Scientific Computing WS 2017/2018

Lecture 11

Jürgen Fuhrmann

juergen.fuhrmann@wias-berlin.de

Solution of SPD system as a minimization procedure

Regard $Au = f$,where A is symmetric, positive definite. Then it defines a bilinear form $a : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$

$$a(u, v) = (Au, v) = v^T Au = \sum_{i=1}^{n} \sum_{j=1}^{n} a_{ij} v_i u_j$$

As A is SPD, for all $u \neq 0$ we have $(Au, u) > 0$.

For a given vector b, regard the function

$$f(u) = \frac{1}{2} a(u, u) - b^T u$$

What is the minimizer of f ?

$$f'(u) = Au - b = 0$$

- Solution of SPD system $\equiv$ minimization of f.

Conjugate directions

For steepest descent, there is no guarantee that a search direction $d_i = r_i = -Ae_i$ is not used several times. If all search directions would be orthogonal, or, indeed, A-orthogonal, one could control this situation.

So, let $d_0, d_1 \ldots d_{n-1}$ be a series of A-orthogonal (or conjugate) search directions, i.e. $(Ad_i, d_j) = 0$, $i \neq j$.

- Look for u_{i+1} in the direction of d_i such that it minimizes f in this direction, i.e. set $u_{i+1} = u_i + \alpha_i d_i$ with α choosen from

$$
\begin{aligned}
0 &= \frac{d}{d\alpha} f(u_i + \alpha d_i) = f'(u_i + \alpha d_i) \cdot d_i \\
&= (b - A(u_i + \alpha d_i), d_i) \\
&= (b - Au_i, d_i) - \alpha(Ad_i, d_i) \\
&= (r_i, d_i) - \alpha(Ad_i, d_i) \\
\alpha_i &= \frac{(r_i, d_i)}{(Ad_i, d_i)}
\end{aligned}
$$

Conjugate directions II

$e_0 = u_0 - \hat{u}$ (such that $Ae_0 = -r_0$) can be represented in the basis of the search directions:

$$e_0 = \sum_{i=0}^{n-1} \delta_j d_j$$

Projecting onto d_k in the A scalar product gives

$$\begin{aligned}
(Ae_0, d_k) &= \sum_{i=0}^{n-1} \delta_j (Ad_j, d_k) \\
&= \delta_k (Ad_k, d_k) \\
\delta_k &= \frac{(Ae_0, d_k)}{(Ad_k, d_k)} = \frac{(Ae_0 + \sum_{i<k} \alpha_i d_i, d_k)}{(Ad_k, d_k)} = \frac{(Ae_k, d_k)}{(Ad_k, d_k)} \\
&= \frac{(r_k, d_k)}{(Ad_k, d_k)} \\
&= -\alpha_k
\end{aligned}$$

Conjugate directions III

Then,

$$e_i = e_0 + \sum_{j=0}^{i-1} \alpha_j d_j = -\sum_{j=0}^{n-1} \alpha_j d_j + \sum_{j=0}^{i-1} \alpha_j d_j$$
$$= -\sum_{j=i}^{n-1} \alpha_j d_j$$

So, the iteration consists in component-wise suppression of the error, and it must converge after n steps. Let $k \leq i$. A-projection on d_k gives

$$(Ae_i, d_k) = -\sum_{j=i}^{n-1} \alpha_j (Ad_j, d_k) = 0$$

Therefore, $r_i = Ae_i$ is orthogonal to $d_0 \ldots d_{i-1}$.

Conjugate directions IV

Looking at the error norm $||e_i||_A$, the method yields the element with the minimum energy norm from all elements of the affine space $e_0 + \mathcal{K}_i$ where $\mathcal{K}_i = \mathrm{span}\{d_0, d_1 \ldots d_{i-1}\}$

$$(Ae_i, e_i) = \left(\sum_{j=i}^{n-1} \delta_j d_j, \sum_{j=i}^{n-1} \delta_j d_j \right) = \sum_{j=i}^{n-1} \sum_{k=i}^{n-1} \delta_j \delta_k (d_j, d_k)$$

$$= \sum_{j=i}^{n-1} \delta_j^2 (d_j, d_j) = \min_{e \in e_0 + \mathcal{K}_i} ||e||_A$$

Furthermore, we have

$$\begin{aligned} u_{i+1} &= u_i + \alpha_i d_i \\ e_{i+1} &= e_i + \alpha_i d_i \\ Ae_{i+1} &= Ae_i + \alpha_i Ad_i \\ r_{i+1} &= r_i - \alpha_i Ad_i \end{aligned}$$

By what magic we can obtain these d_i?

