MA6252 Topics in Applied Mathematics II

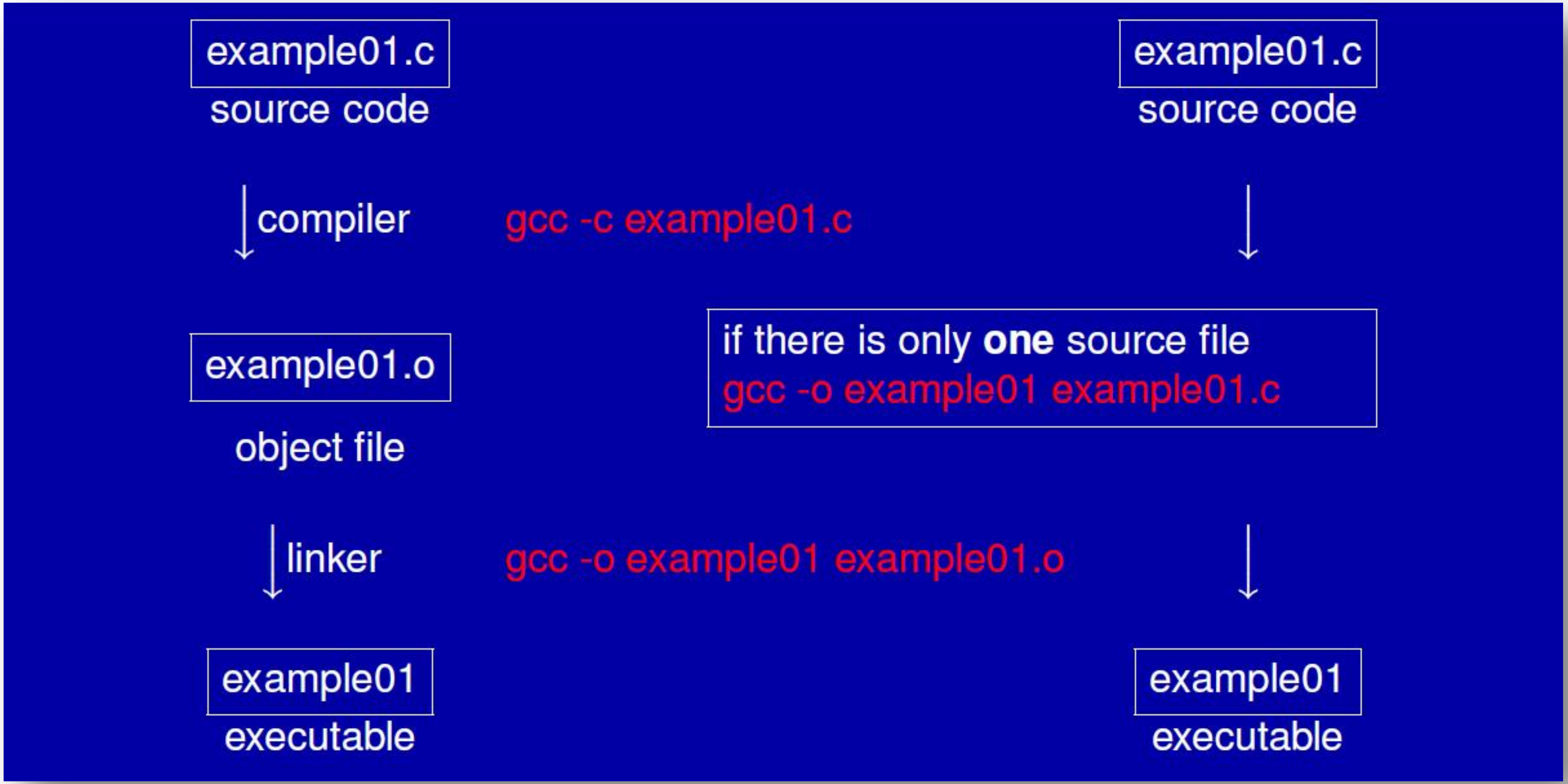
C crash course and red-black Gauss-Seidel

Example

```c
/* This programm will just say hello */

#include <stdio.h>

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

        printf("Hello,␣students\n")
        return 0;
}
```

- But ”hello.c” is not an executable file, but a text file.

