print_output(i,runner,gamma);
}
```

Due to the fact that the algorithm is comparing the target of the for-loop to the previous target, there are three possible situations:

1. **The target stays in the same cell.**

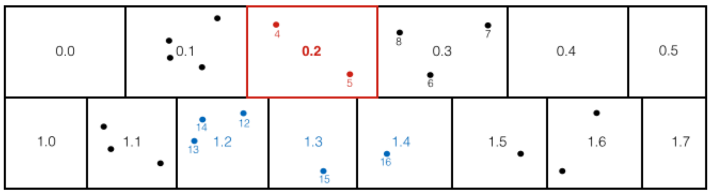

Figure 7: The current target 5 and the previous target 4 are located in the same stripe. The comparative points (12, 13, 14, 15, and 16) don't change after the "target loop" iteration from 4 to 5.

2. **The target moves to a different cell in the stripe.**

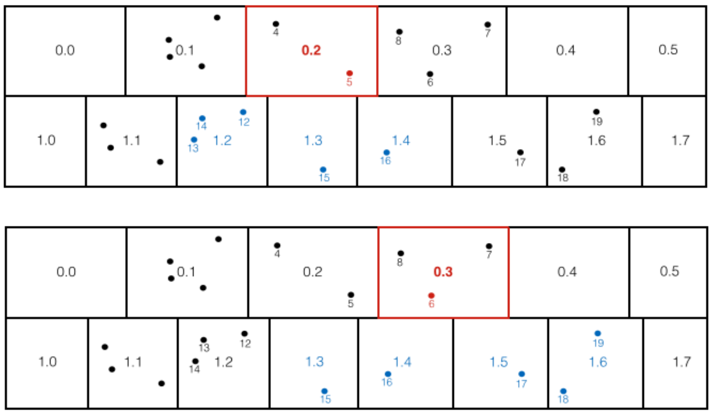

Figure 8: The previous target 5 was located in cell 0.2 and the comparative points were 12, 13, 14, 15 and 16. After the iteration of the "target loop", the comparative points of the current target 6 (cell 0.3) have to synchronize. The new comparative points are 15, 16, 17, 18 and 19.

3. **The target moves to a different stripe.**

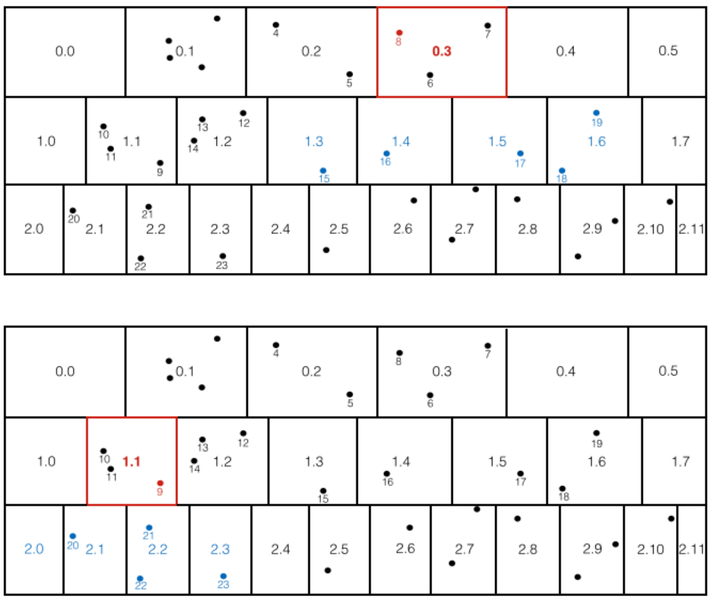

Figure 9: The same procedure as figure 8 is used. The previous target 8 (cell 0.3) with the comparative points (15, 16, 17, 18 and 19) changed to the current target 9 (cell 1.1) with the comparative points (20, 21, 22 and 23).

The integer value i in the for-loop is derived from the initialization above.

```
for (target=i+1; target<skypoints_read; target++) {

        //only take the non shadow point
        if (list[target].phi>=0 && list[target].phi<=2.0*M_PI) {

            //if the stripe below is empty, take the next point in the for
                loop and check
            if (first_point_in_stripe[((list[target].t_cell)+1)]==Empty)
                continue;

            //Part 1

            //the target stays in the same cell (no update of the
                runner_below_begin and runner_below_end)
            if (list[target].p_cell==list[target-1].p_cell && list[target].
                t_cell==list[target-1].t_cell) {

                //if no point in the stripe below is close_enough(), take
                    the next point in the For-loop and check!
                if (check_point==Empty) continue;

                //compute gamma of the points in the cell below which are
                    close_enough() to the target cell
        for (runner=runner_below_begin; runner<=runner_below_end;
            runner++) {
            gamma=angle(&list[target],&list[runner]);
            print_output(target,runner,gamma);
        }

    //Part 2

    //the target moves to a different cell (update of the
        runner_below_begin and runner_below_end)
    } else if (list[target].p_cell!=list[target-1].p_cell && list[
        target].t_cell==list[target-1].t_cell) {

        //check if any point in the stripe below is close_enough()
        for (j=first_point_in_stripe[((list[target].t_cell)+1)];
            list[j].t_cell==list[target].t_cell+1; j++) {
            if (close_enough(&list[target],&list[j])) {
                runner_below_begin=j;
                check_point=j;
                break;
            } else check_point=Empty;
        }