Gram-Schmidt Orthogonalization

- Assume we have been given some linearly independent vectors $v_0, v_1 \dots v_{n-1}$.
- Set $d_0 = v_0$
- Define

$$d_i = v_i + \sum_{k=0}^{i-1} \beta_{ik} d_k$$

- For $j < i$, A-project onto d_j and require orthogonality:

$$(Ad_i, d_j) = (Av_i, d_j) + \sum_{k=0}^{i-1} \beta_{ik} (Ad_k, d_j)$$

$$0 = (Av_i, d_j) + \beta_{ij} (Ad_j, d_j)$$

$$\beta_{ij} = -\frac{(Av_i, d_j)}{(Ad_j, d_j)}$$

- If v_i are the coordinate unit vectors, this is Gaussian elimination!
- If v_i are arbitrary, they all must be kept in the memory

Conjugate gradients (Hestenes, Stiefel, 1952)

As Gram-Schmidt builds up d_i from d_j, $j < i$, we can choose $v_i = r_i$, i.e. the residuals built up during the conjugate direction process.

Let $\mathcal{K}_i = \mathrm{span}\{d_0 \dots d_{i-1}\}$. Then, $r_i \perp \mathcal{K}_i$

But d_i are built by Gram-Schmidt from the residuals, so we also have $\mathcal{K}_i = \mathrm{span}\{r_0 \dots r_{i-1}\}$ and $(r_i, r_j) = 0$ for $j < i$.

From $r_i = r_{i-1} - \alpha_{i-1} A d_{i-1}$ we obtain

$\mathcal{K}_i = \mathcal{K}_{i-1} \cup \mathrm{span}\{A d_{i-1}\}$

This gives two other representations of $\mathcal{K}_i$:

$$\begin{aligned}\mathcal{K}_i &= \mathrm{span}\{d_0, Ad_0, A^2 d_0, \dots, A^{i-1} d_0\} \\ &= \mathrm{span}\{r_0, Ar_0, A^2 r_0, \dots, A^{i-1} r_0\}\end{aligned}$$

Such type of subspace of $\mathbb{R}^n$ is called *Krylov subspace*, and orthogonalization methods are more often called *Krylov subspace methods*.

Conjugate gradients II

Look at Gram-Schmidt under these conditions. The essential data are (setting $v_i = r_i$ and using $j < i$) $\beta_{ij} = -\frac{(Ar_i, d_j)}{(Ad_j, d_j)} = -\frac{(Ad_j, r_i)}{(Ad_j, d_j)}$.

Then, for $j \leq i$:

$$
\begin{aligned}
r_{j+1} &= r_j - \alpha_j A d_j \\
(r_{j+1}, r_i) &= (r_j, r_i) - \alpha_j (A d_j, r_i) \\
\alpha_j (A d_j, r_i) &= (r_j, r_i) - (r_{j+1}, r_i)
\end{aligned}
$$

$$
(Ad_j, r_i) = \begin{cases} -\frac{1}{\alpha_j}(r_{j+1}, r_i), & j+1 = i \\ \frac{1}{\alpha_j}(r_j, r_i), & j = i \\ 0, & \text{else} \end{cases} = \begin{cases} -\frac{1}{\alpha_{i-1}}(r_i, r_i), & j+1 = i \\ \frac{1}{\alpha_i}(r_i, r_i), & j = i \\ 0, & \text{else} \end{cases}
$$

