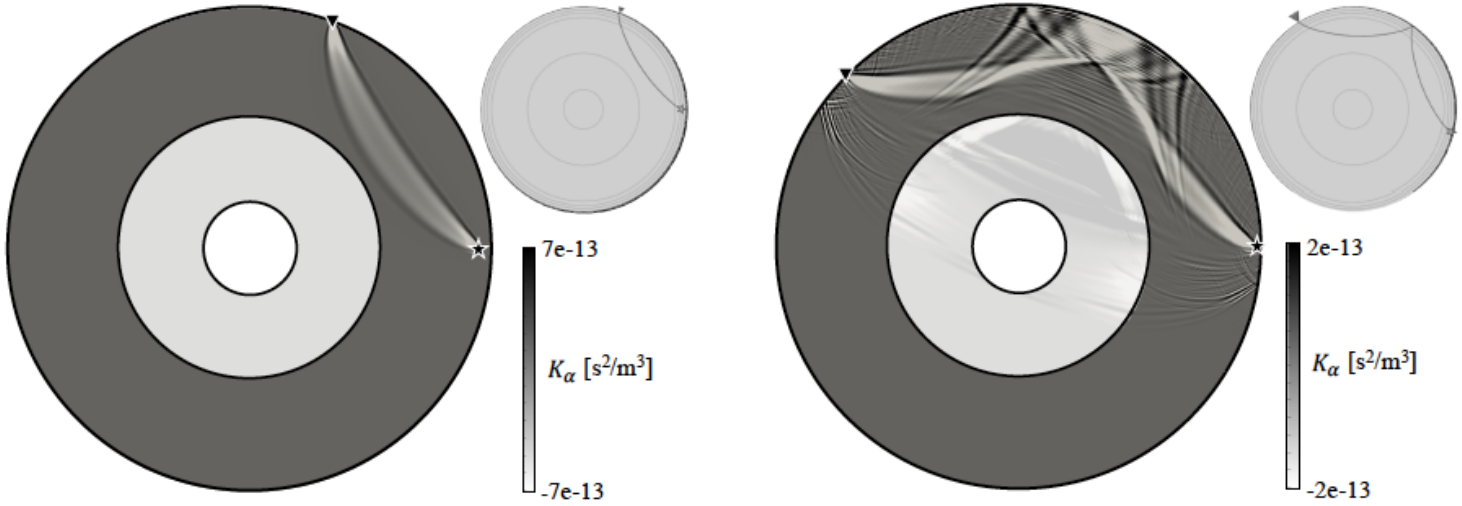
Lecture 2

Non-linear optimisation and adjoint methods

Andreas Fichtner

ETH Zurich - Seismology and Wave Physics



MOTIVATION

- More often than not, the forward model is not linear.
- The beautiful machinery of linear inverse theory is not applicable.
- We still would like to find a model that explains observations acceptably well.

GOALS OF THIS LECTURE

- Introduce basic concepts of non-linear optimisation.
- Cover the most important classes of non-linear, gradient-based optimisation methods.
- Introduce adjoint methods for the efficient computation of misfit gradients.

OUTLINE

PART I: Basic non-linear optimisation concepts

1. Iterative linearisation
2. Local optimisation, global and local minima

PART II: Non-linear optimisation methods

1. General gradient descent methods
2. The method of steepest descent
3. Newton's method
4. Other methods: Conjugate gradients, BFGS, L-BFGS, ...

PART III: Non-linear optimisation Jupyter notebook

PART IV: Adjoint methods

1. Motivation
2. Discrete adjoints
3. Continuous adjoints
4. Sensitivity kernels and kernel gallery

PART V: Adjoint method Jupyter notebook

All of this will closely follow the *Fundamentals of Inverse Theory* lecture notes.

PART I

Basic non-linear optimisation concepts

1. Iterative linearisation

Iterative Linearisation - Concept

- Frequently, the forward problem **d**=**G**(**m**) is not linear.
- The least-squares method develop for linear forward problems is not applicable.
  - No explicit solution for the posterior mean model and posterior covariance.
  - No resolution matrix.
  - ...

- Frequently, the forward problem **d**=**G**(**m**) is not linear.
- The least-squares method develop for linear forward problems is not applicable.
  - No explicit solution for the posterior mean model and posterior covariance.
  - No resolution matrix.
  - ...
- The seemingly most straightforward solution: Pick a prior model $\mathbf{m}_0$ and linearise:

$$\mathbf{G}(\mathbf{m}) \doteq \mathbf{G}(\mathbf{m}_0) + \mathbf{J}_0(\mathbf{m} - \mathbf{m}_0) \qquad J_{0,ij} = \left.\frac{\partial G_i}{\partial m_j}\right|_{\mathbf{m}=\mathbf{m}_0}$$

= Jacobian at $\mathbf{m}_0$

- Now this is a linear problem! We can apply the linear least-squares machinery to obtain a 'solution' $\mathbf{m}_1$.

Iterative Linearisation - Concept

- Frequently, the forward problem **d**=**G**(**m**) is not linear.
- The least-squares method develop for linear forward problems is not applicable.
  - No explicit solution for the posterior mean model and posterior covariance.
  - No resolution matrix.
  - ...
- The seemingly most straightforward solution: Pick a prior model $\mathbf{m}_0$ and linearise:

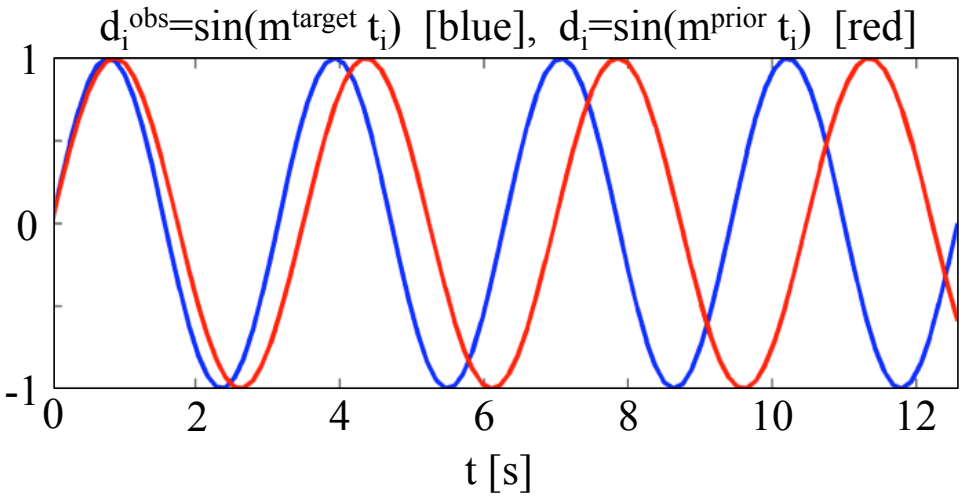
$$\mathbf{G}(\mathbf{m}) \doteq \mathbf{G}(\mathbf{m}_0) + \mathbf{J}_0(\mathbf{m} - \mathbf{m}_0) \qquad J_{0,ij} = \left.\frac{\partial G_i}{\partial m_j}\right|_{\mathbf{m}=\mathbf{m}_0}$$

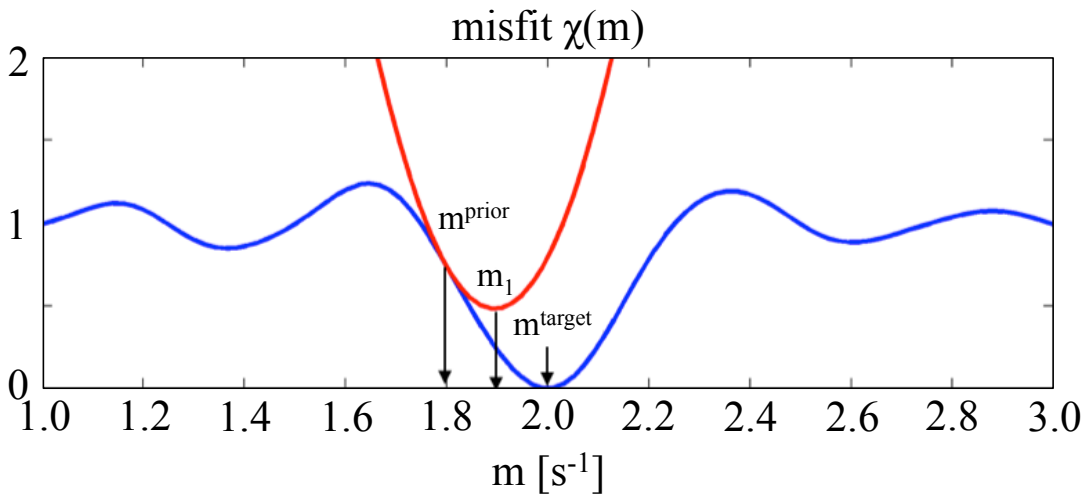
= Jacobian at $\mathbf{m}_0$

- Now this is a linear problem! We can apply the linear least-squares machinery to obtain a 'solution' $\mathbf{m}_1$.
- This can be iterated:
  - Linearise forward problem around $\mathbf{m}_1$ using new Jacobian $\mathbf{J}_1$.
  - Compute new least-squares solution $\mathbf{m}_2$.
  - Repeat ...

