

Software Lab Computational Engineering Science

Vorstellung Aufgabe 9

Gruppe 2

Ghina Aldardari, Gidon Bauer, Stefan Basermann und Stefan Berger

Informatik 12: Software and Tools for Computational Engineering (STCE)
RWTH Aachen University

Vorwort

Analyse

- Benutzeranforderungen
- Systemanforderungen

Design

- Verwendete Software
- Klassen Modelle

Implementation

- Quellcode
- Software Tests
- Applikationen

Live Software Demo

Projekt Management

Laufzeitanalyse

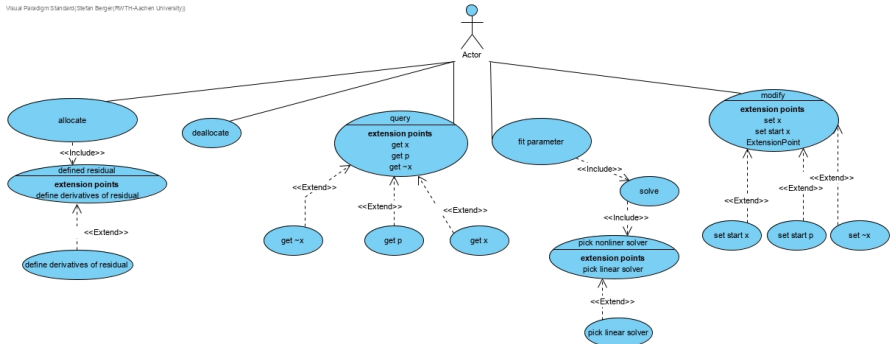
Zusammenfassung und Fazit

- ▶ Wir haben ein nicht-lineares System gegeben und zusätzlich einen Beobachtungswert $\tilde{\mathbf{x}}$
- ▶ Wir möchten die freien Parameter \mathbf{p} so anpassen, dass die Fehlerquadrate $\|\mathbf{x} - \tilde{\mathbf{x}}\|_2^2$ minimiert werden.

Erstellen einer Software Bibliothek, welche es dem Nutzer erlaubt die Parameter \mathbf{p} eines nichtlinearen Systems zu modifizieren, sodass die Fehlerquadrate minimiert werden.

- ▶ Erweitern der NONLINEAR_SYSTEM Bibliothek
- ▶ $\mathbf{x}, \tilde{\mathbf{x}} \in \mathbb{R}^{n_s}$ und $\mathbf{p} \in \mathbb{R}^{n_p}$ eingeben, ausgeben und speichern
- ▶ Nutzung eines einfachen Gradientenverfahrens
- ▶ Benutzung von dco/c++ für die Berechnung des Gradienten des Ziels im Bezug auf \mathbf{p} .
- ▶ Entwurf eines skalierbaren (in der Anzahl der freien Parameter \mathbf{p}) Beispiels für Laufzeit Experimente
- ▶ Nutzung auf dem RWTH-Cluster

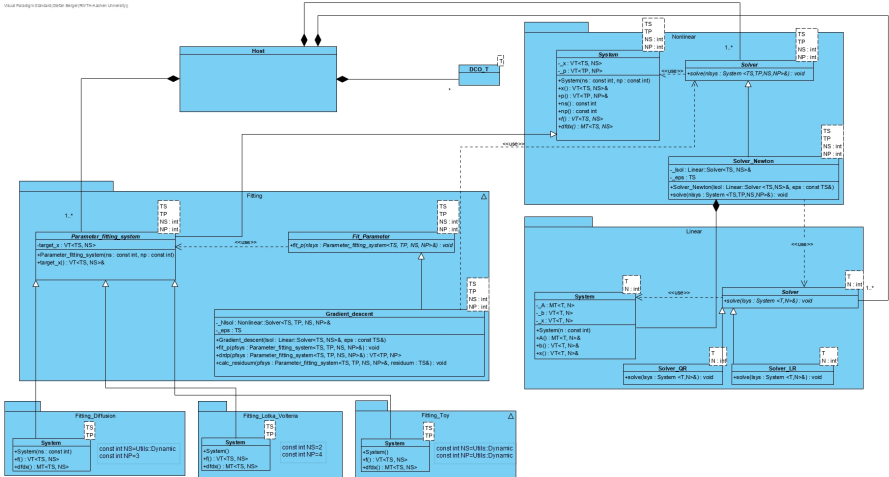
Visual Paradigm Standard (Stefan Berger (RWTH Aachen University))