From compiler to executable



```c
/* This programm will just say hello */

#include <stdio.h>

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

        printf("Hello,␣students\n")
        return 0;
}
```

- /* This is a comment */
- #include <stdio.h> is necessary for the usage of printf(...)
- int main(int nargs, char** pargs): the main function, here starts the running programm
- Functions are subprograms. C programs consist from one or more functions
- printf("Hello ..."); : output, using the function **printf**.
- return 0; : return value 0 at the end of function **main**.

Smallest possible program

```c
/* This program does only return 0*/

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

        /* This is an empty statement */
        ;
        /* A code block with an empty statment */
        { ; }
        /* leave function and return 0  */
        return 0;
}
```

- Every program starts entering the function **main**. So this function must be present in every C program.
- { and } embrace, what should be done in this function **main**. In general { and } delimit a **code block**.
- All statements inside this and every other function are processed sequentially in the listed order. Every statement in C is terminated by a semicolon. ; is the empty statement, so ;{;;{}}; is legal code, but useless.

Variable declarations

- Each variable must be declared before we can use them.
- Optionally we also can initialize the variable with a value.
- Declaring a variable:

  <**storage type**> <**variable name**> [= **value**];

storage type	integers:	((un)signed) char
		((un)signed) ([long,short]) int
	floating point:	float
		double
variable name	Arbitrary length (upper bound ist compiler dependent)	
	Case sensitive. May contain numbers, but not start with one.	

Variables example I

```c
#include <stdio.h>
/*
 * Declaration of vaiables.
 * Variables of the same type can be seperated by a comma.
*/
int main(int argc, char **argv){

        int a=1,
        b=a+2;
        double c=0.1;
        double d=1e-1;
        char e='e';

        /* Just write some output */
        printf("a = %d\n",a);
        printf("b = %d\n",b);
        printf("c = %e\n",c);
        printf("d = %e\n",d);
        printf("e = %c or\ne = %d\n",e,e);
        return 0;
}
```

Variables example II

```c
#include <stdio.h>

/***********HEAD SECTION***********/
int main(int argc,char **argv){

        /*******DECLARATION SECTION********/

        /* The money in our bank account*/
        double bank_account   = 1000.0;
        /* The money we want to withdraw*/
        double withdrawal     = 50.0;

        /********STATEMENT SECTION*********/

        printf("Account before withdrawal: %f\n",bank_account);
        printf("Withdrawal: %f\n",withdrawal);

        double b=1.0;

        /*withdraw the money*/
        bank_account=bank_account-withdrawal; /* 950 = 1000 - 50*/

        printf("Account after withdrawal: %f\n",bank_account);
}
```

Variables – storage sizes

type	typical size [*byte*]	range
(signed) char	1:□	$[-128, 127]$
unsigned char	1:□	$[0, 255]$
(signed) short int	2:□□	$[-32768, -32767]$
unsigned short int	2:□□	$[0, 65535]$
(signed) (long) int	4:□□□□	$[-2^{32}, 2^{32}-1]$

typ	precision[*digits*]	size[*byte*]	$\pm$	exp	m	range
float	6 - 7	4:□□□□	1 Bit	8 Bit	24 Bit	$\pm[10^{-37}, 10^{+37}]$
double	15 - 16	8:□□□□ □□□□	1 Bit	11 Bit	53 Bit	$\pm[10^{-308}, 10^{+308}]$

So we can see, that assigning the value of an **int** to a **char** will only succeed, if we know for sure, that the value does not exceed the range of the storage type **char**. The other way round there will never be a problem.

Keep in mind, that **float**, **double** have a limited precision.

Variable example III

```c
#include <stdio.h>
/* Global variables */
int a;
int b=5;

int main(){

	/* uninitialized */
	int c;

	/* negative unsinged variable */
	unsigned int d = -1;
	float e = 123.456789123456789123456789;
	double f = 123.456789123456789123456789;

	/* Just write some output */
	printf("a = %d\n",a); /*a = 0*/
	printf("b = %d\n",b); /*b = 5*/
	printf("c = %d\n",c); /*c = undefined*/
	printf("d = %u\n",d); /*d = 4294967295*/
	printf("e = %.20f\n",e); /*e = 123.45678...*/
	printf("f = %.20f\n",f); /*f = 123.45678912345678...*/
	return 0;
}
```