- A toy problem:

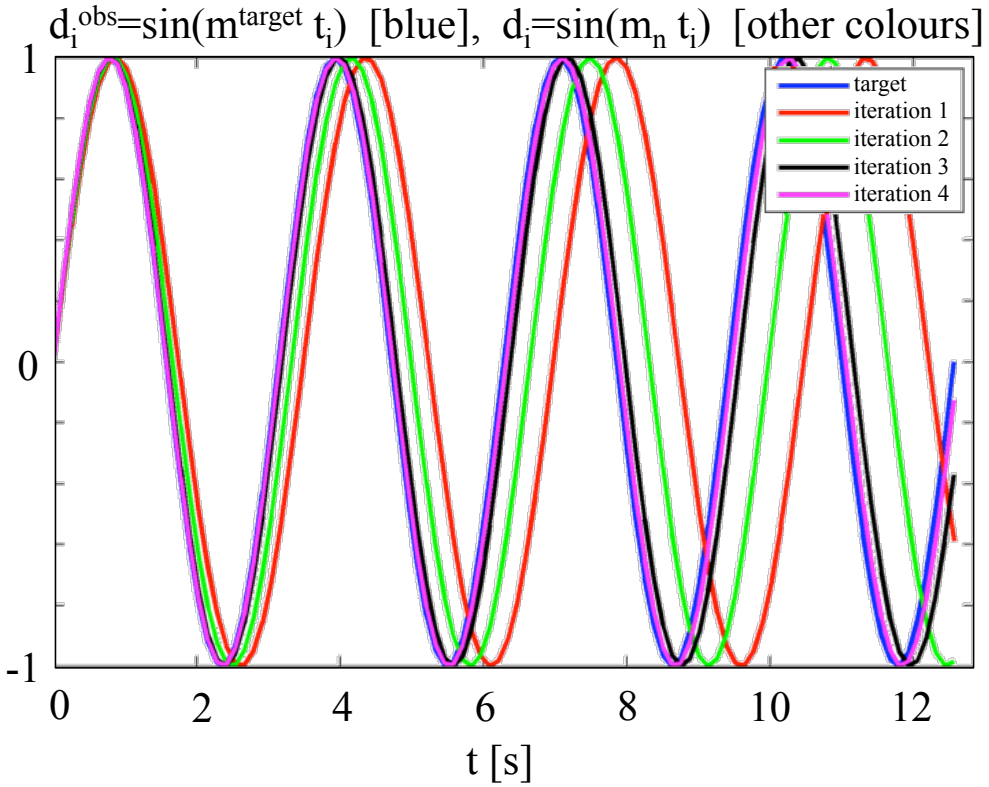
$$d_i = \sin(mt_i) \qquad \chi(m) = \frac{1}{2}\sum_{i=1}^{N}\left[d_i^{\text{obs}} - \sin(mt_i)\right]^2$$

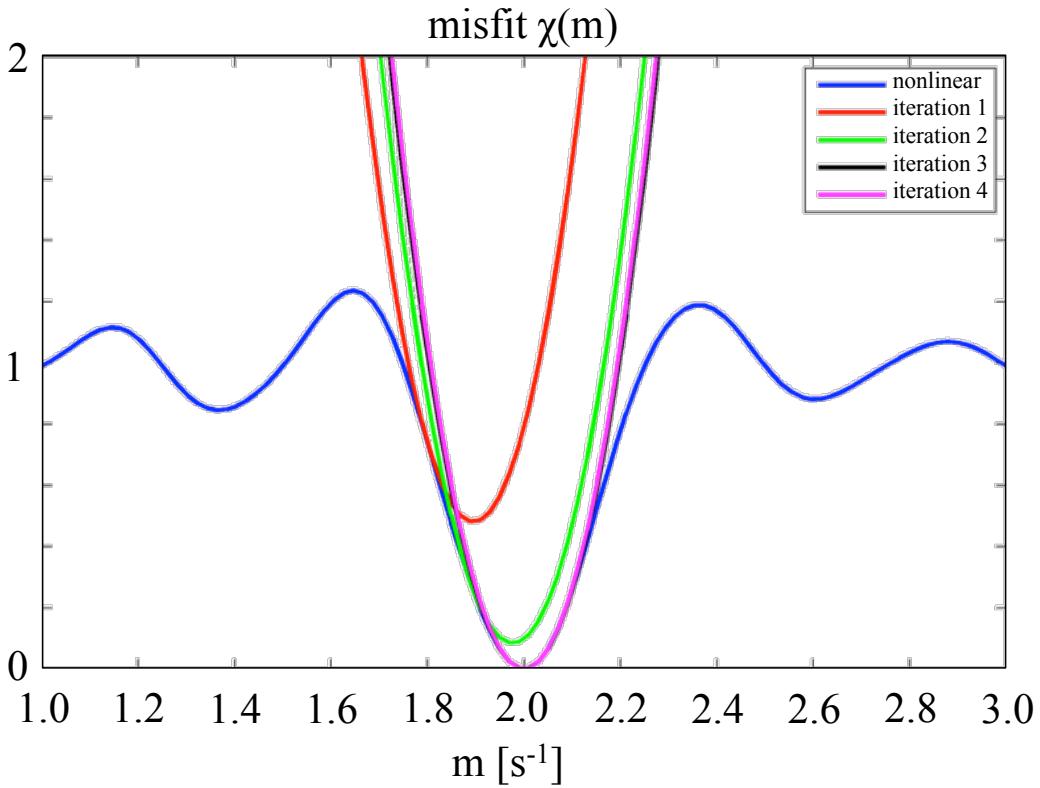




- A toy problem:

$$d_i = \sin(mt_i) \qquad \chi(m) = \frac{1}{2}\sum_{i=1}^{N}\left[d_i^{\text{obs}} - \sin(mt_i)\right]^2$$

