Scientific Computing WS 2018/2019

Lecture 4

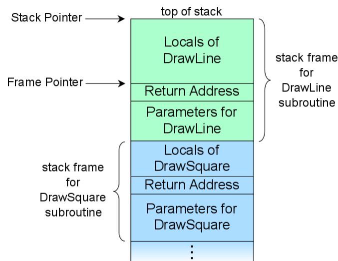
Jürgen Fuhrmann

juergen.fuhrmann@wias-berlin.de

Recap from last time

Memory: stack

- pre-allocated memory where `main()` and all functions called from there put their data.
  - Any time a function is called, the current position in the instruction stream is stored in the stack as the return address, and the called function is allowed to work with the memory space after that




```
void DrawLine(double x0, double y0, double x1, double y1)
{
  paint ...
}

void DrawSquare(double x0,double y0, double a)
{
  DrawLine(x0,y0,x0+a,y0);
  DrawLine(x0+a,y0,x0+a,y0+a);
  DrawLine(x0+a,y0+a,x0,y0+a);
  DrawLine(x0,y0+a,x0,y0);
}
```

By R. S. Shaw, Public Domain,
https://commons.wikimedia.org/w/index.php?curid=1956587

Stack space is scarce

- Variables declared in `{}` blocks get placed on the stack
- All previous examples had their data on the stack, even large arrays
- Stack space should be considered scarce
- Stack size for a program is fixed, set by the system
- On UNIX, use `ulimit -s` to check/set stack size default

Memory: heap

- Chunks from free system memory can be reserved – “allocated” – on demand in order to provide memory space for objects
- The operator `new` reserves the memory and returns an address which can be assigned to a pointer variable
- The operator `delete` (`delete[]` for arrays) releases this memory and makes it available for other processes
- Compared to declarations on the stack, these operations are expensive
- Use cases:
  - Problem sizes unknown at compile time
  - Large amounts of data
  - ...so, yes, we will need this...

```
double *x= new double(5); // allocate space for a double, initialize it with 5
double *y=new double[5];  // allocate space of five doubles, uninitialized
x[3]=1;                   // Segmentation fault
y[3]=1;                   // Perfect...
delete x;                 // Choose the right delete!
delete[] y;               // Choose the right delete!
```

Multidimensional Arrays

- Multidimensional arrays are useful for storing matrices, tensors, arrays of coordinate vectors etc.
- It is easy to declare a multidimensional array on the stack when the size the array is known at compile time

```
double x[5][6];

for (int i=0;i<5;i++)
   for (int j=0;j<6;j++)
      x[i][j]=0;
```

- Determining array dimensions from function parameters may work with some compilers, but are not backed by the standard
- One option to have 2D arrays with arbitrary, run-time defined dimensions is to allocate a an array of pointers to double, and to use `new` to allocate each (!) row

  . . . this leads to nowhere . . .

Classes and members

- Classes are data types which collect different kinds of data, and methods to work on them.

```
class class_name
{
  private:
    private_member1;
    private_member2;
    ...
  public:
    public_member1;
    public_member2;
    ...
};
```

- If not declared otherwise, all members are private
- `struct` is the same as `class` but by default all members are public
- Accessing members of a class object:

```
    class_name x;
    x.public_member1=...
```

- Accessing members of a pointer to class object:

```
    class_name *x;
    (*x).public_member1=...
    x->public_member1=...
```

Example class

- Define a class `vector` which holds data and length information and thus is more comfortable than plain arrays

```
class vector
{
  private:
      double *data;
  public:
       int size;
      double get_value( int i) {return data[i];};
      void set_value( int i, double value); {data[i]=value;};
};

...

{
  vector v;
  v.data=new double(5);  // would work if data would be public
  v.size=5;
  v.set_value(3,5);

  b=v.get_value(3); // now, b=5
  v.size=6;  // size changed, but not the length of the data array...
             // and who is responsible for delete[] at the end of scope ?
}
```

- Methods of a class know all its members
- It would be good to have a method which constructs the vector and another one which destroys it.