For $j < i$:

$$
\beta_{ij} = \begin{cases} \frac{1}{\alpha_{i-1}} \frac{(r_i, r_i)}{(Ad_{i-1}, d_{i-1})}, & j+1 = i \\ 0, & \text{else} \end{cases}
$$

Conjugate gradients III

For Gram-Schmidt we defined (replacing v_i by r_i):

$$\begin{aligned} d_i &= r_i + \sum_{k=0}^{i-1} \beta_{ik} d_k \\ &= r_i + \beta_{i,i-1} d_{i-1} \end{aligned}$$

So, the new orthogonal direction depends only on the previous orthogonal direction and the current residual. We don't have to store old residuals or search directions. In the sequel, set $\beta_i := \beta_{i,i-1}$.

We have

$$\begin{aligned} d_{i-1} &= r_{i-1} + \beta_{i-1} d_{i-2} \\ (d_{i-1}, r_{i-1}) &= (r_{i-1}, r_{i-1}) + \beta_{i-1}(d_{i-2}, r_{i-1}) \\ &= (r_{i-1}, r_{i-1}) \\ \beta_i &= \frac{1}{\alpha_{i-1}} \frac{(r_i, r_i)}{(Ad_{i-1}, d_{i-1})} = \frac{(r_i, r_i)}{(d_{i-1}, r_{i-1})} \\ &= \frac{(r_i, r_i)}{(r_{i-1}, r_{i-1})} \end{aligned}$$

Conjugate gradients IV - The algorithm

Given initial value u_0, spd matrix A, right hand side b.

$$
\begin{aligned}
d_0 &= r_0 = b - Au_0 \\
\alpha_i &= \frac{(r_i, r_i)}{(Ad_i, d_i)} \\
u_{i+1} &= u_i + \alpha_i d_i \\
r_{i+1} &= r_i - \alpha_i A d_i \\
\beta_{i+1} &= \frac{(r_{i+1}, r_{i+1})}{(r_i, r_i)} \\
d_{i+1} &= r_{i+1} + \beta_{i+1} d_i
\end{aligned}
$$

At the i-th step, the algorithm yields the element from $e_0 + \mathcal{K}_i$ with the minimum energy error.

Theorem The convergence rate of the method is

$$
||e_i||_A \leq 2 \left(\frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1} \right)^i ||e_0||_A
$$

where $\kappa = \frac{\lambda_{max}(A)}{\lambda_{min}(A)}$ is the spectral condition number.

Preconditioning

Let M be spd, and spectrally equivalent to A, and assume that $\kappa(M^{-1}A) << \kappa(A)$.

Let E be such that $M = EE^T$, e.g. its Cholesky factorization. Then, $\sigma(M^{-1}A) = \sigma(E^{-1}AE^{-T})$:

Assume $M^{-1}Au = \lambda u$. We have

$$(E^{-1}AE^{-T})(E^T u) = (E^T E^{-T})E^{-1}Au = E^T M^{-1}Au = \lambda E^T u$$

$\Leftrightarrow$ $E^T u$ is an eigenvector of $E^{-1}AE^{-T}$ with eigenvalue λ.

Preconditioned CG I

Now we can use the CG algorithm for the preconditioned system

$$E^{-1}AE^{-T}\tilde{x} = E^{-1}b$$

with $\tilde{u} = E^T u$

$$\begin{aligned}
\tilde{d}_0 &= \tilde{r}_0 = E^{-1}b - E^{-1}AE^{-T}u_0 \\
\alpha_i &= \frac{(\tilde{r}_i, \tilde{r}_i)}{(E^{-1}AE^{-T}\tilde{d}_i, \tilde{d}_i)} \\
\tilde{u}_{i+1} &= \tilde{u}_i + \alpha_i \tilde{d}_i \\
\tilde{r}_{i+1} &= \tilde{r}_i - \alpha_i E^{-1}AE^{-T}\tilde{d}_i \\
\beta_{i+1} &= \frac{(\tilde{r}_{i+1}, \tilde{r}_{i+1})}{(\tilde{r}_i, \tilde{r}_i)} \\
\tilde{d}_{i+1} &= \tilde{r}_{i+1} + \beta_{i+1}\tilde{d}_i
\end{aligned}$$