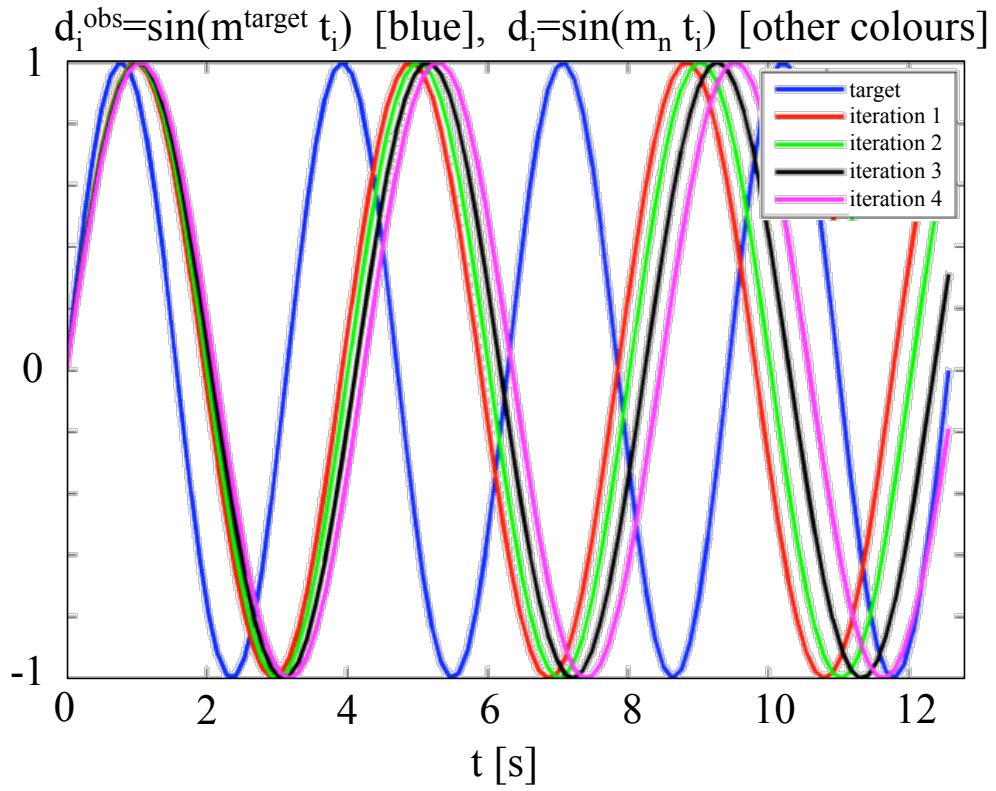


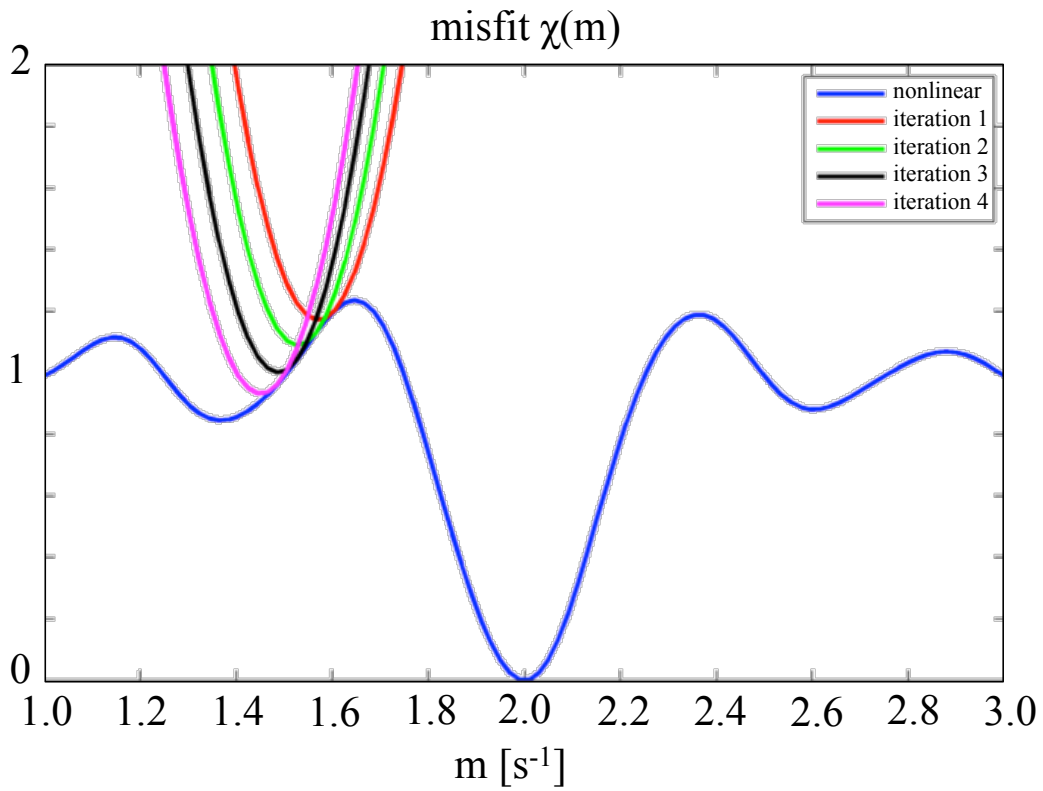


2. Local optimisation, global and local minima

- A toy problem:

$$d_i = \sin(mt_i) \qquad \chi(m) = \frac{1}{2}\sum_{i=1}^{N}\left[d_i^{\text{obs}} - \sin(mt_i)\right]^2$$





- A toy problem:

$$d_i = \sin(mt_i) \qquad \chi(m) = \frac{1}{2}\sum_{i=1}^{N}\left[d_i^{\text{obs}} - \sin(mt_i)\right]^2$$

- Iterative linearisation [and all methods we will see in this lecture] is a local method.
- It may get stuck in a local minimum.
- The 'result' may depend strongly on the choice of the initial model.
- In practice, it may be very hard to check if the minimum is local or global.
  - Analysis of data fit
  - Plausibility arguments
  - Check against independent data

PART II

Non-linear optimisation methods

1. General descent methods

- Our goal: Find a model **m** such that a suitably defined misfit χ is minimal.
- The minimisation proceeds iteratively:

1. Start from initial model $\mathbf{m}_0$
2. Update according to $\mathrm{m}_{i+1} = \mathrm{m}_i + \gamma_i \, \mathrm{h}_i$ with $\chi(m_{i+1}) < \chi(m_i)$

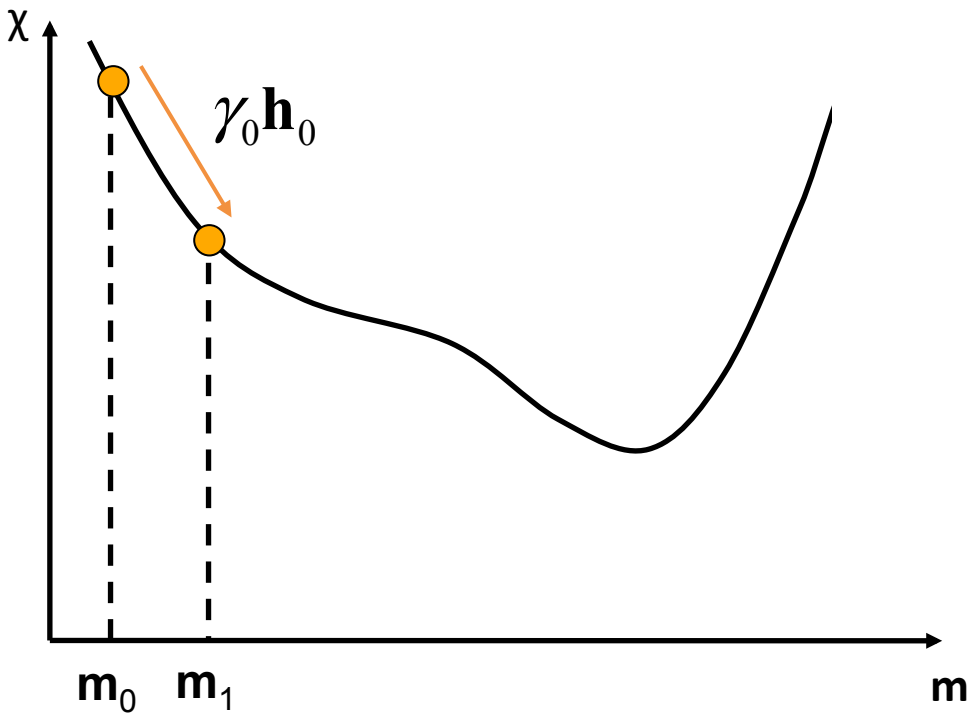
step length — γ_i

descent direction — h_i

- Our goal: Find a model **m** such that a suitably defined misfit χ is minimal.
- The minimisation proceeds iteratively:

1. Start from initial model $\mathbf{m}_0$
2. Update according to $\mathrm{m}_{i+1} = \mathrm{m}_i + \gamma_i \, \mathrm{h}_i$ with $\chi(m_{i+1}) < \chi(m_i)$

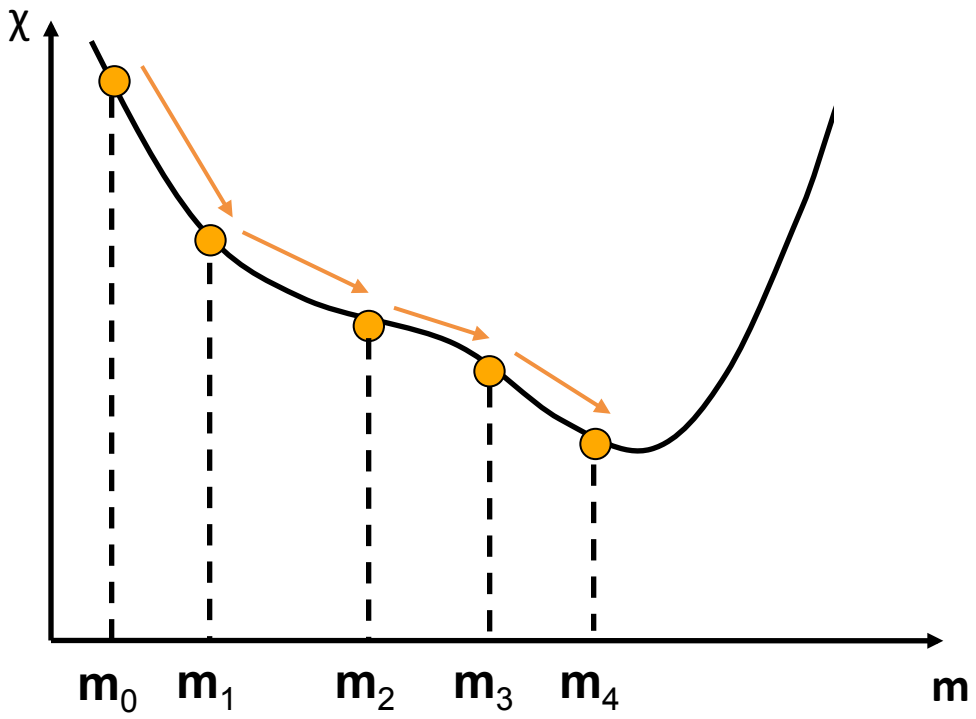
step length — descent direction



- Our goal: Find a model **m** such that a suitably defined misfit χ is minimal.
- The minimisation proceeds iteratively:

1. Start from initial model $\mathbf{m}_0$
2. Update according to $\mathrm{m}_{i+1} = \mathrm{m}_i + \gamma_i \, \mathrm{h}_i$ with $\chi(m_{i+1}) < \chi(m_i)$