        //if no point in the stripe below is close_enough(), take
            the next point in the for loop and check!
        if (check_point==Empty) continue;

        //check how much points in the stripe below are
            close_enough()
        for (k=runner_below_begin; list[k].t_cell==(list[target].
            t_cell+1); k++) {
            if (close_enough(&list[target],&list[k])) {
                runner_below_end=k;
            } else break;
        }

        //compute gamma of the points in the stripe below which are
             close_enough() to the target cell
        for (runner=runner_below_begin; runner<=runner_below_end;
            runner++) {
            gamma=angle(&list[target],&list[runner]);
            print_output(target,runner,gamma);
        }

    //Part 3

    //the target moves to a different stripe (update of the
        runner_below_begin and runner_below_end)
    } else {

        //check if any point in the stripe below is close_enough()
        for (l=first_point_in_stripe[((list[target].t_cell)+1)];
            list[l].t_cell==(list[target].t_cell+1); l++) {
            if (close_enough(&list[target],&list[l])) {
                    runner_below_begin=l;
                    check_point=l;
                    break;
                } else check_point=Empty;
            }

            //if no point in the stripe below is close_enough(), take
                the next point in the for loop and check!
            if (check_point==Empty) continue;

            //check how much points in the stripe below are
                close_enough()
            for (m=runner_below_begin; list[m].t_cell==list[target].
                t_cell+1; m++) {
                if (close_enough(&list[target],&list[m])) {
                    runner_below_end=m;
                } else break;
            }

            //compute gamma of the points in the stripe below which are
                 close_enough() to the target cell
            for (runner=runner_below_begin; runner<=runner_below_end;
                runner++) {
                gamma=angle(&list[target],&list[runner]);
                print_output(target,runner,gamma);
            }
        }
    }
}
```

3 Results

3.1 Further processing of the output

The structure of the output, called *Output_Einstein@home_Cornell.txt*, is shown in section 2.4 *Explanation of the code*. By using the UNIX command **wc** [**options**] **<filename>**, the number of coincidences is identified as 35384 [19].

```
$ wc -l Output_Einstein@home_Cornell.txt
  35384 Output_Einstein@home_Cornell.txt
```

There are too many coincidences to go through by hand. A strategy to filter the list in order to retain the best candidates, is needed. However, "best" must be defined first.

Def.: "Best candidates" appear in a cluster of coincidences between candidates, whose data set's gap is at least one year, and the candidates come from different databases.

The filtering process is carried out with the programming language **awk** (**A**ho, **W**einberger and **K**ernighan). This script language is especially suited for the editing and assessment of structured text data [19]. The filtered file *different_year.txt* features 6163 coincidences.

```
$ cat Output_Einstein@home_Cornell.txt | awk '{a=substr($1,1,8);
b=substr($2,1,8); c=a-b; if(c<0) c*=-1; if(c>10000) print
($1,$2,$3,$4,$5,$6,$7,$8,$9,$10)}' > different_year.txt
$ wc -l different_year.txt
  6163 different_year.txt
```

Subsequently, one has to construct a list called *timelist.txt*. These list contains the date, when the candidates were recorded. It is important for the list to be unique and in order. There are 327 dates.

```
$ cat different_year.txt | awk '{a=substr($1,1,8); print a}'
> datei.txt
$ sort -n datei.txt | uniq > timelist.txt
$ wc -l timelist.txt
  327 timelist.txt
```

What is the purpose of the file *timelist.txt*?

The idea is the following: One takes a row out of *timelist.txt* with the UNIX command **grep [options] PATTERN <filename>** (**g**lobal search for a **r**egular **e**xpression and **p**rint out matched lines) and extracts all coincidences out of *different_year.txt*, which hold the date [19]. The output is sorted by the frequency of the Einstein@home-candidate (third column). In the terminal, the variable $x \in \{1,2,\ldots,327\}$ is used in various rows of *timelist.txt*. Thus, the following command line is also conducted 327 times and the results are examined for strong hints by hand.