Not very practical as we need E

Preconditioned CG II

Assume $\tilde{r}_i = E^{-1} r_i$, $\tilde{d}_i = E^T d_i$, we get the equivalent algorithm

$$
\begin{aligned}
r_0 &= b - Au_0 \\
d_0 &= M^{-1} r_0 \\
\alpha_i &= \frac{(M^{-1} r_i, r_i)}{(Ad_i, d_i)} \\
u_{i+1} &= u_i + \alpha_i d_i \\
r_{i+1} &= r_i - \alpha_i A d_i \\
\beta_{i+1} &= \frac{(M^{-1} r_{i+1}, r_{i+1})}{(r_i, r_i)} \\
d_{i+1} &= M^{-1} r_{i+1} + \beta_{i+1} d_i
\end{aligned}
$$

It relies on the solution of the preconditioning system, the calculation of the matrix vector product and the calculation of the scalar product.

Unsymmetric problems

- By definition, CG is only applicable to symmetric problems.
- The biconjugate gradient (BICG) method provides a generalization:

Choose initial guess x_0, perform

$$r_0 = b - A x_0 \qquad \hat{r}_0 = \hat{b} - \hat{x}_0 A^T$$
$$p_0 = r_0 \qquad \hat{p}_0 = \hat{r}_0$$
$$\alpha_i = \frac{(\hat{r}_i, r_i)}{(\hat{p}_i, A p_i)}$$
$$x_{i+1} = x_i + \alpha_i p_i \qquad \hat{x}_{i+1} = \hat{x}_i + \alpha_i \hat{p}_i$$
$$r_{i+1} = r_i - \alpha_i A p_i \qquad \hat{r}_{i+1} = \hat{r}_i - \alpha_i \hat{p}_i A^T$$
$$\beta_i = \frac{(\hat{r}_{i+1}, r_{i+1})}{(\hat{r}_i, r_i)}$$
$$p_{i+1} = r_{i+1} + \beta_i p_i \qquad \hat{p}_{i+1} = \hat{r}_{i+1} + \beta_i \hat{p}_i$$

The two sequences produced by the algorithm are biorthogonal, i.e., $(\hat{p}_i, A p_j) = (\hat{r}_i, r_j) = 0$ for $i \neq j$.

Unsymmetric problems II

- BiCG is very unstable and additionally needs the transposed matrix vector product, it is seldomly used in practice
- There is as well a preconditioned variant of BiCG which also needs the transposed preconditioner.
- Main practical approaches to fix the situation:
  - "Stabilize" BiCG $\rightarrow$ BiCGstab (H. Van der Vorst, 1992)
  - tweak CG $\rightarrow$ "Conjugate gradients squared" (CGS, Sonneveld, 1989)
  - Error minimization in Krylov subspace $\rightarrow$ "Generalized Minimum Residual" (GMRES, Saad/Schulz, 1986)
- Both CGS and BiCGstab can show rather erratic convergence behavior
- For GMRES one has to keep the full Krylov subspace, which is not possible in practice $\Rightarrow$ restart strategy.
- From my experience, BiCGstab is a good first guess

Next steps

- Put linear solution methods into our toolchest for solving PDE problems test them later in more interesting 2D situations
- Need more "tools":
  - visualization
  - triangulation of polygonal domains
  - finite element, finite volume discretization methods

Visualization in Scientific Computing

- Human perception much better adapted to visual representation than to numbers
- Visualization of computational results necessary for the development of understanding
- Basic needs: curve plots etc
  - python/matplotlib
- Advanced needs: Visualize discretization grids, geometry descriptions, solutions of PDEs
  - Visualization in Scientific Computing: paraview
  - Graphics hardware: GPU
  - How to program GPU: OpenGL
  - vtk