Operators

- Operator take input values (unary, binary, ternary) and computes from these a return value.
- If an arithmetic operator processes two numbers of different storage type, the one with the lower accuracy is converted internally to the accuracy of the higher one.
- The return value of the operation has the higher accuracy type.
- Examples:
  - +, -, /,* : common arithmetic operators
  - = : assignemt operator
  - % : modulo (division without rest)

```c
#include <stdio.h>
int main(){
	int  a = 1000;
	char b = 2;
	int res = ( a + b ) * 2 + 2 * -2;               /* res = 2000 */
	printf("Result: %d\n",res);
	return 0;
}
```

Operator – evaluation order

typ	prio	operators	associativity
	15	[] . $*$ ()	left associative
unary	14	! $\sim$ ++ - - & * (typ) sizeof + -	right associative
	13	* / %	left associative
binary	12	+ -	left associative
bitshift	11	$\ll$ $\gg$	left associative
	10	$< \leq > \geq$	left associative
comparision	9	== !=	left associative
	8	&	left associative
	7	^	left associative
bitwise	6	$\vert$	left associative
	5	&&	left associative
logical	4	$\Vert$	left associative
Conditional	3	? :	right associative
assignment	2	= += -= *= /= %= &= ^= $\ll$= $\gg$=	right associative
sequence	1	,	left assosiative

Operators – example

```c
#include <stdio.h>

/* What does this program return ? */
int main(){

        int ReturnValue     = 4;
        double Buffy        = 1/2,
               MickeyMouse = ReturnValue * Buffy;
        ReturnValue = ReturnValue * MickeyMouse;
        ReturnValue = ReturnValue + 0.5;

        printf("ReturnValue = %d\n",ReturnValue);

        return ReturnValue;
}
```

To build the program and execute it we type:

- gcc Operator1.c –o Operator1
- ./Operator1

Operators – example

- Why does the program return zero?

```c
#include <stdio.h>

/* What does this program return ? */
int main(){

        int ReturnValue     = 4;                        /* ReturnValue = 4   */
        double Buffy        = 1/2,                      /* Buffy       = 0.0 */
               MickeyMouse = ReturnValue * Buffy;       /* MickeyMouse = 0.0 */
        ReturnValue = ReturnValue * MickeyMouse;        /* ReturnValue = 0   */
        ReturnValue = ReturnValue + 0.5;                /* ReturnValue = 0   */

        printf("ReturnValue = %d\n",ReturnValue);

        return ReturnValue;
}
```

Relational and local operators

operator	sign in C	return value
AND	&&	$(a \,\&\&\, b) \neq 0 \quad \Leftrightarrow \quad a \neq 0 \wedge b \neq 0$
OR	||	$(a \,\|\|\, b) \neq 0 \quad \Leftrightarrow \quad a \neq 0 \vee b \neq 0$
NOT	!	$(\,!\,a) \neq 0 \quad \Leftrightarrow \quad a = 0$
	==	$(a == b) \neq 0 \quad \Leftrightarrow \quad a = b$
	!=	$(a \,!\!= b) \neq 0 \quad \Leftrightarrow \quad a \neq b$

- The binding of the logical operators is quite weak. Only the assignment operator and the conditional operator (we don't know it yet) bind even more weakly. So that for example:

$\boxed{i \geq 5 \;\&\&\; i!=10 \;||\; i}$ will be evaluated like $\boxed{(\,(i \geq 5 \;\&\&\; i \,!\!= 10) \;\;||\;\; i)}$