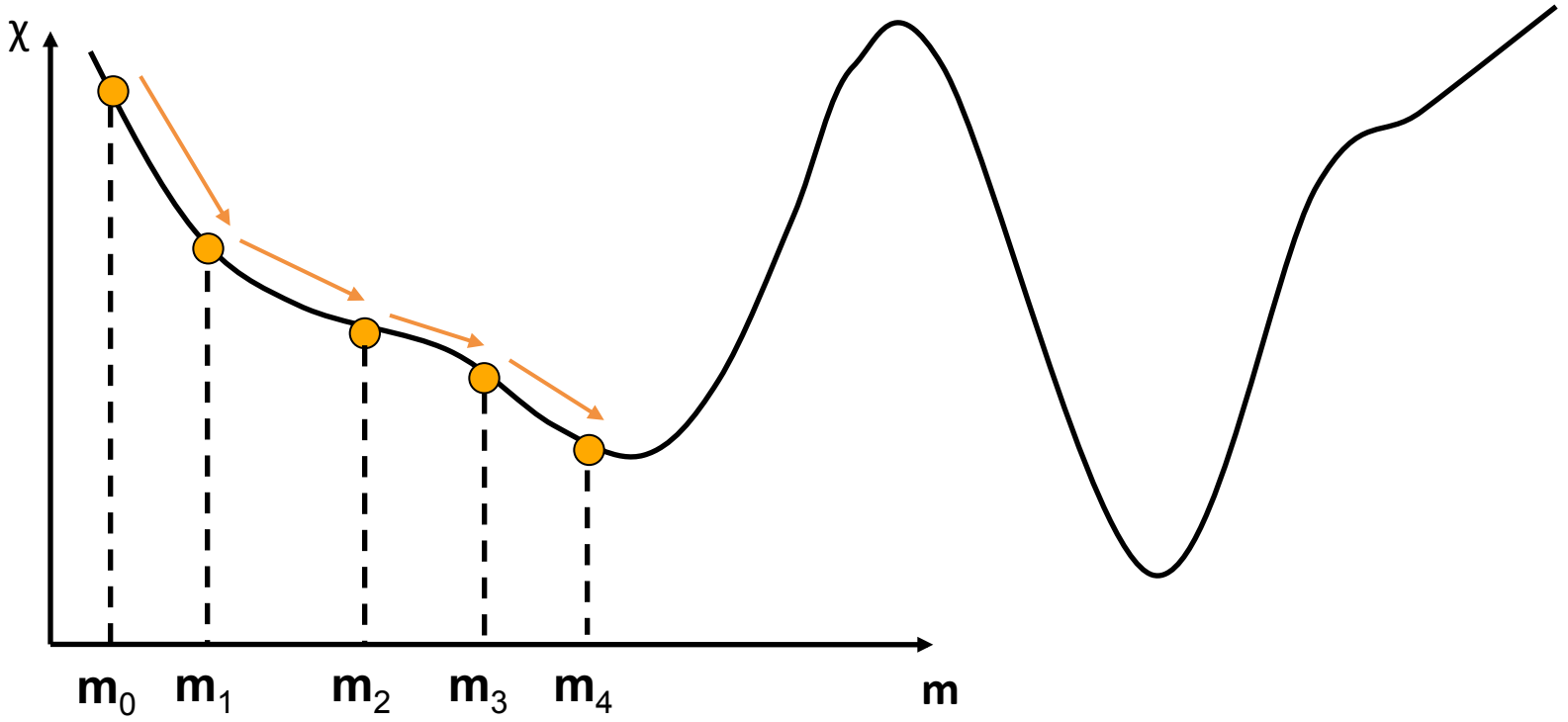
step length — descent direction



- Our goal: Find a model **m** such that a suitably defined misfit χ is minimal.
- The minimisation proceeds iteratively:

1. Start from initial model $\mathbf{m}_0$
2. Update according to $\mathrm{m}_{i+1} = \mathrm{m}_i + \gamma_i \, \mathrm{h}_i$ with $\chi(m_{i+1}) < \chi(m_i)$

step length — descent direction



The General Scheme

- Our goal: Find a model **m** such that a suitably defined misfit χ is minimal.
- The minimisation proceeds iteratively:

  1. Start from initial model $\mathbf{m}_0$
  2. Update according to $\mathrm{m}_{i+1} = \mathrm{m}_i + \gamma_i \, \mathrm{h}_i$ with $\chi(m_{i+1}) < \chi(m_i)$

     step length — descent direction

- A possible descent direction is always: $\mathbf{h}_0 = -\nabla\chi(\mathbf{m}_0)$
- Other possible directions are $\mathbf{h}_0 = -\mathbf{A}\nabla\chi(\mathbf{m}_0)$ for any positive definite matrix **A** [which may change with iteration].
- The members of the descent method family are defined by the choice of **A**.

2. The method of steepest descent

- **A=I** → h is direction of steepest descent

Method of steepest descent

1. Choose an initial model, $\mathbf{m}_0$, and set $i = 0$.

2. Compute the gradient for the current model, $\nabla\chi(\mathbf{m}_i)$.

3. Update $\mathbf{m}_i$ according to

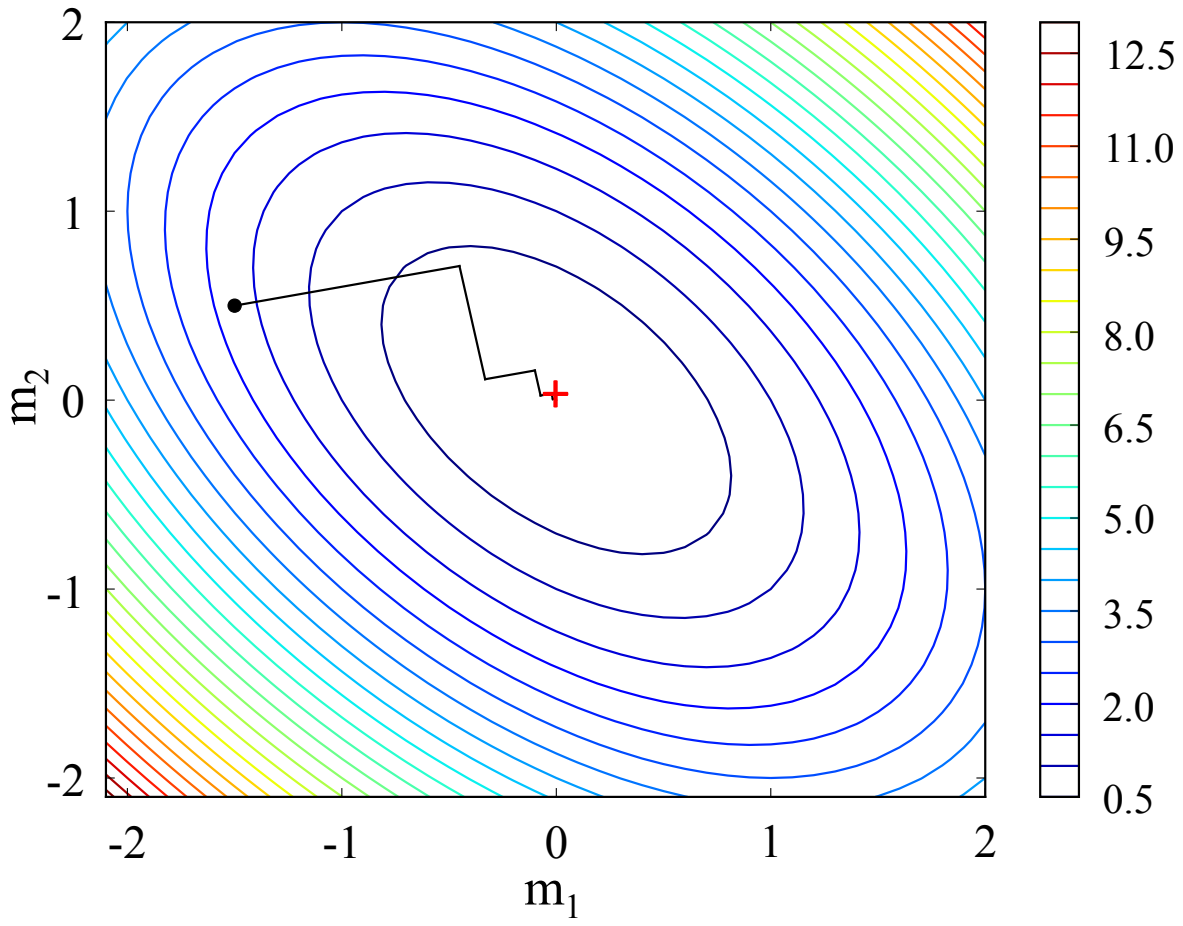
$$\mathbf{m}_{i+1} = \mathbf{m}_i - \gamma_i \nabla\chi(\mathbf{m}_i)\,,$$

with a suitable step length γ_i that ensures $\chi(\mathbf{m}_{i+1}) < \chi(\mathbf{m}_i)$.