```
$ grep 'cat timelist.txt |head -x| tail -1' different_year.txt |
sort -n -k3
```

3.2 The identification of previously discovered pulsars

Through this process, 24 clusters with "best" candidates were found. The number decreases to 17, as seven clusters are obviously RFI (radio-frequency-interference). The mains frequency in Puerto Rico being 60 Hz. So signals with approximately this frequency and their harmonics are easily identifiable as RFI. These seven clusters can be ignored in the following. Furthermore, five clusters coincide with known pulsars of the **ATNF pulsar catalogue**, because of similar frequencies (F0) and coordinates (RAJD, DecJD) [4].

1. Pulsar **B1855+02** (F0=2.40486905991; RAJD=284.43184; DecJD=2.21142)
   Einstein@home-candidates:
   - (I) 20120107.G35.69-00.42.N.b4s0g0.00000_604
     (F0=2.404473; RAJD=284.51589; DecJD=2.21519)
   - (II) 20130403.G35.43-00.40.N.b6s0g0.00000_604
     (F0=2.404473; RAJD=284.44266; DecJD=2.16591)

   Cornell-candidates:
   - (I) 20130403.G35.43-00.40.N.b6.00000_2495487
     (F0=2.404872; RAJD=284.44232; DecJD=2.16580)
   - (II) 20120107.G35.69-00.42.N.b4.00000_4012439
     (F0=2.404783; RAJD=284.51571; DecJD=2.21507)

2. Pulsar **B1913+10** (F0=2.47189777002; RAJD=288.87493; DecJD=10.16213)
   Einstein@home-candidates:
   - (I) 20120107.G44.52-00.72.N.b1s0g0.00000_614
     (F0=2.471230; RAJD=288.85003; DecJD=10.11845)
   - (II) 20130521.G44.78-00.73.C.b3s0g0.00000_614
     (F0=2.472444; RAJD=288.88819; DecJD=10.12836)
   - (III) 20100914.G44.66-00.49.C.b5s0g0.00000_614
     (F0=2.472444; RAJD=288.79737; DecJD=10.15678)

   Cornell-candidates:
   - (I) 20130521.G44.78-00.73.C.b3.00000_3040174
     (F0=2.472120; RAJD=288.88876; DecJD=10.12854)

(II) 20120107.G44.52-00.72.N.b1.00000_1549092
(F0=2.471938; RAJD=288.84997; DecJD=10.11845)

3. Pulsar **B1914+13** (F0=3.5480820974; RAJD=289.24446; DecJD=13.21389)
   Einstein@home-candidates:

   (I) 20111018.G47.54+00.36.N.b2s0g0.00000_884
   (F0=3.547842; RAJD=289.22606; DecJD=13.22228)
   (II) 20121126.G47.54+00.36.S.b1s0g0.00000_884
   (F0=3.547842; RAJD=289.26794; DecJD=13.17925)

   Cornell-candidates:

   (I) 20121126.G47.54+00.36.S.b1.00000_1531868
   (F0=3.547818; RAJD=289.26777; DecJD=13.17919)
   (II) 20111018.G47.54+00.36.N.b2.00000_2436307
   (F0=3.547776; RAJD=289.22594; DecJD=13.22205)

4. Pulsar **B1929+20** (F0=3.72831885206; RAJD=293.03343; DecJD=20.34623)
   Einstein@home-candidate:

   (I) 20100902.G55.46+00.47.C.b1s0g0.00000_935
   (F0=3.729907; RAJD=293.10418; DecJD=20.25694)

   Cornell-candidates:

   (I) 20130529.G55.46+00.47.C.b1.00000—_3096604
   (F0=3.728494; RAJD=293.10601; DecJD=20.25631)
   (II) 20130617.G55.46+00.47.N.b2.00000_3253248
   (F0=3.728512; RAJD=293.03926; DecJD=20.25138)
   (III) 20130802.G55.60+00.70.C.b4.00000_3645682
   (F0=3.728226; RAJD=293.00701; DecJD=20.30243)

5. Pulsar **J1935+2025** (F0=12.4815688643; RAJD=293.92475; DecJD=20.42781)
   Einstein@home-candidates:

   (I) 20090630.G56.12-00.09.N.b6s0g0.00000_3115
   (F0=12.479955; RAJD=293.94924; DecJD=20.44774)
   (II) 20130522.G56.11-00.23.N.b2s0g0.00000_3115
   (F0=12.484811; RAJD=294.01839; DecJD=20.47810)

   Cornell-candidates:

   (I) 20130522.G56.11-00.23.N.b2.00000_2841259
   (F0=12.480759; RAJD=294.01902; DecJD=20.47805)
   (II) 20090630.G56.12-00.09.N.b6.00000_2143915
   (F0=12.480714; RAJD=293.94918; DecJD=20.44768)

3.3 The remain clusters with “best” candidates

The 12 clusters with “best” candidates, which remained after the complex filter process, obtain a special attention.

1. **Einstein@home**-candidates:

   (I) 20120504.G60.03+00.65.N.b1s0g0.00000_354
   (F0=1.437100; RAJD=295.41246; DecJD=24.20628)
   (II) 20110421.G60.29+00.20.C.b4s0g0.00000_355
   (F0=1.437100; RAJD=295.91586; DecJD=24.33944)

   Cornell-candidates:

   (I) 20110421.G60.03+00.65.C.b0.00000_2361007
   (F0=1.434261; RAJD=295.36622; DecJD=24.24473)
   (II) 20120504.G60.29+00.20.N.b5.00000_3516881
   (F0=1.445656; RAJD=295.85380; DecJD=24.33571)

2. **Einstein@home**-candidates:

   (I) 20110220.G176.20-04.39.S.b5s0g0.00000_3835
   (F0=15.338378; RAJD=79.68136; DecJD=29.69791)

   Cornell-candidates:

   (I) 20121225.G176.20-04.39.N.b6.00000_1088251
   (F0=15.345376; RAJD=79.75286; DecJD=29.71327)
   (II) 20130930.G176.20-04.39.N.b6.00000_4420421
   (F0=15.332508; RAJD=79.75487; DecJD=29.71126)

3. **Einstein@home**-candidates:

   (I) 20100827.G57.81+00.44.C.b0s0g0.00000_3984
   (F0=15.975605; RAJD=294.37334; DecJD=22.20546)
   (II) 20100827.G57.81+00.44.C.b5s0g0.00000_3995
   (F0=15.985315; RAJD=294.48191; DecJD=22.16993)

   Cornell-candidates:

   (I) 20130407.G57.81+00.44.N.b3.00000_2251532
   (F0=15.970984; RAJD=294.30802; DecJD=22.20087)
   (II) 20121116.G57.81+00.44.S.b3.00000_1842003
   (F0=15.987504; RAJD=294.43493; DecJD=22.20803)

4. **Einstein@home**-candidates:

   (I) 20090730.G41.27+00.27.C.b0s0g0.00000_4084
   (F0=16.348232; RAJD=286.43627; DecJD=7.54020)
   (II) 20110422.G41.44+00.29.S.b4s0g0.00000_4075
   (F0=16.319102; RAJD=286.50141; DecJD=7.54713)

Cornell-candidates:

   (I) 20110422.G41.44+00.29.S.b3.00000_2161613
   (F0=16.340403; RAJD=286.40940; DecJD=7.58741)

   (II) 20090730.G41.27+00.27.C.b0.00000_2145378
   (F0=16.309424; RAJD=286.43633; DecJD=7.54020)