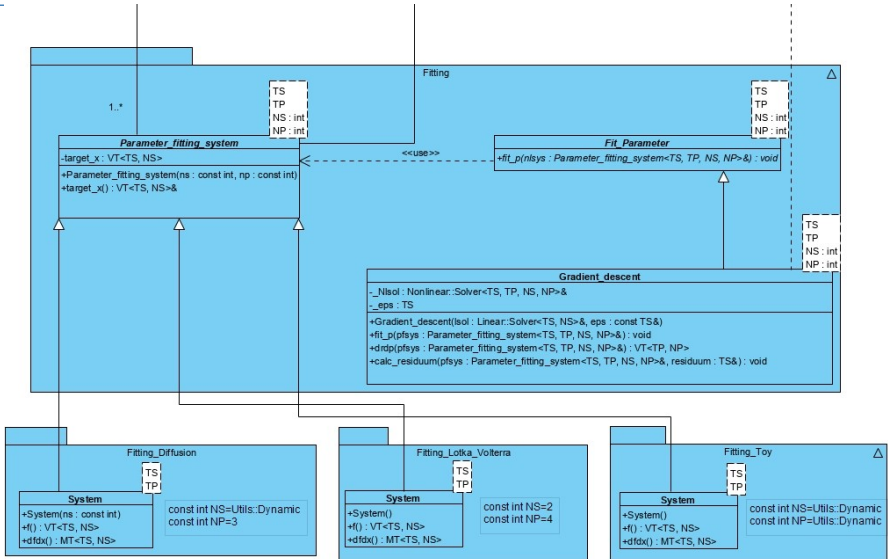


- ▶ "parameter fitting system"
 - ▶ Allokieren
 - ▶ Deallokieren
 - ▶ lese/schreib Zugriff \mathbf{x} , $\tilde{\mathbf{x}}$ und \mathbf{p}
 - ▶ Generische Typen für \mathbf{x} , $\tilde{\mathbf{x}}$ und \mathbf{p}
 - ▶ Implementierung des entsprechenden Residuums und dessen Ableitung
- ▶ "fit parameter"
 - ▶ Allokieren
 - ▶ Deallokieren
 - ▶ Anpassung der Parameter mittels Gradientenverfahren
 - ▶ Generische Typen für \mathbf{x} , $\tilde{\mathbf{x}}$ und \mathbf{p}

- ▶ "dco/c++": Berechnung der Ableitungen
- ▶ "Eigen": Nutzung der Vektor- und Matrixtypen
- ▶ Bibliotheken für (nicht-)lineare Systeme

Visual Modeling Standards Center (RWTH Aachen University)





Makefile

doc

```
|-- Doxyfile  
'-- Makefile
```

include

```
|-- fit_parameter.hpp  
|-- gradient_descent.hpp  
'-- parameter_fitting_system.hpp
```