Example

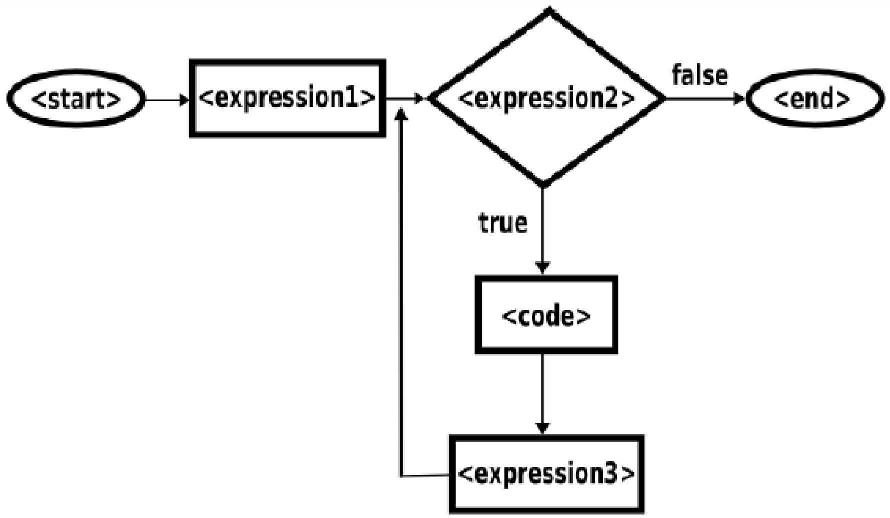
```
#include <stdio.h>
int main(){

        int a = 5,
        b = 0;
        int result;

        /* logical and */
        result = a&&b; /* result = 0 */ printf("a&&b = %d\n\n",result);

        /* logical or */
        result = a||b; /* result = 1 */ printf("a||b = %d\n\n",result);

        /* logical not */
        result = !b;   /* result = 1 */ printf("!b   = %d\n\n",result);

        /* equality */
        result = a==b; /* result = 0 */ printf("a==b = %d\n\n",result);

        /* inequality */
        result = a!=b; /* result = 1 */ printf("a!=b = %d\n\n",result);

        return 0;
}
```

The if-then-else statement

```c
#include <stdio.h>
int main(){
        int a=1, b=2, c=3, max;
        /* calculate max of a ,b and c */
        /* Is a larger than b and c*/
        if(a > b && a > c){
                /* Yes*/
                max=a;
                printf("a is larger than b,c\n");
        }
        else /*b,c are larger than a*/
                /*Is b larger than c*/
                if( b>c ){
                        /* Yes*/
                        max=b;
                        printf("b is larger than a,c\n");
                }
                else{
                        /*c is larger than a,b*/
                        max=c;
                        printf("c is larger than a,b\n");
                }
        return 0;
}
```

The for loop

for(< *expression1* > ; < *expression2* > ; < *expression3* >) < *code* >



- In general < expression1 > is used to initialize some counter variable, whereas < expression3 > is used to modify this counter. The counter is also used in < expression2 > as termination criterion.
- All < expression >s may be left out, but the semicolons have to be present.

The for loop – example

```c
#include <stdio.h>
int main(){

        int max = 10;
        int sum = 0;
        int i;

        /* Calculate the sum of 0+1+2+3+4+5+6+7+8+9+10 = 47 */
        for(i=0; i<=max;++i){
                sum+=i;
        }
        printf("Sum of for loop  : %d\n",sum);

        /* Yields the same result*/
        sum=0;
        i=0;
        while(i<=10){
                sum+=i;
                ++i;
        }
        printf("Sum of while loop: %d\n\n",sum);
        return 0;
}
```

The for loop

- continue; The continue statement is an unconditional jump used inside of loop bodies. The instruction pointer jumps directly behind the last instruction in the loop body.

- break; The break-statement causes an unconditional jump inside a loop, too. But with break you immediately leave the loop.

Functions

- Let's assume that you want to calculate the value of the polynomial $P(x) = x^2+x+1$.
- You could insert the code y =x*x+x+1; every time you want to calculate the value of the polynomial.
- But this would be very error-prone if you decide later, that x^2+x+2 is the correct polynomial you want to calculate.

```
double polynomial( double x )
{
    return x*(x + 1) + 1;
}

int main( int nargs, char** pargs )
{
    double my_x = 2,
        my_y = polynomial( my_x );
    return 0;
}
```

General syntax of a function

```
< returntype > < functionname > ( [< parametertype1 >< parametername1 >,
    < parametertype2 >< parametername2 >,
    ... ] )
{
[ return < returnvalue >; ]
}
```

- If the purpose of a function is more just doing something than returning a result, the return value of that function can be void. The statement *return;* does not need to be used then at the end of the function
- The *return*-statement can occur at any position in a function. The function will be left at this point.