5. **Einstein@home**-candidates:

   (I) 20110122.G175.19-03.85.S.b5s0g0.00000_6325
   (F0=25.332538; RAJD=79.52356; DecJD=30.82446)

   (II) 20110122.G175.34-03.64.S.b1s0g0.00000_6325
   (F0=25.332538; RAJD=79.94646; DecJD=30.69543)

   Cornell-candidates:

   (I) 20121010.G175.04-04.07.C.b2.00000_508333
   (F0=25.336332; RAJD=79.47280; DecJD=30.82223)

   (II) 20120222.G175.34-03.64.S.b1.00000_767231
   (F0=25.339214; RAJD=79.91426; DecJD=30.69285)

6. **Einstein@home**-candidates:

   (I) 20101122.G47.30+00.83.C.b5s0g0.00000_8544
   (F0=34.223383; RAJD=288.85948; DecJD=13.11680)

   Cornell-candidates:

   (I) 20121126.G47.42+00.59.S.b2.00000_1587866
   (F0=34.219830; RAJD=288.91340; DecJD=13.11388)

   (II) 20130625.G47.30+00.83.N.b5.00000_3229378
   (F0=34.224986; RAJD=288.86750; DecJD=13.16023)

7. **Einstein@home**-candidates:

   (I) 20131005.G64.01-01.62.C.b1s0g0.00000_9465
   (F0=37.927801; RAJD=299.79169; DecJD=26.41005)

   (II) 20140531.G63.88-01.85.C.b4s0g0.00000_9464
   (F0=37.922946; RAJD=299.88880; DecJD=26.38478)

   Cornell-candidates:

   (I) 20100629.G63.93-01.74.C.b0.00000_132383
   (F0=37.924389; RAJD=299.83580; DecJD=26.38261)

8. **Einstein@home**-candidates:

   (I) 20111207.G177.41-02.20.S.b0s0g0.00000_17695
   (F0=70.896149; RAJD=82.61960; DecJD=29.88246)

   (II) 20121225.G177.52-02.44.N.b0s0g0.00000_17695
   (F0=70.899790; RAJD=82.55875; DecJD=29.78214)

Cornell-candidates:

(I) 20121225.G177.52-02.44.N.b3.00000_1084961
(F0=70.898280; RAJD=82.65214; DecJD=29.85066)
(II) 20111207.G177.41-02.20.S.b6.00000_1927435
(F0=70.897675; RAJD=82.52592; DecJD=29.81285)

9. **Einstein@home**-candidates:

(I) 20111205.G176.53-03.51.N.b1s0g0.00000_49435
(F0=198.236119; RAJD=80.85930; DecJD=29.89690)

Cornell-candidates:

(I) 20121228.G176.65-03.74.N.b3.00000_1466082
(F0=198.244567; RAJD=80.84944; DecJD=29.85049)
(II) 20130930.G176.65-03.74.N.b3.00000_4421028
(F0=198.230655; RAJD=80.84366; DecJD=29.85284)

10. **Einstein@home**-candidates:

(I) 20110121.G192.72-04.29.N.b3s0g0.00000_75845
(F0=303.819310; RAJD=89.34933; DecJD=15.80936)
(II) 20121015.G192.59-04.52.C.b2s0g0.00000_75743
(F0=303.474600; RAJD=89.05878; DecJD=15.84866)

Cornell-candidates:

(I) 20121015.G192.72-04.29.C.b0.00000_3402545
(F0=303.829308; RAJD=89.43270; DecJD=15.82185)
(II) 20110121.G192.59-04.52.N.b3.00000_206840
(F0=303.464671; RAJD=89.07122; DecJD=15.80947)

11. **Einstein@home**-candidates:

(I) 20110111.G192.19-04.30.S.b5s0g0.00000_75998
(F0=304.235632; RAJD=89.19876; DecJD=16.17336)
(II) 20121007.G192.97-03.39.N.b6s0g0.00000_75992
(F0=304.213784; RAJD=90.48098; DecJD=16.16081)

Cornell-candidates:

(I) 20120921.G192.19-04.30.S.b0.00000_538313
(F0=304.237955; RAJD=89.13304; DecJD=16.22080)
(II) 20110111.G192.84-03.16.S.b4.00000_191654
(F0=304.222758; RAJD=90.51358; DecJD=16.10689)

12. **Einstein@home**-candidates:

(I) 20110111.G192.84-03.16.S.b2s0g0.00000_88495
(F0=354.492881; RAJD=90.37997; DecJD=16.24927)
(II) 20120917.G192.84-03.16.N.b6s0g0.00000_88558
(F0=354.927410; RAJD=90.45124; DecJD=16.27087)

Cornell-candidates:

(I) 20120917.G192.84-03.16.N.b6.00000_629420
(F0=354.489777; RAJD=90.45101; DecJD=16.27087)

(II) 20110111.G192.84-03.16.S.b2.00000_194595
(F0=354.924103; RAJD=90.40695; DecJD=16.24887)

They were given to collaborators of Allen to have a closer look at the structure of their signal. The result of this further investigation is, that all candidates of the 12 clusters are different types of RFIs. Since new pulsars haven't been discovered, this project seems to be a good start for further studies, which may follow.

4 References

[1] https://www.cfa.harvard.edu/~dfabricant/huchra/ay145/constants.html (2014/11/22)

[2] Bülent Kiziltan, Athanasios Kottas and Stephen E. Thorsett: The neutron star mass distribution (2010/11/18)

[3] http://heasarc.gsfc.nasa.gov/docs/xte/learning_center/ASM/ns.html (2014/11/19)

[4] ATNF pulsar catalogue version 1.51: http://www.atnf.csiro.au/people/pulsar/psrcat/ (2014/11/13)

[5] http://www.physics.mcgill.ca/~pulsar/scematic.jpg (2014/11/13)

[6] Duncan Lorimer and Michael Kramer: Handbook of Pulsar Astronomy (ISBN 978-0-521-53534-2)

[7] Antony Hewish and Jocelyn Bell: Observation of a Rapidly Pulsating Radio Source (1968/02/09)

[8] Russell Hulse and Joseph Taylor: Discovery of a pulsar in a binary system (1975/01/15)

[9] Torsten Fließbach: Allgemeine Relativitätstheorie (ISBN 978-3-8274-3032-8)

[10] Shrinivas Kulkarni and Donald Backer: A millisecond pulsar (1982/12/16)

[11] Andrew Lyne: The discovery of a millisecond pulsar in the globular cluster M28 (1987/07/30)

[12] Aleksander Wolszczan and Dale Frail: A planetary system around the millisecond pulsar PSR1257+12 (1992/09/01)

[13] Stephen Thorsett: The Triple Pulsar System PSR B1620-26 in M4 (1999/03/15)

[14] Marta Burgay: An increased estimate of the merger rate of double neutron stars from observations of a highly relativistic system (2003/12/04)

[15] Bruce Allen: The Einstein@home search for radio pulsars and PSR J2007+2722 Discovery (2013/07/29)

[16] http://einsteinathome.org/ (2014/11/17)

[17] Jürgen Wolf: C von A bis Z (ISBN 978-3-8362-1411-7)

[18] Gerd Fischer: Lineare Algebra (ISBN 978-3-8348-0996-4)

[19] Arnold Willemer: UNIX (ISBN 978-3-8362-1071-3)

Appendix

The complete code:

```
//
//  Bachelorarbeit.c
//
//  Created by Tjark Miener and Bruce Allen on 29.07.14.
//  Copyright (c) 2014 Tjark Miener and Bruce Allen. All rights reserved.
//

#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <math.h>
#include <stdbool.h>
#include <string.h>

#define Allocate_blocksize 1024
#define Stripestheta 3001
#define Stripesphi (2*Stripestheta)
#define Gamma_max (2*M_PI/Stripesphi)
#define Empty -12345
#define End_of_first_list 1265589
#define Minimum_frequency 0.01
#define Maximum_scan 64

int num_in_stripe[Stripestheta];

//The structure skypoint have two doubles for the position and two ints for
//the cell the skypoint is in. Every skypoint has a number (int) and a
    frequence (double).

struct Skypoint{
    int number;
    char id[Maximum_scan];

    double phi;       //0,...,2*PI
    double theta;     //0,...,PI

    int p_cell;
    int t_cell;

    double frequency;
    float significance;
};
struct Skypoint *list=NULL;

//function to compute the cells for each skypoint

void cellcomputing(struct Skypoint *a){

    if ((a->theta)<M_PI) {
        a->t_cell= (int)((a->theta)*Stripestheta/M_PI);
    } else {
        a->t_cell=Stripestheta-1;
    }
    a->p_cell=floor((a->phi)*(num_in_stripe[a->t_cell]-1)/(2.0*M_PI));
    return;
}

//calculate gamma of the skypoints

double angle(struct Skypoint *a,struct Skypoint *b) {
    double x = sin(a->theta)*sin(b->theta)*cos((a->phi)-(b->phi))+cos(a->
        theta)*cos(b->theta);
    if (x > 1.0) {
        x = 1.0;
    }
    return acos(x)*(180.0/M_PI);
}