src

```
|-- gradient_descent.cpp  
'-- parameter_fitting_system.cpp
```

```
1 namespace Fitting
2 {
3
4     template <typename TS, typename TP, int NS, int NP>
5     class Parameter_fitting_system : public Nonlinear::System<TS, TP, NS, NP>
6     {
7     public:
8         using VTS = Eigen::Matrix<TS, NS, 1>;
9         using MTS = Eigen::Matrix<TS, NS, NS>;
10
11     protected:
12         VTS _target_x;
13
14     public:
15         Parameter_fitting_system(int const, int const);
16         VTS &target_x();
17     };
18
19 }
20
21 #include "../src/parameter_fitting_system.cpp"
```

```
1 namespace Fitting
2 {
3
4     template <typename TS, typename TP, int NS, int NP>
5     Parameter_fitting_system<TS, TP, NS, NP>::Parameter_fitting_system(int const ns, int
6         const np) : Nonlinear::System<TS, TP, NS, NP>(ns, np), _target_x(ns) {}
7
8     template <typename TS, typename TP, int NS, int NP>
9     typename Parameter_fitting_system<TS, TP, NS, NP>::VTS &
10     Parameter_fitting_system<TS, TP, NS, NP>::target_x()
11     {
12         return _target_x;
13     }
14 }
```

```
1 namespace Fitting {  
2  
3 template<typename TS, typename TP, int NS, int NP>  
4 class Fit_parameter{  
5     public:  
6     virtual void fit_p(Parameter_fitting_system<TS,TP,NS,NP>&) =0;  
7 };  
8  
9 }
```

```
1 namespace Fitting
2 {
3     template <typename TS, typename TP, int NS, int NP>
4     class Gradient_descent : public Fit_parameter<TS, TP, NS, NP>
5     {
6         Nonlinear::Solver<TS, TP, NS, NP> &_Nlsol;
7         TS _eps;
8
9     public:
10         Gradient_descent(Nonlinear::Solver<TS, TP, NS, NP> &, const TS &);
11
12         void fit_p(Parameter_fitting_system<TS, TP, NS, NP> &);
13
14         typename Nonlinear::System<TS, TP, NS, NP>::VTP
15         drdp(Parameter_fitting_system<TS, TP, NS, NP> &);
16
17         void calc_residuum(Parameter_fitting_system<TS, TP, NS, NP> &, TS &);
18     };
19
20 }
21
22 #include "../src/gradient_descent.cpp"
```

```
1 namespace Fitting
2 {
3
4     template <typename TS, typename TP, int NS, int NP>
5     Gradient_descent<TS, TP, NS, NP>::Gradient_descent(
6         Nonlinear::Solver<TS, TP, NS, NP> &Nlsol, const TS &eps)
7         : _Nlsol(Nlsol), _eps(eps) {}
8
9     template <typename TS, typename TP, int NS, int NP>
10    void Gradient_descent<TS, TP, NS, NP>::calc_residuum(
11        Parameter_fitting_system<TS, TP, NS, NP> &pfsys, TS &residuum)
12    {
13        _Nlsol.solve(pfsys);
14        residuum = pow(((pfsys.x() - pfsys.target_x()).norm()), 2);
15    }
16    // ...
```

```
1 // ...
2 template <typename TS, typename TP, int NS, int NP>
3 typename Nonlinear::System<TS, TP, NS, NP>::VTP
4 Gradient_descent<TS, TP, NS, NP>::drdp(Parameter_fitting_system<TS, TP, NS, NP>
    &pfsys)
5 {
6     typename Nonlinear::System<TS, TP, NS, NP>::VTP grad(pfsys.np());
7     typename Nonlinear::System<TS, TP, NS, NP>::VTS last;
8     TS residuum;
9     for (int i = 0; i < pfsys.np(); i++)
10    {
11        last = pfsys.x();
12        dco::derivative(pfsys.p()(i)) = 1;
13        calc_residuum(pfsys, residuum);
14        grad(i) = dco::derivative(residuum);
15        dco::derivative(pfsys.p()(i)) = 0;
16        pfsys.x() = last;
17    }
18    calc_residuum(pfsys, residuum);
19    return grad;
20 }
21 // ...
```

```
1 // ...
2 template <typename TS, typename TP, int NS, int NP>
3 void Gradient_descent<TS, TP, NS, NP>::fit_p(
4     Parameter_fitting_system<TS, TP, NS, NP> &pfsys)
5 {
6     int iteration = 0;
7     TS residuum;
8     TS residuum_prev;
9     auto last_p = pfsys.p();
10    typename Nonlinear::System<TS, TP, NS, NP>::VTP grad = drdp(pfsys);
11
12    calc_residuum(pfsys, residuum);
13
14    while (grad.norm() > _eps)
15    {
16        calc_residuum(pfsys, residuum_prev);
17        double alpha = 2;
18        // ...
```

```
1 // ...
2 while (residuum_prev <= residuum && alpha >= _eps)
3 {
4     alpha /= 2;
5
6     auto p_trial = last_p;
7     p_trial -= alpha * grad;
8
9     pfsys.p() = p_trial;
10
11     calc_residuum(pfsys, residuum);
12     iteration++;
13 }
14 pfsys.p() = last_p;
15 calc_residuum(pfsys, residuum);
16 pfsys.p() = last_p - alpha * grad;
17 last_p = pfsys.p();
18
19 grad = drdp(pfsys);
20 }
21 }
22 } // namespace Fitting
```

- ▶ Unit Tests für die Teilroutinen ("calc_residuum", "drdp", etc.)
- ▶ Testapplikationen
 - ▶ Fitting_toy
 - ▶ Fitting_diffusion
 - ▶ Fitting_lotka_volterra