Constructors and Destructors

```
class vector
{ private:
      double *data=nullptr;
      int size=0;
  public:
      int get_size(){ return size;};
      double get_value( int i ) { return data[i]; };
      void set_value( int i, double value ) { data[i]=value; };
      Vector( int new_size ) { data = new double[new_size];
                              size=new_size; };
      ~Vector() { delete [] data; };
};
...
{ vector v(5);
  for (int i=0;i<5;i++) v.set_value(i,0.0);
  v.set_value(3,5);
  b=v.get_value(3); // now, b=5
  v.size=6;  // Size is now private and can not be set;
  vector w(5);
  for (int i=0;i<5;i++) w.set_value(i,v.get_value(i));
  // Destructors automatically called at end of scope.
}
```

- Constructors are declared as `classname(...)`
- Destructors are declared as `~classname()`

Interlude: References

- C style access to objects is direct or via pointers
- C++ adds another option - references
  - References essentially are alias names for already existing variables
  - Must always be initialized
  - Can be used in function parameters and in return values
  - No pointer arithmetics with them
- Declaration of reference

```
double a=10.0;
double &b=a;

b=15; // a=15 now as well
```

- Reference as function parameter: no copying of data!

```
void do_multiplication(double x, double y, double &result)
{
    result=x*y;
}
...
double x=5,y=9;
double result=0;
do_multiplication(x,y,result) // result now contains 45
```

Vector class again

- We can define () and [] operators!

```
class vector
{
  private:
      double *data=nullptr;
      int size=0;
  public:
      int get_size( return size);
      double & operator()(int i) { return data[i]; };
      double & operator[](int i) { return data[i]; };
      vector( int new_size) { data = new double[new_size];
                              size=new_size;}
      ~vector() { delete [] data;}
};
...
{
  vector v(5);
  for (int i=0;i<5;i++) v[i]=0.0;
  v[3]=5;
  b=v[3]; // now, b=5
  vector w(5);
  for (int i=0;i<5;i++) w(i)=v(i);
}
```

Matrix class

- We can define (i,j) but not [i,j]

```
class matrix
{ private:
      double *data=nullptr;
      int size=0;  int nrows=0;
      int ncols=0;
  public:
      int get_nrows( return nrows);
      int get_ncols( return ncols);
      double & operator()(int i,int j) { return data[i*nrow+j]);
      matrix( int new_rows,new_cols)
      { nrows=new_rows; ncols=new_cols;
        size=nrows*ncols;
        data = new double[size];
      }
      ~matrix() { delete [] data;}
};
...
{
  matrix m(3,3);
  for (int i=0;i<3;i++)
    for (int j=0;j<3;j++)
        m(i,j)=0.0;
}
```

Inheritance

- Classes in C++ can be extended, creating new classes which retain characteristics of the base class.
- The *derived class* inherits the members of the *base class*, on top of which it can add its own members.

```
class vector2d
{  private:
     double *data;
     int nrow, ncol;
     int size;
   public:
     double & operator(int i, int j);
     vector2d(int nrow, ncol);
     ~vector2d();
}
class matrix: public vector2d
{ public:
     apply(const vector1d & u, vector1d &v);
     solve(vector1d &u, const vector1d &rhs);
}
```

- All operations which can be performed with instances of `vector2d` can be performed with instances of `matrix` as well
- In addition, `matrix` has methods for linear system solution and matrix-vector multiplication

Generic programming: templates

- Templates allow to write code where a data type is a parameter
- We want do be able to have vectors of any basic data type.
- We do not want to write new code for each type

```
template <typename T>
class vector
{
  private:
      T *data=nullptr;
      int size=0;
  public:
      int get_size( return size);
      T & operator[](int i) { return data[i]; };
      vector( int new_size) { data = new T[new_size];
                              size = new_size;};
      ~vector() { delete [] data;};
};
...
{
  vector<double> v(5);
  vector<int> iv(3);
}
```

C++ template libray

- The standard template library (STL) became part of the C++11 standard
- Whenever you can, use the classes available from there
- For one-dimensional data, std::vector is appropriate
- For two-dimensional data, things become more complicated
  - There is no reasonable matrix class
    - std::vector< std::vector> is possible but has to allocate each matrix row and is inefficient
  - it is hard to create a std::vector from already existing data

Smart pointers

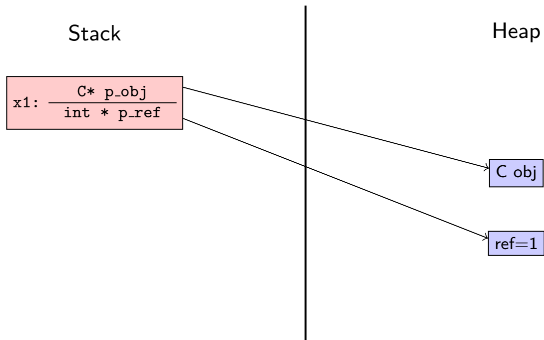
. . . with a little help from Timo Streckenbach from WIAS who introduced smart pointers into our simulation code.