//the function return if we need to check the cells
//the runnercell is always below the target cell

int close_enough(struct Skypoint *a, struct Skypoint *b){

    struct Skypoint targetcell_left_bottom;
    struct Skypoint targetcell_right_bottom;
    struct Skypoint runnercell_left_top;
    struct Skypoint runnercell_right_top;

    int p1=a->p_cell;
    int nextp1=(a->p_cell)+1;
    int nextt1=(a->t_cell)+1;

    int p2=b->p_cell;
    int t2=b->t_cell;
    int nextp2=(b->p_cell)+1;

    targetcell_right_bottom.theta=targetcell_left_bottom.theta=nextt1*M_PI/
        Stripestheta;
    targetcell_left_bottom.phi=2*M_PI*p1/num_in_stripe[a->t_cell];
    targetcell_right_bottom.phi=2*M_PI*nextp1/num_in_stripe[a->t_cell];

    runnercell_right_top.theta=runnercell_left_top.theta=t2*M_PI/
        Stripestheta;
    runnercell_left_top.phi=2*M_PI*p2/num_in_stripe[b->t_cell];
    runnercell_right_top.phi=2*M_PI*nextp2/num_in_stripe[b->t_cell];

    double gamma_1=angle(&targetcell_left_bottom,&runnercell_left_top);
    double gamma_2=angle(&targetcell_left_bottom,&runnercell_right_top);
    double gamma_3=angle(&targetcell_right_bottom,&runnercell_left_top);
    double gamma_4=angle(&targetcell_right_bottom,&runnercell_right_top);

    if (gamma_1<Gamma_max || gamma_2<Gamma_max || gamma_3<Gamma_max ||
        gamma_4<Gamma_max) {
```

```
        return 1;
    } else {
        return 0;
    }
}

//dynamic allocation function

void my_Allocate(int a) {
    list=realloc(list,a*sizeof(struct Skypoint));
    if (!list) {
        printf("Error! No more RAM available, skypoints_max = %d!\n", a);
        exit(1);
    }
}

//comparefunktion for the qsort() of <stdlib.h>

int compare(const void *a, const void *b){
    const struct Skypoint *elementa = a;
    const struct Skypoint *elementb = b;

    if (elementa->t_cell > elementb->t_cell) return 1;
    if (elementa->t_cell < elementb->t_cell) return -1;

    if (elementa->p_cell > elementb->p_cell) return 1;
    if (elementa->p_cell < elementb->p_cell) return -1;

    return 0;
}

//print out function

void print_output(int i,int j,double g){

    char string1[Maximum_scan];
    char string2[Maximum_scan];

    strncpy(string1, list[i].id, Maximum_scan);
    strncpy(string2, list[j].id, Maximum_scan);

    char *ptr1;
    char *ptr2;

    if (list[i].significance==0.0 && list[j].significance!=0.0) {

        if (fabs(list[i].frequency-list[j].frequency) < Minimum_frequency){

            ptr1 = strtok(string1,"b");
            ptr2 = strtok(string2,"b");

            int result = strcmp (ptr1,ptr2);
            if (result != 0) {
                printf("%s %s %f %f %f %f %f %f %f %f\n",
                                  list[j].id,list[i].id,list[j].frequency,list[i].
                                       frequency,
                                  list[j].phi,list[j].theta,list[i].phi,list[i].theta,
                                  list[j].significance,g);
                   }
             }

       } else if (list[j].significance==0.0 && list[i].significance!=0.0){

             if (fabs(list[i].frequency-list[j].frequency) < Minimum_frequency){

                   ptr1 = strtok(string1,"b");
                   ptr2 = strtok(string2,"b");

                   int result = strcmp (ptr1,ptr2);
                   if (result != 0) {
                         printf("%s %s %f %f %f %f %f %f %f %f\n",
                                  list[i].id,list[j].id,list[i].frequency,list[j].
                                       frequency,
                                  list[i].phi,list[i].theta,list[j].phi,list[j].theta,
                                  list[i].significance,g);
                   }
             }
       }
       return;
}

//main function

int main (int argc, char** argv) {

       int i,j,k,l,m;

       int skypoints_read=0;
       int skypoints_allocated=0;

       int target;
       int runner,runner_below_begin,runner_below_end;
       int check_point;
       int first_point_in_stripe[Stripestheta+1];
       int last_stripe;

       double gamma;

       //check if Stripestheta is odd
       if (Stripestheta%2==0) {
             printf("Error! Stripestheta = %d is not odd!\n", Stripestheta);
             exit(2);
       }

       //initialisation of num_in_stripe[]

       for (i=0; i<Stripestheta; i++) {
             num_in_stripe[i]=ceil(Stripesphi*sin(M_PI*(i+0.5)/Stripestheta));
```

```
    }

    //check if arguments in the command line is right

    if (argc!=2) {
        printf("Error! Missing command line arguments, only found %d not 1\
           n", argc-1);
        exit(3);
    }

    //open the .txt file

    FILE *input_candidate = fopen(argv[1],"r");

    if (!input_candidate) {
        printf("Error! File %s not found\n", argv[1]);
        exit(4);
    }

    while (true) {

        //reallocating memory if necessary
        if (skypoints_read==skypoints_allocated) {
            skypoints_allocated += Allocate_blocksize;
            my_Allocate(skypoints_allocated);
        }

        if (skypoints_read < End_of_first_list) {

            //scan the elements of the first part of the file
            if (6==(i=fscanf(input_candidate, "%d %s %lf %lf %lf %f",&list[
                skypoints_read].number, list[skypoints_read].id,&list[
                skypoints_read].phi,&list[skypoints_read].theta,&list[
                skypoints_read].frequency,&list[skypoints_read].
                significance))) {

                list[skypoints_read].phi=(list[skypoints_read].phi*15.0)
                    *2.0*M_PI/360.0;
                list[skypoints_read].theta=(90.0-list[skypoints_read].theta
                    )*2.0*M_PI/360.0;