Makefile

doc

- |— Doxyfile
- '— Makefile

include

- |— Fitting_diffusion.hpp
- |— Fitting_lotka_volterra.hpp
- '— Fitting_toy.hpp

src

- |— Fitting_diffusion.cpp
- |— Fitting_lotka_volterra.cpp
- '— Fitting_toy.cpp

toy.cpp

diffusion.cpp

lotka_volterra.cpp

```
1 namespace Parameter_fitting_Toy {
2     const int NS = Utils::Dynamic;
3     const int NP = Utils::Dynamic;
4
5     template <typename TS, typename TP>
6     class System : public Fitting::Parameter_fitting_system<TS, TP, NS, NP> {
7         using Nonlinear::System<TS, TP, NS, NP>::x;
8         using Nonlinear::System<TS, TP, NS, NP>::p;
9     public:
10         System(int);
11         typename Nonlinear::System<TS, TP, NS, NP>::VTS f();
12         typename Nonlinear::System<TS, TP, NS, NP>::MTS dfdx();
13     };
14 } // namespace Parameter_fitting_Toy
15 #include "../src/Fitting_toy.cpp"
```

```
1 namespace Parameter_fitting_Toy {
2     template <typename TS, typename TP>
3     System<TS, TP>::System(int n) : Fitting::Parameter_fitting_system<TS, TP, NS, NP>(n
4         , n) {}
5
6     template <typename TS, typename TP>
7     static inline typename Nonlinear::System<TS, TP, NS, NP>::VTS
8     F(const typename Nonlinear::System<TS, TP, NS, NP>::VTS &x, const typename
9         Nonlinear::System<TS, TP, NS, NP>::VTP &p) {
10         typename Nonlinear::System<TS, TP, NS, NP>::VTS r(x.size());
11         for (int i = 0; i < x.size(); ++i) r(i) = pow(p(i), 2) + x(i);
12         return r;
13     }
14
15     template <typename TS, typename TP>
16     typename Nonlinear::System<TS, TP, NS, NP>::VTS
17     System<TS, TP>::f() { return F<TS, TP>(-x, -p); }
18     // ...
```

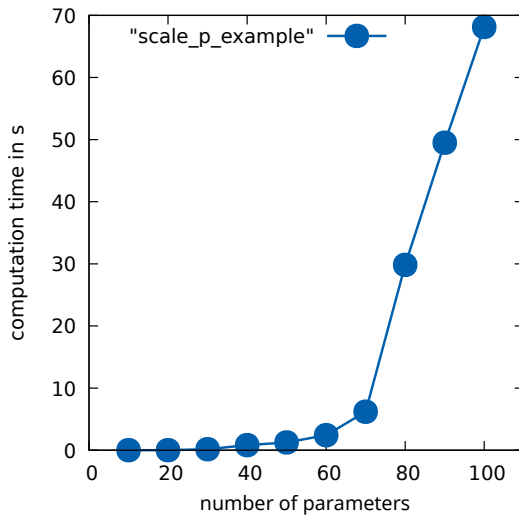
```

1 // ...
2 template <typename TS, typename TP>
3 typename Nonlinear::System<TS, TP, NS, NP>::MTS
4 System<TS, TP>::dfdX() {
5     typename Nonlinear::System<TS, TP, NS, NP>::MTS drdx(_x.size(), _x.size());
6     using DCO_T = typename dco::gt1s<TS>::type;
7     typename Nonlinear::System<DCO_T, TP, NS, NP>::VTS x_t(_x.size()), r_t(_x.size());
8     for (auto i = 0; i < _x.size(); i++) dco::value(x_t(i)) = _x(i);
9     for (auto i = 0; i < _x.size(); i++) {
10         dco::derivative(x_t(i)) = 1;
11         r_t = F<DCO_T, TP>(x_t, _p);
12         for (auto j = 0; j < _x.size(); j++) drdx(j, i) = dco::derivative(r_t(j));
13         dco::derivative(x_t(i)) = 0;
14     }
15     return drdx;
16 }
17 } // namespace Parameter_fitting_Toy
    
```