- Automatic book-keeping of pointers to objects in memory.
- Instead of the meory address of an object aka. pointer, a structure is passed around *by value* which holds the memory address and a pointer to a *reference count* object.
- It delegates the member access operator `->` and the address resolution operator `*` to the pointer it contains.
- Each assignment of a smart pointer increases this reference count.
- Each destructor invocation from a copy of the smart pointer structure decreases the reference count.
- If the reference count reaches zero, the memory is freed.
- `std::shared_ptr` is part of the C++11 standard

Smart pointer schematic

(this is one possibe way to implement it)

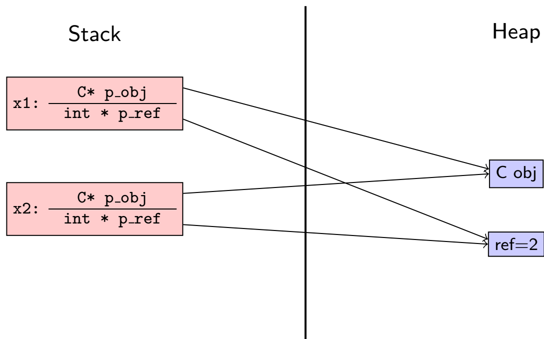
```
class C;
```



Smart pointer schematic

(this is one possibe way to implement it)

```
class C;
```

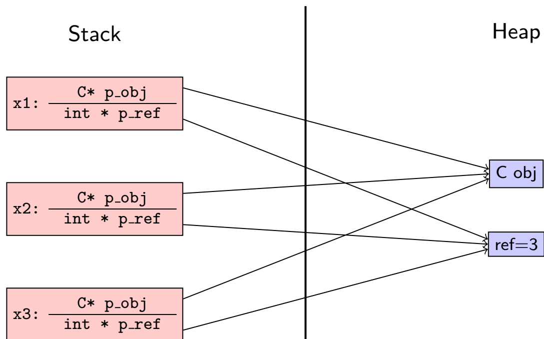



```
std::shared_ptr<C> x1= std::make_shared<C>();
std::shared_ptr<C> x2= x1;
```

Smart pointer schematic

(this is one possibe way to implement it)

```
class C;
```




```
std::shared_ptr<C> x1= std::make_shared<C>();
std::shared_ptr<C> x2= x1;
std::shared_ptr<C> x3= x1;
```

C/C++: Expression templates

Vector operations

- So far we are able to perform vector operations by explicitely writing loops over the length of the vector
- Generally, C++ allows to *overload* operators like +,-,*,/,= etc. allowing to use vector expressions (like in matlab, python/numpy, Julia)

```
    inline const vector
    operator+( const vector& a, const vector& b )
    {
      vector tmp( a.size() );
      for( std::size_t i=0; i<a.size(); ++i )
      tmp[i] = a[i] + b[i];
      return tmp;
    }
    ...

  vector a,b,c;
  c=a+b;
```

- *But* this involves the creation of a temporary object for each operation in an expression
- Temporary object creation is prohibitively expensive for large objects

Expression templates I

C++ technique which allows to implement expressions of vectors while avoiding introduction and copies of temporary objects.

- Expression class definition:

```
template< typename A, typename B >
class Sum {
  public:
  Sum(const A& a, const B& b): a_( a ), b_( b ){} // Constructor from two vectors
  std::size_t size() const { return a_.size(); }  // Delegate size() to argument
  double operator[]( std::size_t i ) const        // Access operator
  { return a_[i] + b_[i]; }
  private:
  const A& a_;    // Reference to the left-hand side operand
  const B& b_;    // Reference to the right-hand side operand
};
```

- Overloaded + operator:

```
template< typename A, typename B >
const Sum<A,B> operator+( const A& a, const B& b )
{
  return Sum<A,B>( a, b ); // Return instance of Sum<A,B>
}
```

- a,b can be vectors, other expressions or any object which implements `size()` and `[]`.

Expression templates II

- Method in vector class to copy vector data from expression:

```
class vector
{
  public:
  ...
  template< typename A >
  vector& operator=( const A& expr )
  {
    for( std::size_t i=0; i<expr.size(); i++ )
        v_[i] = expr[i];
    return *this; // Return reference to target vector
  }
  ...
};
```

- Usage:

```
vector a,b,c;
c=a+b;
```

- After template instantiation and inlining, the compiler will generate code without temporary vector objects:

```
for( std::size_t i=0; i<a.size(); i++ )
   c[i] = a[i] + b[i];
```