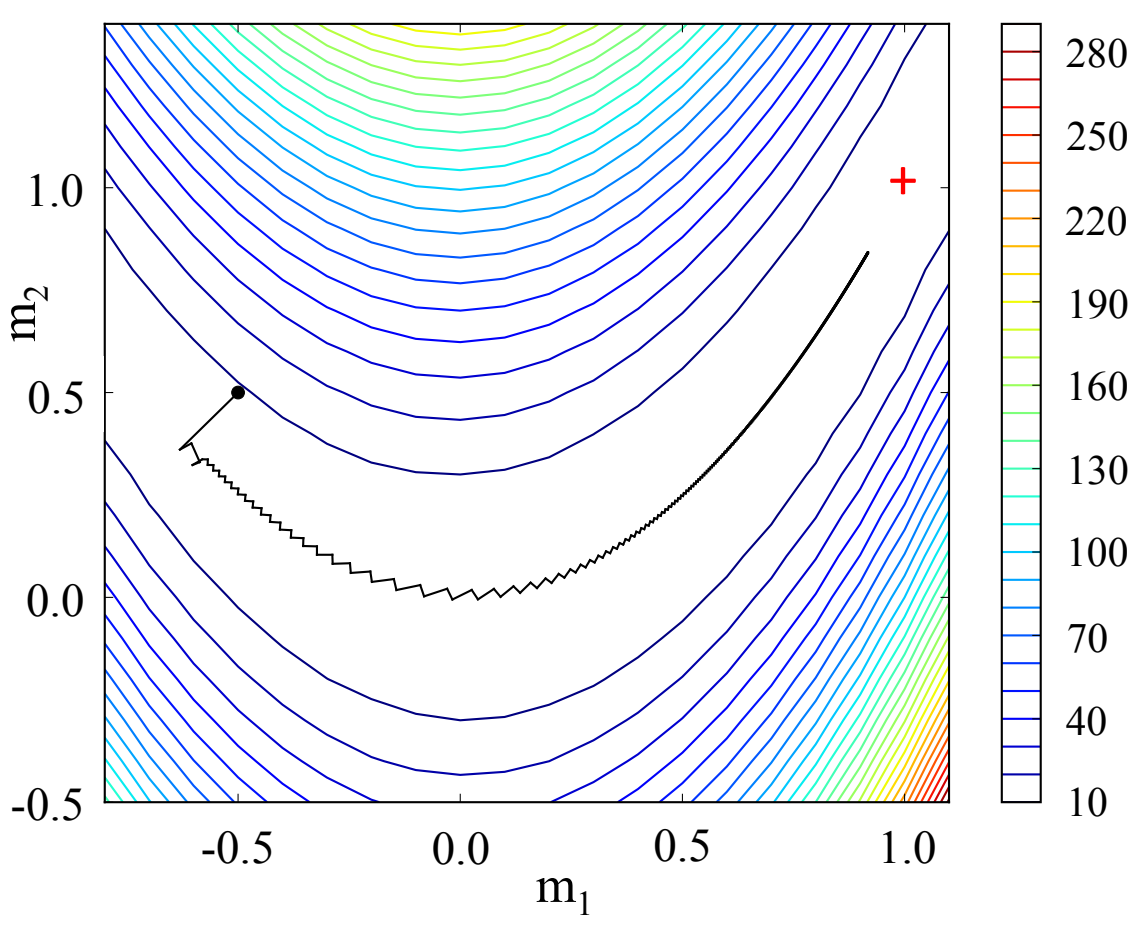
4. Set $i \to i+1$, go back to (2) and repeat until the data are explained sufficiently well.

- **A=I** → h is direction of steepest descent

a) quadratic function



b) Rosenbrock function



Steepest Descent – Pros & Cons

- **A**=**I** → h is direction of steepest descent

\+ Very easy to implement.
\+ Method of choice for testing purposes.

\− Potentially very slow convergence.

3. Newton’s method

- Newton's method is motivated by a Taylor expansion:

$$\mathbf{0} = \nabla\chi(\tilde{\mathbf{m}}) \approx \nabla\chi(\mathbf{m}) + \mathbf{H}(\mathbf{m})\,(\tilde{\mathbf{m}} - \mathbf{m}) \qquad \tilde{\mathbf{m}} \approx \mathbf{m} - \mathbf{H}^{-1}(\mathbf{m})\,\nabla\chi(\mathbf{m})$$

- $\mathbf{A}=\mathbf{H}^{-1}$, where **H** is the Hessian [matrix of second derivatives].

- Newton's method is motivated by a Taylor expansion:

$$\mathbf{0} = \nabla\chi(\tilde{\mathbf{m}}) \approx \nabla\chi(\mathbf{m}) + \mathbf{H}(\mathbf{m})\,(\tilde{\mathbf{m}} - \mathbf{m}) \qquad \tilde{\mathbf{m}} \approx \mathbf{m} - \mathbf{H}^{-1}(\mathbf{m})\,\nabla\chi(\mathbf{m})$$

- $\mathbf{A}=\mathbf{H}^{-1}$, where $\mathbf{H}$ is the Hessian [matrix of second derivatives].

Newton's method

1. Choose an initial model, $\mathbf{m}_0$, and set $i = 0$.
2. Compute the gradient for the current model, $\nabla\chi(\mathbf{m}_i)$.
3. Determine the descent direction, $\mathbf{h}_i$, as the solution of

$$\mathbf{H}(\mathbf{m}_i)\,\mathbf{h}_i = -\nabla\chi(\mathbf{m}_i)\,.$$

4. Update $\mathbf{m}_i$ according to

$$\mathbf{m}_{i+1} = \mathbf{m}_i + \mathbf{h}_i\,.$$

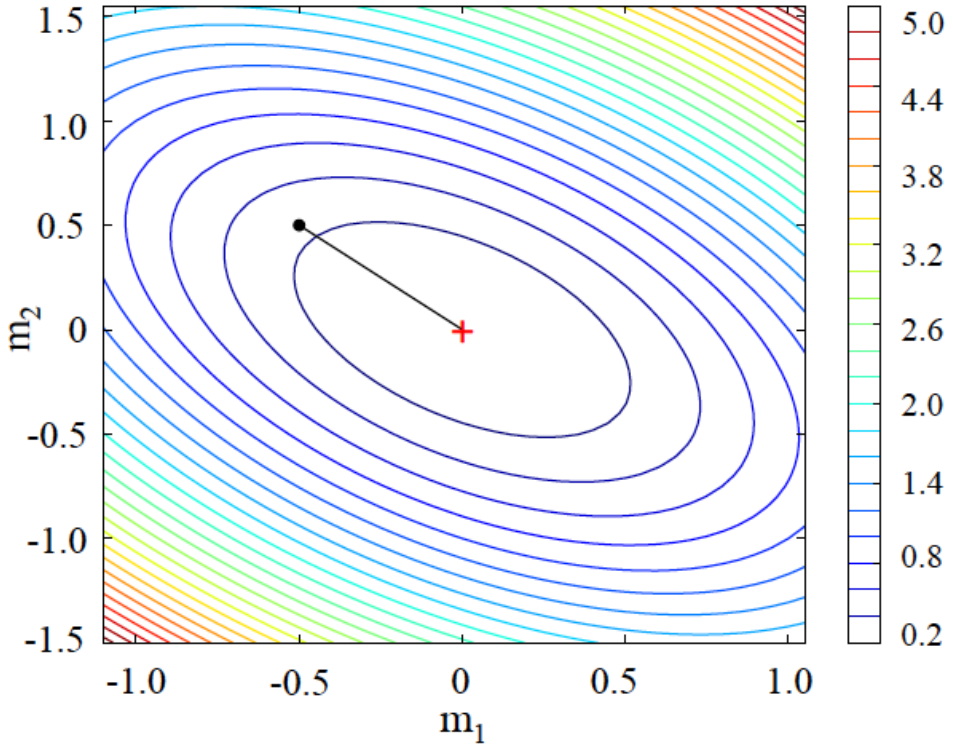
5. Set $i \rightarrow i+1$, go back to (2) and repeat as often as needed.

- Newton's method is motivated by a Taylor expansion:

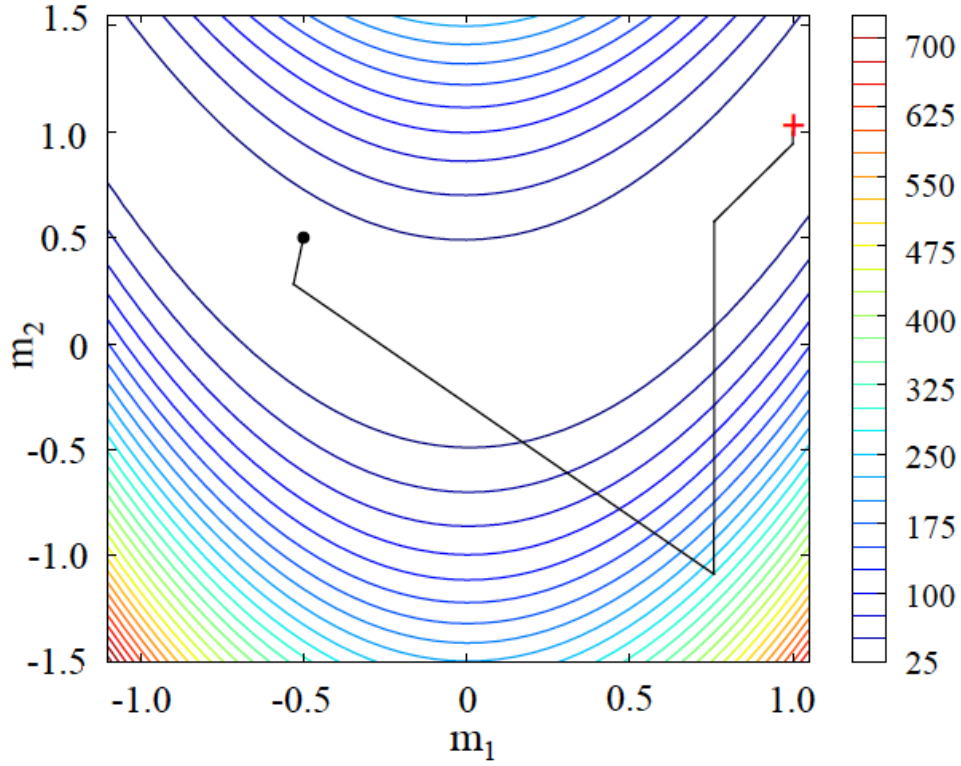
$$\mathbf{0} = \nabla\chi(\tilde{\mathbf{m}}) \approx \nabla\chi(\mathbf{m}) + \mathbf{H}(\mathbf{m})(\tilde{\mathbf{m}} - \mathbf{m}) \qquad \tilde{\mathbf{m}} \approx \mathbf{m} - \mathbf{H}^{-1}(\mathbf{m})\nabla\chi(\mathbf{m})$$

- $\mathbf{A}=\mathbf{H}^{-1}$, where **H** is the Hessian [matrix of second derivatives].

a) quadratic function



b) Rosenbrock function



Newton's Method – Pros & Cons

- Newton's method is motivated by a Taylor expansion:

$$\mathbf{0} = \nabla\chi(\tilde{\mathbf{m}}) \approx \nabla\chi(\mathbf{m}) + \mathbf{H}(\mathbf{m})\,(\tilde{\mathbf{m}} - \mathbf{m}) \qquad \tilde{\mathbf{m}} \approx \mathbf{m} - \mathbf{H}^{-1}(\mathbf{m})\,\nabla\chi(\mathbf{m})$$

- $\mathbf{A}=\mathbf{H}^{-1}$, where **H** is the Hessian [matrix of second derivatives].