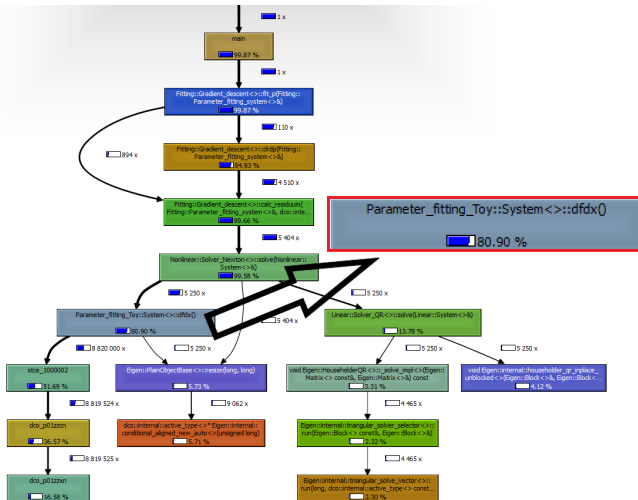
```
1 int main(int argc, char* argv[]) {
2     assert(argc == 2);
3     const int n = std::stoi(argv[1]);
4     assert(n > 0);
5     using T = double;
6     using DCO_T = dco::gtls<T>::type;
7
8     //Erzeugug System + Solver
9
10    Parameter_fitting_Toy::System<DCO_T, DCO_T> pfsys(n);
11    Linear::Solver_QR<DCO_T, Parameter_fitting_Toy::NS> Isol;
12    Nonlinear::Solver_Newton<DCO_T, DCO_T, Parameter_fitting_Toy::NS,
13        Parameter_fitting_Toy::NP> nlsol(Isol, 1e-7);
14    Fitting::Gradient_descent<DCO_T, DCO_T, Parameter_fitting_Toy::NS,
15        Parameter_fitting_Toy::NP> pfsol(nlsol, 1e-7);
16
17    //Zufällige Startwertgenerierung
18
19    std::default_random_engine generator(std::random_device{}());
20    std::uniform_real_distribution<T> dist_start_x (-10, 0);
21    std::uniform_real_distribution<T> dist_start_p (-5, 5);
22    std::uniform_real_distribution<T> dist_target_x (-10, 0);
```

```
1  for (int i = 0; i < n; ++i) {
2      pfsys.x()(i) = dist_start_x(generator);
3      pfsys.p()(i) = dist_start_p(generator);
4      pfsys.target_x()(i) = dist_target_x(generator);
5  }
6  // Ausgaben der Startwerte
7
8  std::cout << "x_start=\n" << pfsys.x() << std::endl << std::endl << std::endl;
9  std::cout << "x_target=\n" << pfsys.target_x() << std::endl << std::endl << std::endl;
10 std::cout << "p_start=\n" << pfsys.p() << std::endl << std::endl << std::endl;
11
12 pfsol.fit_p(pfsys); //Lösen des Systems
13
14 // Ausgaben der Lösungen
15
16 std::cout << "p_solution=\n" << pfsys.p() << std::endl << std::endl << std::endl;
17 std::cout << "||x-target_x||^2 = " << pow((pfsys.x() - pfsys.target_x()).norm(), 2) <<
    std::endl << std::endl;
18
19 return 0;
20 }
```

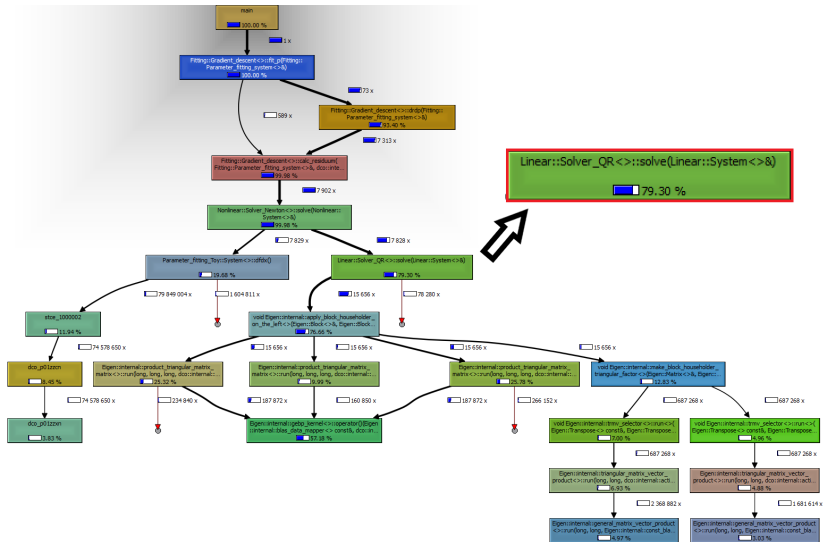

- ▶ Git zur Versionskontrolle
- ▶ Doxygen zur Dokumentation
- ▶ "Pair Programming"



Toy-Applikation 40 p



Toy-Applikation 100 p



- ▶ Cache ist kein Engpass, Trefferrate liegt bei beiden Testläufen bei 99.5%

Zusammenfassung:

- ▶ Erstellen einer neuen Bibliothek
 - ▶ Neues System: Speichert zusätzlich \tilde{x}
 - ▶ Neuer "Solver": Passt p an
- ▶ Drei Applikationen: Lotka-Volterra, Diffusion und Toy

Erweiterungsideen:

- ▶ Anderer Algorithmus zum lösen von linearen Gleichungssystemen
- ▶ Multithreading mittels MPI/OpenMP