                //check the data ranges, exit if error
                if (list[skypoints_read].phi < 0 || list[skypoints_read].
                    phi > 2*M_PI || list[skypoints_read].theta < 0 || list[
                    skypoints_read].theta > M_PI) {
                    printf("Error! Problem reading %s at line %d. Point (%f
                        ,%f) isn't in the data range!\n", argv[1],
                        skypoints_read, list[skypoints_read].phi, list[
                        skypoints_read].theta);
                    exit(5);
                }

                cellcomputing(list+skypoints_read);
                skypoints_read++;
            } else break;

        } else {

            //scan the elements of the secound part of the file
            if (5==(i=fscanf(input_candidate, "%d %s %lf %lf %lf",&list[
                skypoints_read].number, list[skypoints_read].id,&list[
                skypoints_read].phi,&list[skypoints_read].theta,&list[
                skypoints_read].frequency))) {

                list[skypoints_read].phi=list[skypoints_read].phi*2.0*M_PI
                    /360.0;
                list[skypoints_read].theta=(90.0-list[skypoints_read].theta
                    )*2.0*M_PI/360.0;
                list[skypoints_read].significance=0.0;

                //check the data ranges, exit if error
                if (list[skypoints_read].phi < 0 || list[skypoints_read].
                    phi > 2*M_PI || list[skypoints_read].theta < 0 || list[
                    skypoints_read].theta > M_PI) {
                    printf("Error! Problem reading %s at line %d. Point (%f
                        ,%f) isn't in the data range!\n", argv[1],
                        skypoints_read, list[skypoints_read].phi, list[
                        skypoints_read].theta);
                    exit(5);
                }

                cellcomputing(list+skypoints_read);
                skypoints_read++;

            } else break;

        }

    }

    if (i!=EOF) {
        fprintf(stderr, "Error! Problem reading %s at line %d (number:%d).
            fscanf() returned %d\n", argv[1], skypoints_read, list[
            skypoints_read].number,i);
        exit(6);
    }


    //adding shadow points

    k=j=0;
    for (i=0; i<skypoints_read; i++) {

        //reallocating memory if necessary
        if ((skypoints_read+k)==skypoints_allocated && j==1) {
            skypoints_allocated += Allocate_blocksize;
            my_Allocate(skypoints_allocated);
        }

        //adding shadow points on the right
        if (list[i].p_cell < 2){

            list[skypoints_read+k].phi=((list[i].phi)+2.0*M_PI);
            list[skypoints_read+k].theta=list[i].theta;
            list[skypoints_read+k].t_cell=list[i].t_cell;
            list[skypoints_read+k].p_cell=list[i].p_cell+num_in_stripe[list
                [i].t_cell];
            list[skypoints_read+k].significance=list[i].significance;
            list[skypoints_read+k].frequency=list[i].frequency;
            list[skypoints_read+k].number=list[i].number;
            strncpy(list[skypoints_read+k].id, list[i].id, Maximum_scan);

            k++;
            j=1;

        //adding shadow points on the left
        } else if (list[i].p_cell > num_in_stripe[list[i].t_cell]-3) {

            list[skypoints_read+k].phi=((list[i].phi)-2.0*M_PI);
            list[skypoints_read+k].theta=list[i].theta;
            list[skypoints_read+k].t_cell=list[i].t_cell;
            list[skypoints_read+k].p_cell=list[i].p_cell-num_in_stripe[list
                [i].t_cell];
            list[skypoints_read+k].significance=list[i].significance;
            list[skypoints_read+k].frequency=list[i].frequency;
            list[skypoints_read+k].number=list[i].number;
            strncpy(list[skypoints_read+k].id, list[i].id, Maximum_scan);

            k++;
            j=1;
        } else {
            j=0;
        }

    }

    skypoints_read += k;

    //sorting the list with the shadow points

    qsort (list,skypoints_read,sizeof(struct Skypoint), compare);

    //find the first point in the stripe

    for (i=0; i<=Stripestheta; i++) {
        first_point_in_stripe[i]=Empty;
    }

    last_stripe=first_point_in_stripe[list[1].t_cell]=0;
    for (i=1; i<skypoints_read; i++) {
        if (list[i].t_cell != last_stripe) {
            last_stripe=list[i].t_cell;
            first_point_in_stripe[list[i].t_cell]=i;
        }
    }

    //compute gamma of the points in the same cell and in the next cell

    for (target=0; target<skypoints_read; target++) {

        //only take the non shadow point
        if (list[target].phi>=0 && list[target].phi<=2.0*M_PI) {

            for (runner=target+1; runner<skypoints_read; runner++) {

                //target and runner are in the same cell
                if (list[target].p_cell==list[runner].p_cell && list[target
                    ].t_cell==list[runner].t_cell) {
                    gamma=angle(&list[target],&list[runner]);
                    print_output(target,runner,gamma);