- Large potential for optimization for more complex expressions

Vector classes for linear algebra

- Expression templates and overloading of component access allow to implement classes for linear algebra which are almost as easy to use as in matlab or python/numpy
- These techniques are used by libraries like
  - Eigen `http://eigen.tuxfamily.org`
  - Armadillo `http://arma.sourceforge.net/`
  - Blaze `https://bitbucket.org/blaze-lib/blaze/overview`
- Regrettably, none of this is standardized in C++ ...
- During the course, we will use our own, small and therefore hopefully easy to understand library named numcxx

C++ topics not covered so far

- To be covered later
  - Threads/parallelism
  - Graphics (via library)
- To be covered on occurence (possibly)
  - Character strings
  - Overloading
  - malloc/free/realloc (C-style memory management)
  - cmath library
  - Interfacing C/Fortran
- To be omitted (probably)
  - Functor classes, lambdas
  - optional arguments, variable parameter lists
  - Exceptions
  - Move semantics
  - GUI libraries
  - Interfacing Python/numpy

Recap from numerical analysis

Representation of real numbers

- Any real number $x \in \mathbb{R}$ can be expressed via representation formula:

$$x = \pm \sum_{i=0}^{\infty} d_i \beta^{-i} \beta^e$$

  - $\beta \in \mathbb{N}, \beta \geq 2$: base
  - $d_i \in \mathbb{N}, 0 \leq d_i < \beta$: mantissa digits
  - $e \in \mathbb{Z}$: exponent
- Scientific notation of floating point numbers: e.g. $x = 6.022 \cdot 10^{23}$
  - $\beta = 10$
  - $d = (6, 0, 2, 2, 0 \dots)$
  - $e = 23$
- Non-unique: $x = 0.6022 \cdot 10^{24}$
  - $\beta = 10$
  - $d = (0, 6, 0, 2, 2, 0 \dots)$
  - $e = 24$
- Infinite for periodic decimal numbers, irrational numbers

Floating point numbers

- Computer representation uses $\beta = 2$, therefore $d_i \in \{0, 1\}$
- Truncation to fixed finite size

$$x = \pm \sum_{i=0}^{t-1} d_i \beta^{-i} \beta^e$$

  - t: mantissa length
  - Normalization: assume $d_0 = 1 \Rightarrow$ save one bit for mantissa
  - k: exponent size $-\beta^k + 1 = L \leq e \leq U = \beta^k - 1$
  - Extra bit for sign
  - $\Rightarrow$ storage size: $(t - 1) + k + 1$
- IEEE 754 single precision (C++ `float`): $k = 8, t = 24 \Rightarrow 32$ bit
- IEEE 754 double precision (C++ `double`): $k = 11, t = 53 \Rightarrow 64$ bit

Floating point limits

Finite size of reprensentation $\Rightarrow$ there are minimal and maximal possible numbers which can be represented

- symmetry wrt. 0 because of sign bit
- smallest positive normalized number: $d_0 = 1, d_i = 0, i = 1 \dots t-1$
  $x_{min} = \beta^L$
  - `float`: 1.175494351e–38
  - `double`: 2.2250738585072014e–308
- smallest positive denormalized number: $d_i = 0, i = 0 \dots t-2, d_{t-1} = 1$
  $x_{min} = \beta^{1-t}\beta^L$
- largest positive normalized number: $d_i = \beta - 1, 0 \dots t-1$
  $x_{max} = \beta(1 - \beta^{1-t})\beta^U$
  - `float`: 3.402823466e+38
  - `double`: 1.7976931348623158e+308

Machine precision

- There cannot be more than 2^{t+k} floating point numbers $\Rightarrow$ almost all real numbers have to be approximated
- Let x be an exact value and $\tilde{x}$ be its approximation Then: $|\frac{\tilde{x}-x}{x}| < \epsilon$ is the best accuracy estimate we can get, where
  - $\epsilon = \beta^{1-t}$ (truncation)
  - $\epsilon = \frac{1}{2}\beta^{1-t}$ (rounding)
- Also: ϵ is the smallest representable number such that $1 + \epsilon > 1$.
- Relative errors show up in partiular when
  - subtracting two close numbers
  - adding smaller numbers to larger ones

Matrix + Vector norms

- Vector norms: let $x = (x_i) \in \mathbb{R}^n$
  - $||x||_1 = \sum_{i=1}^n |x_i|$: sum norm, l_1-norm
  - $||x||_2 = \sqrt{\sum_{i=1}^n x_i^2}$: Euclidean norm, l_2-norm
  - $||x||_\infty = \max_{i=1}^n |x_i|$: maximum norm, l_∞-norm
- Matrix $A = (a_{ij}) \in \mathbb{R}^n \times \mathbb{R}^n$
  - Representation of linear operator $\mathcal{A} : \mathbb{R}^n \to \mathbb{R}^n$ defined by $\mathcal{A} : x \mapsto y = Ax$ with

$$y_i = \sum_{j=1}^n a_{ij} x_j$$

  - Induced matrix norm:

$$\begin{aligned} ||A||_\nu &= \max_{x \in \mathbb{R}^n, x \neq 0} \frac{||Ax||_\nu}{||x||_\nu} \\ &= \max_{x \in \mathbb{R}^n, ||x||_\nu = 1} \frac{||Ax||_\nu}{||x||_\nu} \end{aligned}$$