Python

- Scripting language with huge impact in Scientific Computing
- Open Source, exhaustive documentation online
  - https://docs.python.org/3/
  - https://www.python.org/about/gettingstarted/
- Possibility to call C/C++ code from python
  - Library API
  - swig - simple wrapper and interface generator (not only python)
  - pybind11 - C++11 specific
- Glue language for projects from different sources
- Huge number of libraries
- numpy/scipy
  - Array + linear algebra library implemented in C
- matplotlib: graphics library
  https://matplotlib.org/users/pyplot_tutorial.html

C++/matplotlib workflow

- Run C++ program
- Write data generated during computations to disk
- Use python/matplotlib for to visualize results
- Advantages:
  - Rich possibilities to create publication ready plots
  - Easy to handle installation (write your code, install python+matplotlib)
  - Python/numpy to postprocess calculated data
- Disadvantages
  - Long way to in-depth understanding of API
  - Slow for large datasets
  - Not immediately available for creating graphics directly from C++

Matplotlib: Alternative tools

- Similar workflow
  - gnuplot
  - Latex/tikz
- Call graphics from C++ ?
  - ???
  - Best shot: call C++ from python, return data directly to python
  - Send data to python through UNIX pipes
  - Link pyton interpreter into C++ code
- Faster graphics ?

Processing steps in visualization

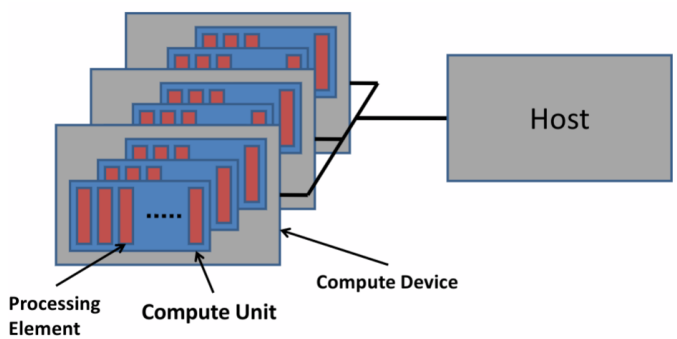
- Representation of data using elementary primitives: points, lines, triangles, ...
- Coordinate transformation form world coordinates to screen coordinates
- Transformation 3D $\rightarrow$ 2D - what is visible ?
- Rasterization: smooth data into pixels
- Coloring, lighting, transparency
- Similar tasks in CAD, gaming etc.
- Huge number of very similar operations

GPU aka “Graphics Card”

- SIMD parallelism “Single instruction, multiple data” inherent to processing steps in visualization
- Mostly `float` (32bit) accuracy is sufficient
- $\Rightarrow$ Create specialized coprocessors devoted to this task, free CPU from it
- Pionieering: Silicon Graphics (SGI)
- Today: nvidia, AMD
- Multiple parallel pipelines, fast memory for intermediate results

GPU Programming

- As GPU is a different processor, one needs to write extra programs to handle data on it – "shaders"
- Typical use:
  - Include shaders as strings in C++ (or load then from extra source file)
  - Compile shaders
  - Send compiled shaders to GPU
  - Send data to GPU – critical step for performance
  - Run shaders with data
- OpenGL, Vulkan



GPU Programming as it used to be

- Specify transformations
- Specify parameters for lighting etc.
- Specify points, lines etc. via API calls
- Graphics library sends data and manages processing on GPU
- No shaders - “fixed functions”
- Iris GL (SGI), OpenGL 1.x, now deprecated
- No simple, standardized API for 3D graphics with equivalent functionality
- Hunt for performance (gaming)

vtk

https://www.vtk.org/

- Visualization primitives in scientific computing
  - Datasets on rectangular and unstructured discretization grids
  - Scalar data
  - Vector data
- The *Visualization Toolkit* vtk provides an API with these primitives and uses up-to data graphics API (OpenGL) to render these data
- Well maintained, "working horse" in high performance computing
- Open Source
- Paraview, VisIt: GUI programs around vtk

Working with paraview

https://www.paraview.org/

- Write data into files using vtk specific data format
- Call paraview, load data

In-Situ visualization

- Using “paraview catalyst”
  - Send data via network from simulation server to desktop running paraview
- Call vtk API directly
  - vtkfig: small library for graphics primitives compatible with numcxx