\+ Converges rapidly. In 1 iteration, when this misfit is quadratic.

\− Requires computation of second derivatives.
\− Requires inversion of a big matrix that may not be invertible.

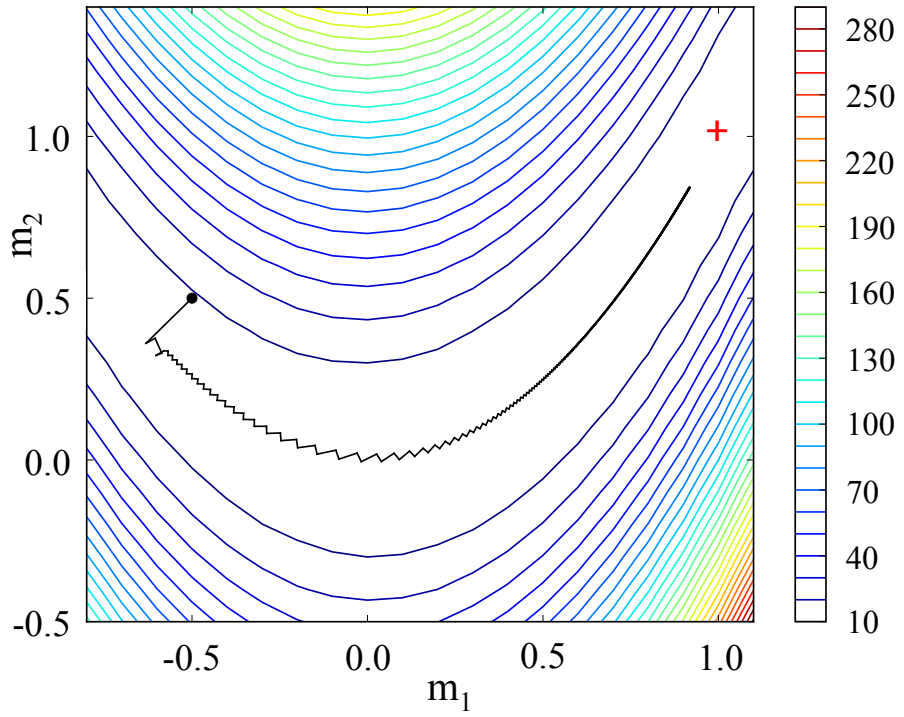
4. Other methods: Conjugate gradients, BFGS, L-BFGS, ...

- Steepest descent and Newton‘s method: end members in a broad spectrum of gradient-based descent methods

- Steepest descent and Newton's method: end members in a broad spectrum of gradient-based descent methods

Conjugate Gradients

- Steepest descent tends to go again and again in a similar direction. → Makes it pretty inefficient.



Steepest-descent minimisation of the Rosenbrock function

- Steepest descent tends to go again and again in a similar direction. → Makes it pretty inefficient.
- Basic idea of conjugate gradients: March in directions that are mutually orthogonal in the sense of $\mathbf{h}_i\mathbf{H}\mathbf{h}_{i+1}=\mathbf{0}$.

- Steepest descent tends to go again and again in a similar direction. → Makes it pretty inefficient.
- Basic idea of conjugate gradients: March in directions that are mutually orthogonal in the sense of $\mathbf{h}_i\mathbf{H}\mathbf{h}_{i+1}=\mathbf{0}$.

- The conjugate-gradient method constructs a sequence of orthogonal descent directions, $\mathbf{h}_0$, $\mathbf{h}_1$, ..., automatically.
- There are different ways of doing that, and each of these ways has its own name:
  - Fletcher-Reves method
  - Hestenes-Stiefel method
  - Polak-Ribiere method

Conjugate Gradients

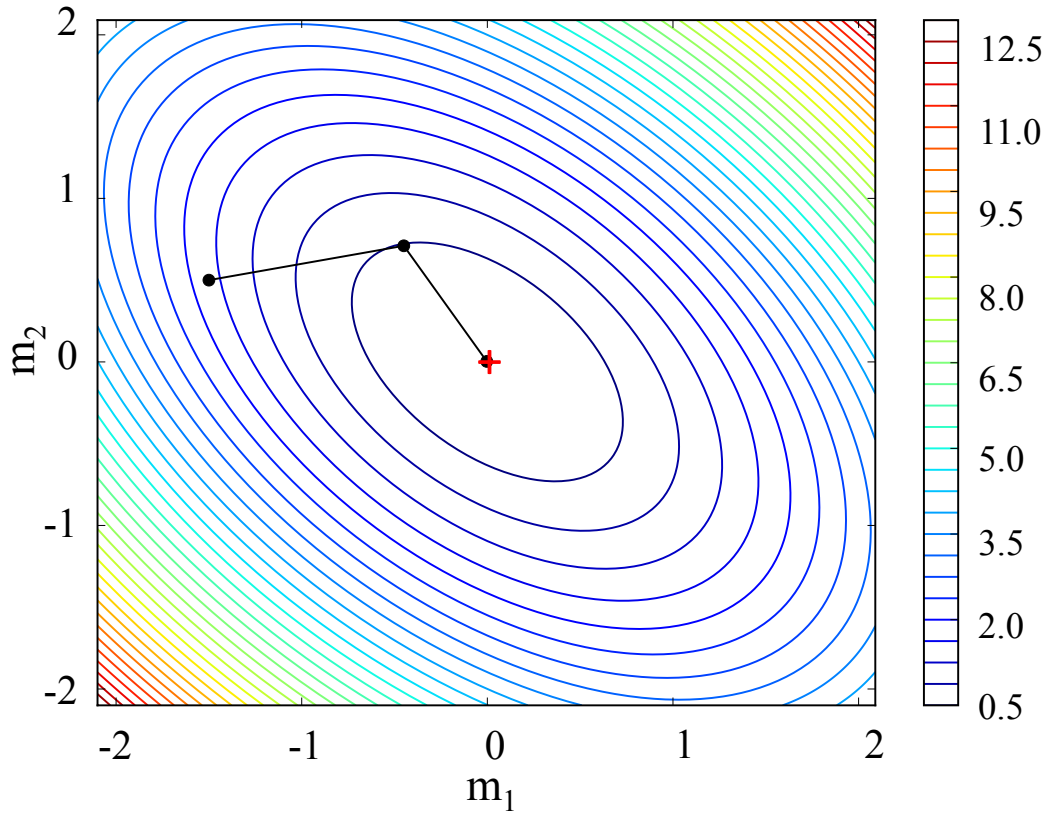
- Steepest descent tends to go again and again in a similar direction. → Makes it pretty inefficient.
- Basic idea of conjugate gradients: March in directions that are mutually orthogonal in the sense of $\mathbf{h}_i\mathbf{H}\mathbf{h}_{i+1}=\mathbf{0}$.
- The conjugate-gradient method constructs a sequence of orthogonal descent directions, $\mathbf{h}_0$, $\mathbf{h}_1$, ..., automatically.
- There are different ways of doing that, and each of these ways has its own name:
  - Fletcher-Reves method
  - Hestenes-Stiefel method
  - Polak-Ribiere method
- The conjugate-gradient method for exactly quadratic misfit functions:
  - Converges in *n* steps at most because:
  - **H** is constant and there are only *n* mutually orthogonal directions.