Matrix norms

- $||A||_1 = \max_{j=1}^{n} \sum_{i=1}^{n} |a_{ij}|$ maximum of column sums
- $||A||_\infty = \max_{i=1}^{n} \sum_{j=1}^{n} |a_{ij}|$ maximum of row sums
- $||A||_2 = \sqrt{\lambda_{max}}$ with λ_{max}: largest eigenvalue of $A^T A$.

Matrix condition number and error propagation

- Problem: solve $Ax = b$, where b is inexact
- Let Δb be the error in b and Δx be the resulting error in x such that

$$A(x + \Delta x) = b + \Delta b.$$

- Since $Ax = b$, we get $A\Delta x = \Delta b$
- Therefore

$$\left\{ \begin{array}{rl} \Delta x & = A^{-1}\Delta b \\ Ax & = b \end{array} \right\} \Rightarrow \left\{ \begin{array}{rl} ||A|| \cdot ||x|| & \geq ||b|| \\ ||\Delta x|| & \leq ||A^{-1}|| \cdot ||\Delta b|| \end{array} \right.$$

$$\Rightarrow \frac{||\Delta x||}{||x||} \leq \kappa(A) \frac{||\Delta b||}{||b||}$$

where $\kappa(A) = ||A|| \cdot ||A^{-1}||$ is the *condition number* of A.

Solution of linear systems of equations

Approaches to linear system solution

Let A: $n \times n$ matrix, $b \in \mathbb{R}^n$.

Solve $Ax = b$

- Direct methods:
  - Exact
    - up to machine precision
    - condition number
  - Expensive (in time and space)
    - where does this matter ?
- Iterative methods:
  - Only approximate
    - with good convergence and proper accuracy control, results are not worse than for direct methods
  - May be cheaper in space and (possibly) time
  - Convergence guarantee is problem dependent and can be tricky

Complexity: "big O notation"

- Let $f, g : \mathbb{V} \to \mathbb{R}^+$ be some functions, where $\mathbb{V} = \mathbb{N}$ or $\mathbb{V} = \mathbb{R}$.

  We write
  $$f(x) = O(g(x)) \quad (x \to \infty)$$
  if there exist a constant $C > 0$ and $x_0 \in \mathbb{V}$ such that
  $$\forall x > x_0, \quad |f(x)| \leq C|g(x)|$$

- Often, one skips the part "$(x \to \infty)$"
- Examples:
  - Addition of two vectors: $O(n)$
  - Matrix-vector multiplication (for matrix where all entries are assumed to be nonzero): $O(n^2)$

Really bad example of direct method

Solve $Ax = b$ by Cramer's rule

$$x_i = \begin{vmatrix} a_{11} & a_{12} & \dots & a_{1i-1} & b_1 & a_{1i+1} & \dots & a_{1n} \\ a_{21} & & \dots & & b_2 & & \dots & a_{2n} \\ \vdots & & & & \vdots & & & \vdots \\ a_{n1} & & \dots & & b_n & & \dots & a_{nn} \end{vmatrix} / |A| \quad (i = 1 \dots n)$$

This takes $O(n!)$ operations...

Gaussian elimination

- Essentially the only feasible direct solution method
- Solve $Ax = b$ with square matrix A.
- While formally, the algorithm is always the same, its implementation depends on
  - data structure to store matrix
  - possibility to ignore zero entries for matrices with many zeroes
  - sorting of elements

Gaussian elemination: pass 1

$$\begin{pmatrix} 6 & -2 & 2 \\ 12 & -8 & 6 \\ 3 & -13 & 3 \end{pmatrix} x = \begin{pmatrix} 16 \\ 26 \\ -19 \end{pmatrix}$$

Step 1: $\text{equation}_2 \leftarrow \text{equation}_2 - 2\,\text{equation}_1$
$\text{equation}_3 \leftarrow \text{equation}_3 - \frac{1}{2}\,\text{equation}_1$

$$\begin{pmatrix} 6 & -2 & 2 \\ 0 & -4 & 2 \\ 0 & -12 & 2 \end{pmatrix} x = \begin{pmatrix} 16 \\ -6 \\ -27 \end{pmatrix}$$