                //runner are in the next cell
                } else if(list[target].p_cell==((list[runner].p_cell)-1) &&
                     list[target].t_cell==list[runner].t_cell) {
                    gamma=angle(&list[target],&list[runner]);
                    print_output(target,runner,gamma);

                //runner are too far away -> break
                } else break;
            }
        }
    }


    //initialization of the first point

    for (i=0; i<skypoints_read; i++) {

        //only take the non shadow point
        if (list[i].phi>=0 && list[i].phi<=2.0*M_PI) {

            //if the stripe below is empty, take the next point in the For-
                loop and check!
            if (first_point_in_stripe[((list[i].t_cell)+1)]==Empty)
                continue;

            //check if any point in the stripe below is close_enough()
            for (j=first_point_in_stripe[list[i].t_cell+1]; list[j].t_cell
                ==(list[i].t_cell+1); j++) {
                if (close_enough(&list[i],&list[j])) {
                    runner_below_begin=j;
                    check_point=j;
                    break;
                } else check_point=Empty;
            }
            //if no point in the stripe below is close_enough(), take the
                next point in the For-loop and check
            if (check_point==Empty) continue;

            //check how much points in the stripe below are close_enough()
            for (k=runner_below_begin; list[k].t_cell==(list[i].t_cell+1);
                k++) {
                if (close_enough(&list[i],&list[k])) {
                    runner_below_end=k;
                } else break;
            }

            break;
        }
    }

    //if we can't initial the first point, it means that for every point
        there is no point in the stripe below close_enough()
    //we can skip the following part and jump with goto to end of the
        programm
    if (i==skypoints_read) goto end_of_program;

    //compute gamma of the points in the stripe below which are
        close_enough() to the target cell

    for (runner=runner_below_begin; runner<=runner_below_end; runner++) {
        gamma=angle(&list[i],&list[runner]);
        print_output(i,runner,gamma);
    }

    //after initialize the first point, compute the other points

    for (target=i+1; target<skypoints_read; target++) {

        //only take the non shadow point
        if (list[target].phi>=0 && list[target].phi<=2.0*M_PI) {

            //if the stripe below is empty, take the next point in the For-
                loop and check
            if (first_point_in_stripe[((list[target].t_cell)+1)]==Empty)
                continue;

            //the target stays in the same cell (no update of the
                runner_below_begin and runner_below_end)
            if (list[target].p_cell==list[target-1].p_cell && list[target].
                t_cell==list[target-1].t_cell) {

                //if no point in the stripe below is close_enough(), take
                    the next point in the For-loop and check!
                if (check_point==Empty) continue;

                //compute gamma of the points in the cell below which are
                    close_enough() to the target cell
                for (runner=runner_below_begin; runner<=runner_below_end;
                    runner++) {
                    gamma=angle(&list[target],&list[runner]);
                    print_output(target,runner,gamma);
                }

            //the target moves to a different cell (update of the
                runner_below_begin and runner_below_end)
            } else if (list[target].p_cell!=list[target-1].p_cell && list[
                target].t_cell==list[target-1].t_cell) {

                //check if any point in the stripe below is close_enough()
                for (j=first_point_in_stripe[((list[target].t_cell)+1)];
                    list[j].t_cell==list[target].t_cell+1; j++) {
                    if (close_enough(&list[target],&list[j])) {
                        runner_below_begin=j;
                        check_point=j;
                        break;
                    } else check_point=Empty;
                }

                //if no point in the stripe below is close_enough(), take
                    the next point in the For-loop and check!
                if (check_point==Empty) continue;

                //check how much points in the stripe below are
                    close_enough()
                for (k=runner_below_begin; list[k].t_cell==(list[target].
                    t_cell+1); k++) {
                    if (close_enough(&list[target],&list[k])) {
                        runner_below_end=k;
                    } else break;
                }

                //compute gamma of the points in the stripe below which are
                     close_enough() to the target cell
                for (runner=runner_below_begin; runner<=runner_below_end;
                    runner++) {
                    gamma=angle(&list[target],&list[runner]);
                    print_output(target,runner,gamma);
                }

            //the target moves to a different stripe (update of the
                runner_below_begin and runner_below_end)
            } else {

                //check if any point in the stripe below is close_enough()
                for (l=first_point_in_stripe[((list[target].t_cell)+1)];
                    list[l].t_cell==(list[target].t_cell+1); l++) {
                    if (close_enough(&list[target],&list[l])) {
                        runner_below_begin=l;
                        check_point=l;
                        break;
                    } else check_point=Empty;
                }

                //if no point in the stripe below is close_enough(), take
                    the next point in the For-loop and check!
                if (check_point==Empty) continue;

                //check how much points in the stripe below are
                    close_enough()
                for (m=runner_below_begin; list[m].t_cell==list[target].
                    t_cell+1; m++) {
                    if (close_enough(&list[target],&list[m])) {
                        runner_below_end=m;
                    } else break;
                }

                //compute gamma of the points in the stripe below which are
                     close_enough() to the target cell
                for (runner=runner_below_begin; runner<=runner_below_end;
                    runner++) {
                    gamma=angle(&list[target],&list[runner]);
                    print_output(target,runner,gamma);
                }
            }
        }
    }

    end_of_program : printf("There are no more points to compare!\n");

    free(list);
    fclose(input_candidate);

    return 0;
}
```