Conjugate Gradients

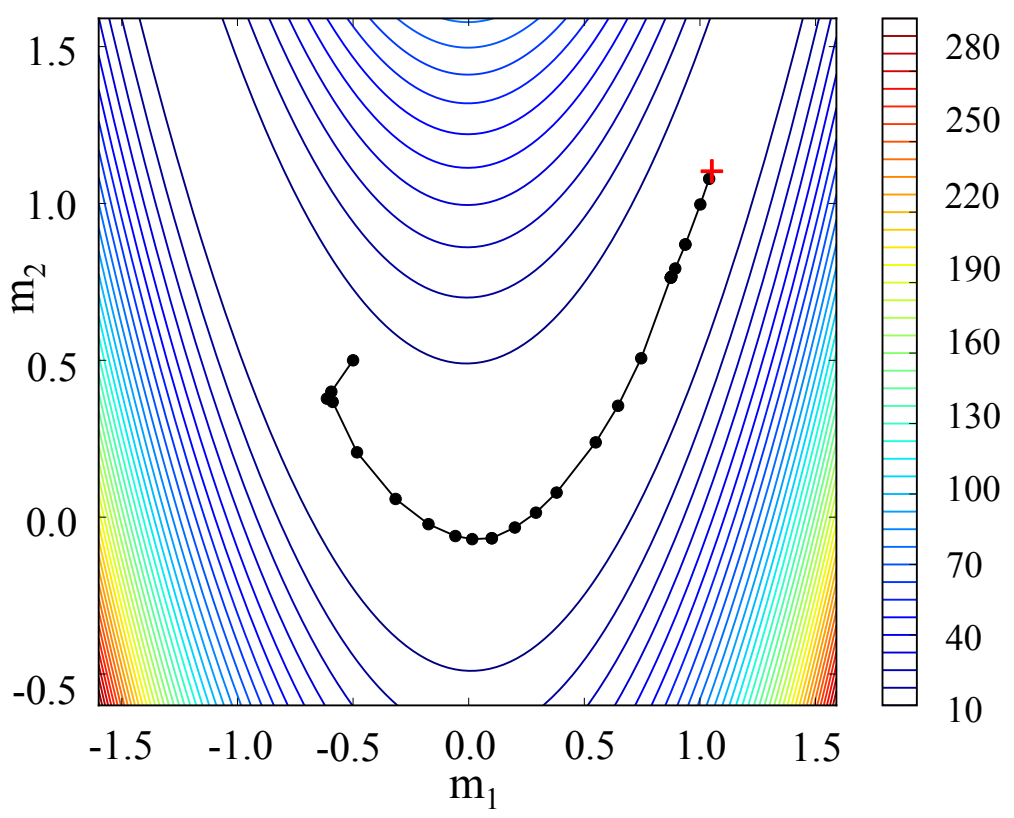
- Steepest descent tends to go again and again in a similar direction. → Makes it pretty inefficient.
- Basic idea of conjugate gradients: March in directions that are mutually orthogonal in the sense of $\mathbf{h}_i\mathbf{H}\mathbf{h}_{i+1}=\mathbf{0}$.
- The conjugate-gradient method constructs a sequence of orthogonal descent directions, $\mathbf{h}_0$, $\mathbf{h}_1$, ..., automatically.
- There are different ways of doing that, and each of these ways has its own name:
  - Fletcher-Reves method
  - Hestenes-Stiefel method
  - Polak-Ribiere method
- The conjugate-gradient method for exactly quadratic misfit functions:
  - Converges in *n* steps at most because:
  - **H** is constant and there are only *n* mutually orthogonal directions.
- The conjugate-gradient method for general misfit functions:
  - Is basically a leap of faith.
  - Tends to converge more quickly than steepest descent for many applications.

- Steepest descent tends to go again and again in a similar direction. → Makes it pretty inefficient.
- Basic idea of conjugate gradients: March in directions that are mutually orthogonal in the sense of $\mathbf{h}_i\mathbf{H}\mathbf{h}_{i+1}=\mathbf{0}$.

a) quadratic function



b) Rosenbrock function



BFGS Method [Broyden, Fletcher, Goldfarb, Shanno]

BFGS Method [Broyden, Fletcher, Goldfarb, Shanno]

- The Newton method converges quickly because it uses second-derivative information in the form of $\mathbf{H}^{-1}$.
- Second derivatives are really difficult to compute! And **H** is difficult to invert!
- Basic idea of the BFGS method: Construct an approximation of $\mathbf{H}^{-1}$ as we march through model space.

BFGS Method [Broyden, Fletcher, Goldfarb, Shanno]

- The Newton method converges quickly because it uses second-derivative information in the form of $\mathbf{H}^{-1}$.
- Second derivatives are really difficult to compute! And **H** is difficult to invert!
- Basic idea of the BFGS method: Construct an approximation of $\mathbf{H}^{-1}$ as we march through model space.

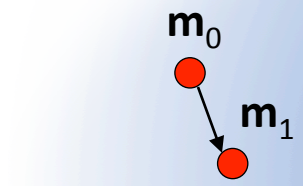
0^{th} approximation of $\mathbf{H}^{-1} = \mathbf{I}$

BFGS Method [Broyden, Fletcher, Goldfarb, Shanno]

- The Newton method converges quickly because it uses second-derivative information in the form of $\mathbf{H}^{-1}$.
- Second derivatives are really difficult to compute! And **H** is difficult to invert!
- Basic idea of the BFGS method: Construct an approximation of $\mathbf{H}^{-1}$ as we march through model space.

1st approximation of $\mathbf{H}^{-1} = \mathbf{A}_1$

based on $\mathbf{m}_0$, $\mathbf{m}_1$

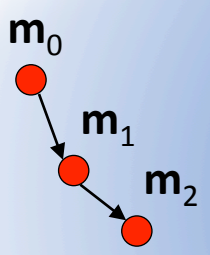


BFGS Method [Broyden, Fletcher, Goldfarb, Shanno]

- The Newton method converges quickly because it uses second-derivative information in the form of $\mathbf{H}^{-1}$.
- Second derivatives are really difficult to compute! And **H** is difficult to invert!
- Basic idea of the BFGS method: Construct an approximation of $\mathbf{H}^{-1}$ as we march through model space.

2^{nd} approximation of $\mathbf{H}^{-1} = \mathbf{A}_2$

based on $\mathbf{m}_0$, $\mathbf{m}_1$, $\mathbf{m}_2$

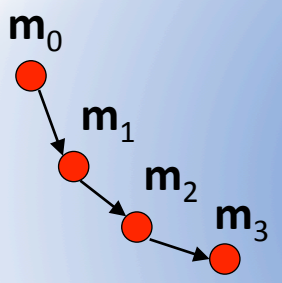


BFGS Method [Broyden, Fletcher, Goldfarb, Shanno]

- The Newton method converges quickly because it uses second-derivative information in the form of $\mathbf{H}^{-1}$.
- Second derivatives are really difficult to compute! And **H** is difficult to invert!
- Basic idea of the BFGS method: Construct an approximation of $\mathbf{H}^{-1}$ as we march through model space.

3rd approximation of $\mathbf{H}^{-1} = \mathbf{A}_3$

based on $\mathbf{m}_0$, $\mathbf{m}_1$, $\mathbf{m}_2$, $\mathbf{m}_3$

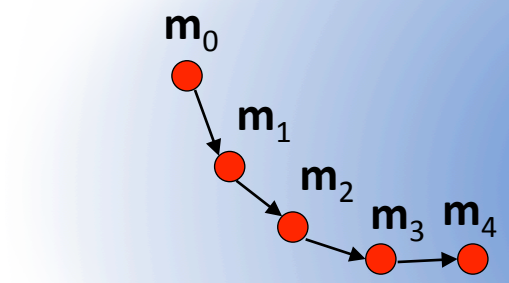


BFGS Method [Broyden, Fletcher, Goldfarb, Shanno]

- The Newton method converges quickly because it uses second-derivative information in the form of $\mathbf{H}^{-1}$.
- Second derivatives are really difficult to compute! And **H** is difficult to invert!
- Basic idea of the BFGS method: Construct an approximation of $\mathbf{H}^{-1}$ as we march through model space.

4^{th} approximation of $\mathbf{H}^{-1} = \mathbf{A}_4$

based on $\mathbf{m}_0$, $\mathbf{m}_1$, $\mathbf{m}_2$, $\mathbf{m}_3$, $\mathbf{m}_4$

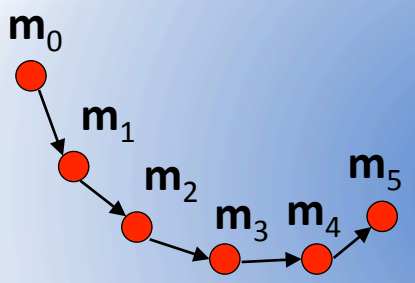


BFGS Method [Broyden, Fletcher, Goldfarb, Shanno]

- The Newton method converges quickly because it uses second-derivative information in the form of $\mathbf{H}^{-1}$.
- Second derivatives are really difficult to compute! And **H** is difficult to invert!
- Basic idea of the BFGS method: Construct an approximation of $\mathbf{H}^{-1}$ as we march through model space.