Step 2: $\text{equation}_3 \leftarrow \text{equation}_3 - 3\,\text{equation}_2$

$$\begin{pmatrix} 6 & -2 & 2 \\ 0 & -4 & 2 \\ 0 & 0 & -4 \end{pmatrix} x = \begin{pmatrix} 16 \\ -6 \\ -9 \end{pmatrix}$$

Gaussian elimination: pass 2

Solve upper triangular system

$$\begin{pmatrix} 6 & -2 & 2 \\ 0 & -4 & 2 \\ 0 & 0 & -4 \end{pmatrix} x = \begin{pmatrix} 16 \\ -6 \\ -9 \end{pmatrix}$$

$$\begin{aligned}
-4x_3 &= -9 & & & \Rightarrow x_3 &= \frac{9}{4} \\
-4x_2 + 2x_3 &= -6 & \Rightarrow -4x_2 &= -\frac{21}{2} & \Rightarrow x_2 &= \frac{21}{8} \\
6x_1 - 2x_2 + 2x_3 &= 2 & \Rightarrow 6x_1 &= 2 + \frac{21}{4} - \frac{18}{4} = \frac{11}{4} & \Rightarrow x_1 &= \frac{11}{4}
\end{aligned}$$

LU factorization

Pass 1 expressed in matrix operation

$$L_1 A x = \begin{pmatrix} 6 & -2 & 2 \\ 0 & -4 & 2 \\ 0 & -12 & 2 \end{pmatrix} x = \begin{pmatrix} 16 \\ -6 \\ -27 \end{pmatrix} = L_1 b, \qquad L_1 = \begin{pmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ -\frac{1}{2} & 0 & 1 \end{pmatrix}$$

$$L_2 L_1 A x = \begin{pmatrix} 6 & -2 & 2 \\ 0 & -4 & 2 \\ 0 & 0 & -4 \end{pmatrix} x = \begin{pmatrix} 16 \\ -6 \\ -9 \end{pmatrix} = L_2 L_1 b, \qquad L_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -3 & 1 \end{pmatrix}$$

- Let $L = L_1^{-1} L_2^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ \frac{1}{2} & 3 & 1 \end{pmatrix}$, $U = L_2 L_1 A$. Then $A = LU$
- Inplace operation. Diagonal elements of L are always 1, so no need to store them $\Rightarrow$ work on storage space for A and overwrite it.

LU factorization

Solve $Ax = b$

- Pass 1: factorize $A = LU$ such that L, U are lower/upper triangular
- Pass 2: obtain $x = U^{-1}L^{-1}b$ by solution of lower/upper triangular systems
  - 1. solve $L\tilde{x} = b$
  - 2. solve $Ux = \tilde{x}$
- We never calculate A^{-1} as this would be more expensive

Problem example

- Consider $\begin{pmatrix} \epsilon & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1+\epsilon \\ 1 \end{pmatrix}$
- Solution: $\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$
- Machine arithmetic: Let $\epsilon << 1$ such that $1+\epsilon = 1$.
- Equation system in machine arithmetic:
  $1 \cdot \epsilon + 1 \cdot 1 = 1 + \epsilon$

  $1 \cdot 1 + 1 \cdot 1 = 2$
- Still fulfilled!

Problem example II: Gaussian elimination

- Ordinary elimination: $\mathrm{equation}_2 \leftarrow \mathrm{equation}_2 - \frac{1}{\epsilon}\,\mathrm{equation}_1$
  $\begin{pmatrix} \epsilon & 1 \\ 0 & 1-\frac{1}{\epsilon} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1+\epsilon \\ 2-\frac{1+\epsilon}{\epsilon} \end{pmatrix}$
- In exact arithmetic:

$$\Rightarrow x_2 = \frac{1-\frac{1}{\epsilon}}{1-\frac{1}{\epsilon}} = 1 \Rightarrow x_1 = \frac{1+\epsilon-x_2}{\epsilon} = 1$$

- In floating point arithmetic: $1+\epsilon = 1$, $1-\frac{1}{\epsilon} = -\frac{1}{\epsilon}$, $2-\frac{1}{\epsilon} = -\frac{1}{\epsilon}$:
  $\begin{pmatrix} \epsilon & 1 \\ 0 & -\frac{1}{\epsilon} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ -\frac{1}{\epsilon} \end{pmatrix}$
  $\Rightarrow x_2 = 1 \Rightarrow \epsilon x_1 + 1 = 1 \Rightarrow x_1 = 0$