Functions – examples

```
double power( double base,
              int exponent )
{
    int i;
    double pow = 1;

    for( i = 0; i <= exponent; i++ )
        pow *= base;

    return pow;
}
```

```
double approx_sqrt( double radicant,
                    int nSteps )
{
    double root = radicant;
    int i = 0;

    if( radicant < 0 ) return -1;
    for( ; i < nSteps; i++ )
        root = 0.5 * (root + radicant/root);

    return root;
}

double geometricAverage( double a,
                         double b )
{
    return approx_sqrt( a*b, 100 );
}
```

Where is the problem?

```
void printMyNameOnce()
{
printMyName( 1 );
}

void printMyName( int n )
{
int i;
for( i = 0; i < n; i++ )
printf( "Homer Simpson/n" );
}
```

```
void printMyName( int n )
{
int i;
for( i = 0; i < n; i++ )
printf( "Homer Simpson/n" );
}

void printMyNameOnce()
{
printMyName( 1 );
}
```

- These two following examples are nearly identical. But the one on the right hand side will work, the one on the left hand side will not even compile.

Function declarations

- To enable the compiler to check the correctness of a function call we must declare the function before its first usage.

```c
#include <stdio.h>
/************************ Declatation ********************************/
void printMyNameOnce();
void printMyName( int n );
/*******************************************************************/

void printMyNameOnce(){ printMyName(1); }

void printMyName(int n){
	int i;
	for(i=0;i < n;++i) printf("Homer Simpson\n");
}
int main(){
	printMyNameOnce();
	return 0;
}
```

Array access

- An array can be seen as a contiguous area in the memory containing variables of the same type arranged in a row. They are kind of serially numbered **starting with 0.**

```
int a[10], i;
for( i = 0; i < 10; i++ ) a[i] = i*i;
for( i = 3; i < 7; i++ )
printf( "a[%d]/2 = %3d/n", i, a[i]/2 );
```

```
a[3]/2 = 4
a[4]/2 = 8
a[5]/2 = 12
a[6]/2 = 18
```

- In C there are no boundary checks for arrays. That means that you can access a[10] in the above example, but the behaviour is not defined.

Array access

- In C there are no boundary checks for arrays. That means that you can access a[10] in the above example, but the behaviour is not defined.

```
int a[10], i;
for( i = 0; i < 10; i++ ) a[i] = i*i;
for( i = 3; i < 7; i++ )
printf( ”a[%d]/2 = %3d/n”, i, a[i]/2 );
```

```
a[3]/2 = 4
a[4]/2 = 8
a[5]/2 = 12
a[6]/2 = 18
```

E.g.:

```
printf( ”%d”, a[10] );
```

```
-1073744920
```

	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]
a :	0	1	4	9	16	25	36	49	64	81	?

Mapping from 2d to 1d

- // double a[nrows][ncols]
- double a[nrows*ncols];
- // a[i][j] = 0.0
- a[i*ncols + j] = 0.0;

j

ncols

i

nrows

0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17
18	19	20	21	22	23

valgrind

- To find memory access problems:
  - valgrind ./executable

cout (c++)

- #include <iostream>
- *using namespace* std;

- int main () {
-   int x = 1.4;
-   cout << "Output sentence"; *// prints Output sentence on screen*
-   cout << 120; *// prints number 120 on screen*
-   cout << “value: “<< x; *// prints “value” and the content of x on screen*
- }

Writing to a file (c++)

```
// basic file operations
#include <iostream>
#include <fstream>
using namespace std;

int main () {
  double result = 2.1;
  ofstream myfile;
  myfile.open ("example.txt");
  myfile << „The result is: “ << result << endl;
  myfile.close();
  return 0;
}
```

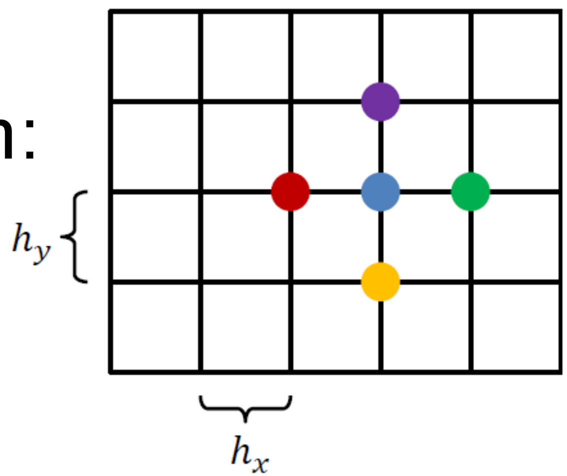
Discretizing an elliptic PDE

- Discretization using the differential quotient for $\Delta u(x,y)$:

$$-\frac{1}{h_x^2}[u(x-h_x,y)+u(x+h_x,y)]-\frac{1}{h_y^2}[u(x,y-h_y)+u(x,y+h_y)]$$

$$=f(x,y)$$

- Solving on a discretized domain:

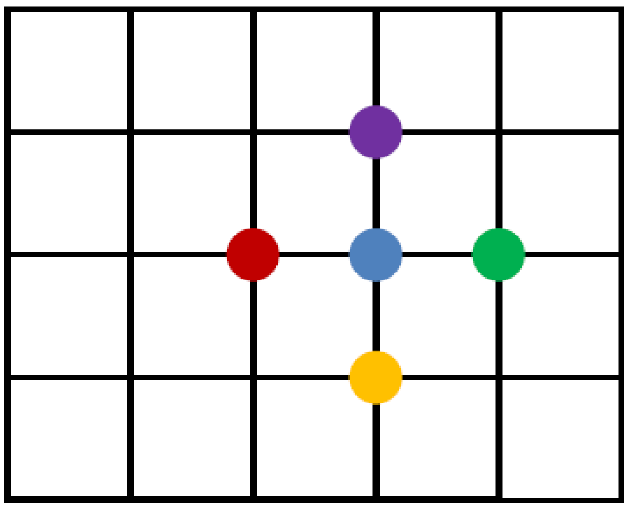


$u(x,y)$ / $f(x,y)$
$u(x-h_x,y)$
$u(x+h_x,y)$
$u(x,y-h_y)$
$u(x,y+h_y)$

Discretizing an elliptic PDE

- For every point we formulate an equation:

$$-\frac{1}{h_x^2}\left[u_{x-1,y} + u_{x+1,y}\right] - \frac{1}{h_y^2}\left[u_{x,y-1} + u_{x,y+1}\right] \quad = f_{x,y}$$



$u_{x,y}$ / $f_{x,y}$

$u_{x-1,y}$

$u_{x+1,y}$

$u_{x,y-1}$

$u_{x,y+1}$

Jacobi Method

- The Jacobi method **iteratively** solves the LSE

(k represents the number of the iteration) according to the following formula:

$$x_i^{k+1} = \frac{1}{a_{ii}}\left(b_i - \sum_{j \neq i} a_{ij} x_j^k\right)$$

- This corresponds to solving the i-th equation of the LSE using the unknowns from the **previous** iteration.

Gauss-Seidel Method

- Solving the i-th equation using the Gauss-Seidel
- method: $$x_i^{k+1} = \frac{1}{a_{ii}}\left(b_i - \underbrace{\sum_{j<i} a_{ij}x_j^{k+1}}_{\text{same iteration}} - \underbrace{\sum_{j>i} a_{ij}x_j^{k}}_{\text{previous iteration}}\right)$$
- Some unknowns come from the previous iteration, some have just been computed in the same iteration $k + 1$.
- Generally **better convergence** compared to the Jacobi method
- There are definitely **data dependencies**: Updating x_i requires some other x_j to be already computed.

Solving an elliptic PDE

- Matrix–free solution of:

$$-\Delta u(x,y) = f(x,y)$$

- can be done by the “stencil”:

$$\begin{bmatrix} & -\frac{1}{h_y^2} & \\ -\frac{1}{h_x^2} & \frac{2}{h_x^2}+\frac{2}{h_y^2} & -\frac{1}{h_x^2} \\ & -\frac{1}{h_y^2} & \end{bmatrix}$$

- Every grid point u_{xy} can be computed by:

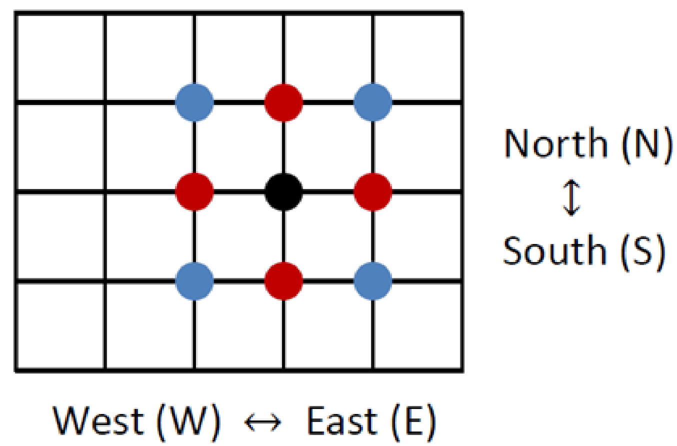
$$u_{x,y} = \frac{1}{\left(\frac{2}{h_x^2}+\frac{2}{h_y^2}\right)}\left(f_{x,y} + \frac{1}{h_x^2}\left[u_{x-1,y} + u_{x+1,y}\right] + \frac{1}{h_y^2}\left[u_{x,y-1} + u_{x,y+1}\right]\right)$$

Red–Black Gauss–Seidel Method

- Data dependencies (take a look at the “update rule”/stencil):

$$u_{x,y} = \frac{1}{\left(\frac{2}{h_x^2}+\frac{2}{h_y^2}\right)} \left(f_{x,y} + \frac{1}{h_x^2}\left[u_{x-1,y} + u_{x+1,y}\right] + \frac{1}{h_y^2}\left[u_{x,y-1} + u_{x,y+1}\right] \right)$$

$$\begin{bmatrix} & -\frac{1}{h_y^2} & \\ -\frac{1}{h_x^2} & \frac{2}{h_x^2}+\frac{2}{h_y^2} & -\frac{1}{h_x^2} \\ & -\frac{1}{h_y^2} & \end{bmatrix}$$

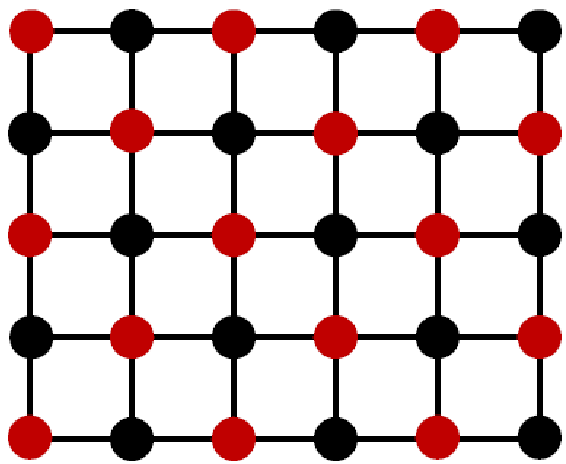


no data dependencies in NW, NE, SW, and SE direction

data dependencies only in W, E, N, and S direction

Red–Black Gauss–Seidel Method

- Parallelization → **checkerboard reordering**:



$$\begin{bmatrix} & NORTH & \\ WEST & CENTER & EAST \\ & SOUTH & \end{bmatrix}$$

A general stencil for the PDE

- Idea: Partition the unknowns into two groups (red and black) so Idea: that there are no data dependencies within each group.
- Consequence: The unknowns within each group can be updated Consequence: independently, i.e. in parallel!

Dynamic memory allocation

- int size = 20;
- double *vec = new double[size];
- vec[2] = 3.1;
- delete [] vec;

printf

”%[flags][field width][precision][length modifier]< *conversionspecifier* >”

conversion specifier	
d,i	*int*
o,u,x,X	*unsigned int*, converted to *octal* (o), *unsigned decimal* (u), *hexadecimal* (x for abcdef, X for ABCDEF) format.
e,E	*double* in exponential format
f,F	*double*, rounded to the decimal format [-]*ddd.ddd*. Default precision is 6.
g,G	*double*, similar to f,e,F,E
c	*int*, converted to the corresponding character
s	constant string

```
int c = ’a’, i;
for( i = 0; i < 5; i++ )
printf( ”ASCII-code: %d, character %c/ n”, c+i, c+i );
```

```
ASCII-code: 97, character a
ASCII-code: 98, character b
ASCII-code: 99, character c
ASCII-code: 100, character d
ASCII-code: 101, character e
```