Problem example III: Partial Pivoting

- Before elimination step, look at the element with largest absolute value in current column and put the corresponding row "on top" as the "pivot"
- This prevents near zero divisions and increases stability

$$\begin{pmatrix} 1 & 1 \\ \epsilon & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 2 \\ 1+\epsilon \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & 1 \\ 0 & 1-\epsilon \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 2 \\ 1-\epsilon \end{pmatrix}$$

- Independent of ϵ:

$$x_2 = \frac{1-\epsilon}{1-\epsilon} = 1, \qquad x_1 = 2 - x_2 = 1$$

- Instead of A, factorize PA: $PA = LU$, where P is a permutation matrix which can be encoded using an integer vector

Gaussian elimination and LU factorization

- Full pivoting: in addition to row exchanges, perform column exchanges to ensure even larger pivots. Seldomly used in practice.
- Gaussian elimination with partial pivoting is the "working horse" for direct solution methods
- Complexity of LU-Factorization: $O(N^3)$, some theoretically better algorithms are known with e.g. $O(N^{2.736})$
- Complexity of triangular solve: $O(N^2)$
  $\Rightarrow$ overall complexity of linear system solution is $O(N^3)$

Cholesky factorization

- $A = LL^T$ for symmetric, positive definite matrices

BLAS, LAPACK

- BLAS: Basic Linear Algebra Subprograms `http://www.netlib.org/blas/`
  - Level 1 - vector-vector: $\mathbf{y} \leftarrow \alpha\mathbf{x} + \mathbf{y}$
  - Level 2 - matrix-vector: $\mathbf{y} \leftarrow \alpha A\mathbf{x} + \beta\mathbf{y}$
  - Level 3 - matrix-matrix: $C \leftarrow \alpha AB + \beta C$
- LAPACK: Linear Algebra PACKage `http://www.netlib.org/lapack/`
  - Linear system solution, eigenvalue calculation etc.
  - `dgetrf`: LU factorization
  - `dgetrs`: LU solve
- Used in overwhelming number of codes (e.g. matlab, scipy etc.). Also, C++ matrix libraries use these routines. Unless there is special need, they should be used.
- Reference implementations in Fortran, but many more implementations available which carefully work with cache lines etc.

Matrices from PDEs

- So far, we assumed that matrices are stored in a two-dimensional, $n \times n$ array of numbers
- This kind of matrices are also called *dense* matrices
- As we will see, matrices from PDEs (can) have a number of structural properties one can take advantage of when storing a matrix and solving the linear system

1D heat conduction

- v_L, v_R: ambient temperatures, α: heat transfer coefficient
- Second order boundary value problem in $\Omega = [0, 1]$:

$$
\begin{aligned}
-u''(x) &= f(x) \qquad \text{in}\Omega \\
-u'(0) + \alpha(u(0) - v_L) &= 0 \\
u'(1) + \alpha(u(1) - v_R) &= 0
\end{aligned}
$$

- Let $h = \frac{1}{n-1}$, $x_i = x_0 + (i-1)h$ $i = 1 \dots n$ be discretization points, let u_i approximations for $u(x_i)$ and $f_i = f(x_i)$
- Finite difference approximation:

$$
\begin{aligned}
-u'(0) + \alpha(u(0) - v_L) &\approx \frac{1}{h}(u_0 - u_1) + \alpha(u_0 - v_L) \\
-u''(x_i) - f(x_i) &\approx \frac{1}{h^2}(-u_{i+1} + 2u_i - u_{i-1}) - f_i \quad (i = 2 \dots n-1) \\
u'(1) + \alpha(u(1) - v_R) &\approx \frac{1}{h}(u_n - u_{n-1}) + \alpha(u_n - v_R)
\end{aligned}
$$

1D heat conduction: discretization matrix

- equations $2 \dots n-1$ multiplied by h
- only nonzero entries written

$$
\begin{pmatrix}
\alpha + \frac{1}{h} & -\frac{1}{h} & & & & & \\
-\frac{1}{h} & \frac{2}{h} & -\frac{1}{h} & & & & \\
 & -\frac{1}{h} & \frac{2}{h} & -\frac{1}{h} & & & \\
 & \ddots & \ddots & \ddots & \ddots & & \\
 & & -\frac{1}{h} & \frac{2}{h} & -\frac{1}{h} & & \\
 & & & -\frac{1}{h} & \frac{2}{h} & -\frac{1}{h} & \\
 & & & & -\frac{1}{h} & \frac{1}{h} + \alpha
\end{pmatrix}
\begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_{N-2} \\ u_{N-1} \\ u_N \end{pmatrix}
=
\begin{pmatrix} \alpha v_L \\ hf_2 \\ hf_3 \\ \vdots \\ hf_{N-2} \\ hf_{N-1} \\ \alpha v_R \end{pmatrix}
$$

- Each row contains ≤ 3 elements
- Only $3n-2$ of n^2 elements are non-zero

General tridiagonal matrix

$$
\begin{pmatrix}
b_1 & c_1 & & & \\
a_2 & b_2 & c_2 & & \\
 & a_3 & b_3 & \ddots & \\
 & & \ddots & \ddots & c_{n-1} \\
 & & & a_n & b_n
\end{pmatrix}
\begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_n \end{pmatrix}
=
\begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ \vdots \\ f_n \end{pmatrix}
$$

- To store matrix, it is sufficient to store only nonzero elements in three one-dimensional arrays for a_i, b_i, c_i, respectively

Gaussian elimination for tridiagonal systems

Gaussian elimination using arrays a, b, c as matrix storage ?

From what we have seen, this question arises in a quite natural way, and historically, the answer has been given several times

- TDMA (tridiagonal matrix algorithm)
- "Thomas algorithm" (Llewellyn H. Thomas, 1949 (?))
- "Progonka method" (from Russian "run through"; Gelfand, Lokutsievski, 1952, published 1960)

Progonka: derivation

- $a_i u_{i-1} + b_i u_i + c_i u_{i+1} = f_i \quad (i = 1 \dots n)$; $a_1 = 0$, $c_N = 0$
- For $i = 1 \dots n-1$, assume there are coefficients α_i, β_i such that $u_i = \alpha_{i+1} u_{i+1} + \beta_{i+1}$.
- Then, we can express u_{i-1} and u_i via u_{i+1}: $(a_i\alpha_i\alpha_{i+1} + b_i\alpha_{i+1} + c_i)u_{i+1} + a_i\alpha_i\beta_{i+1} + a_i\beta_i + b_i\beta_{i+1} - f_i = 0$
- This is true independently of u if

$$\begin{cases} a_i\alpha_i\alpha_{i+1} + b_i\alpha_{i+1} + c_i & = 0 \\ a_i\alpha_i\beta_{i+1} + a_i\beta_i + b_i\beta_{i+1} - f_i & = 0 \end{cases}$$

- or for $i = 1 \dots n-1$:

$$\begin{cases} \alpha_{i+1} & = -\frac{c_i}{a_i\alpha_i + b_i} \\ \beta_{i+1} & = \frac{f_i - a_i\beta_i}{a_i\alpha_i + b_i} \end{cases}$$

Progonka: realization

- Forward sweep:

$$\begin{cases} \alpha_2 & = -\frac{c_1}{b_1} \\ \beta_2 & = \frac{f_i}{b_1} \end{cases}$$

for $i = 2 \dots n-1$

$$\begin{cases} \alpha_{i+1} & = -\frac{c_i}{a_i\alpha_i + b_i} \\ \beta_{i+1} & = \frac{f_i - a_i\beta_i}{a_i\alpha_i + b_i} \end{cases}$$

- Backward sweep:

$$u_n = \frac{f_n - a_n\beta_n}{a_n\alpha_n + b_n}$$

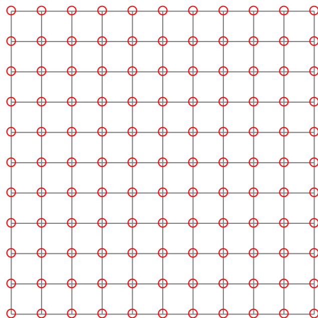
for $n-1 \dots 1$:

$$u_i = \alpha_{i+1}u_{i+1} + \beta_{i+1}$$

Progonka: properties

- n unknowns, one forward sweep, one backward sweep
  $\Rightarrow O(n)$ operations vs. $O(n^3)$ for algorithm using full matrix
- No pivoting $\Rightarrow$ stability issues
  - Stability for diagonally dominant matrices ($|b_i| > |a_i| + |c_i|$)
  - Stability for symmetric positive definite matrices

2D finite difference grid



- Each discretization point has not more then 4 neighbours
- Matrix can be stored in five diagonals,
  LU factorization not anymore $\equiv$ "fill-in"
- Certain iterative methods can take advantage of the regular and hierachical structure (multigrid) and are able to solve system in $O(n)$ operations
- Another possibility: fast Fourier transform with $O(n \log n)$ operations

No lecture on Thu Nov 1!

(Project review of MATH$^+$ excellence cluster)

First homework will be on course homepage (probably by Nov.1)