5^{th} approximation of $\mathbf{H}^{-1} = \mathbf{A}_5$

based on $\mathbf{m}_0$, $\mathbf{m}_1$, $\mathbf{m}_2$, $\mathbf{m}_3$, $\mathbf{m}_4$, $\mathbf{m}_5$

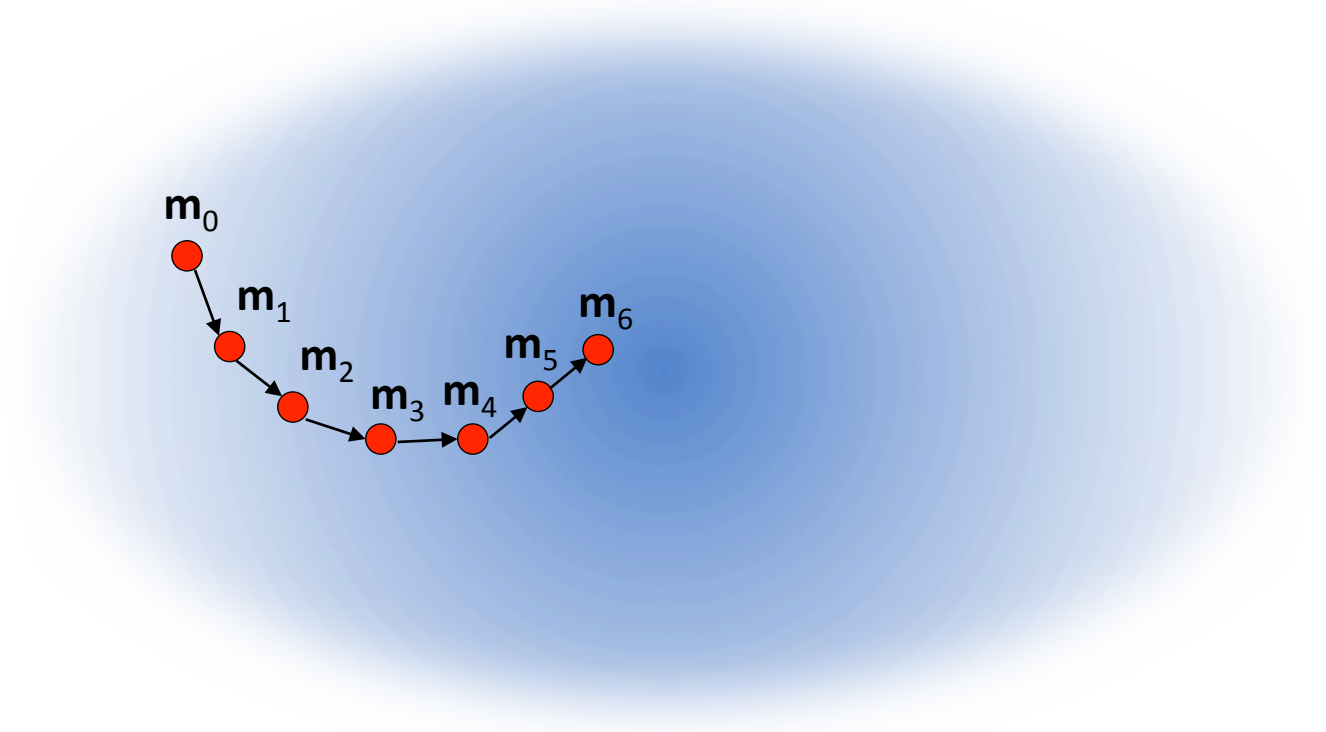


BFGS Method [Broyden, Fletcher, Goldfarb, Shanno]

- The Newton method converges quickly because it uses second-derivative information in the form of $\mathbf{H}^{-1}$.
- Second derivatives are really difficult to compute! And **H** is difficult to invert!
- Basic idea of the BFGS method: Construct an approximation of $\mathbf{H}^{-1}$ as we march through model space.

6^{th} approximation of $\mathbf{H}^{-1} = \mathbf{A}_6$

based on $\mathbf{m}_0$, $\mathbf{m}_1$, $\mathbf{m}_2$, $\mathbf{m}_3$, $\mathbf{m}_4$, $\mathbf{m}_5$, $\mathbf{m}_6$

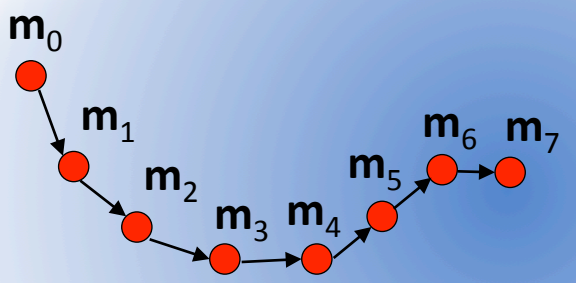


BFGS Method [Broyden, Fletcher, Goldfarb, Shanno]

- The Newton method converges quickly because it uses second-derivative information in the form of $\mathbf{H}^{-1}$.
- Second derivatives are really difficult to compute! And **H** is difficult to invert!
- Basic idea of the BFGS method: Construct an approximation of $\mathbf{H}^{-1}$ as we march through model space.

7^{th} approximation of $\mathbf{H}^{-1} = \mathbf{A}_7$

based on $\mathbf{m}_0$, $\mathbf{m}_1$, $\mathbf{m}_2$, $\mathbf{m}_3$, $\mathbf{m}_4$, $\mathbf{m}_5$, $\mathbf{m}_6$, $\mathbf{m}_7$



BFGS Method [Broyden, Fletcher, Goldfarb, Shanno]

- The Newton method converges quickly because it uses second-derivative information in the form of $\mathbf{H}^{-1}$.
- Second derivatives are really difficult to compute! And **H** is difficult to invert!
- Basic idea of the BFGS method: Construct an approximation of $\mathbf{H}^{-1}$ as we march through model space.
- For exactly quadratic misfit functions:
  - $\mathbf{A}_n=\mathbf{H}^{-1}$ [after *n* iterations we get the exact inverse Hessian]
  - Descent directions are mutually orthogonal, as in conjugate-gradient methods → convergence after *n* iterations.
- For general misfit functions: leap of faith but empirically very efficient in many cases.
- Behaviour for test functions: See Jupyter notebook!

L-BFGS METHOD

L-BFGS Method

- The BFGS method requires a lot of storage [matrix $\mathbf{A}_i$].
- Basic idea of the L-BFGS method: Just do not use all of the models along the path!

- The BFGS method requires a lot of storage [matrix $\mathbf{A}_i$].
- Basic idea of the L-BFGS method: Just do not use all of the models along the path!

0^{th} approximation of $\mathbf{H}^{-1} = \mathbf{I}$

- The BFGS method requires a lot of storage [matrix $\mathbf{A}_i$].
- Basic idea of the L-BFGS method: Just do not use all of the models along the path!

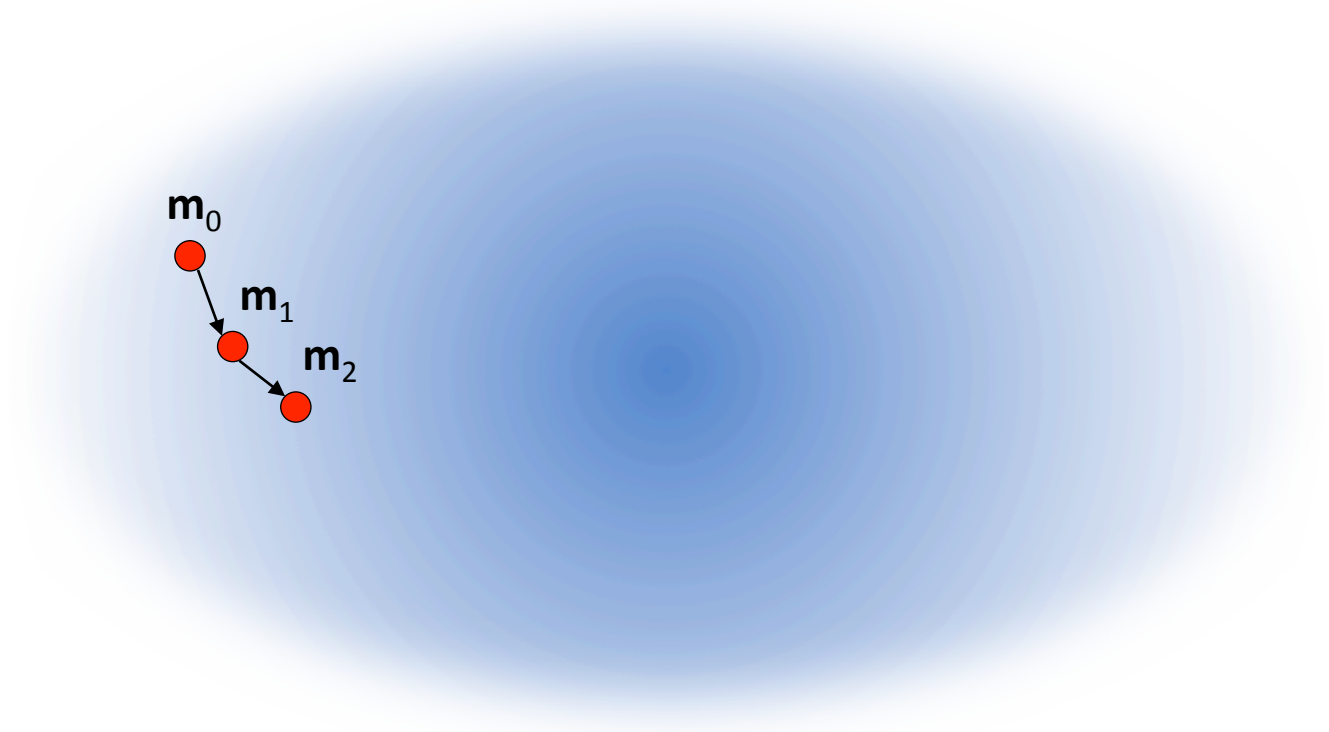
1st approximation of $\mathbf{H}^{-1} = \mathbf{A}_1$

based on $\mathbf{m}_0$, $\mathbf{m}_1$

- The BFGS method requires a lot of storage [matrix $\mathbf{A}_i$].
- Basic idea of the L-BFGS method: Just do not use all of the models along the path!

2nd approximation of $\mathbf{H}^{-1} = \mathbf{A}_2$

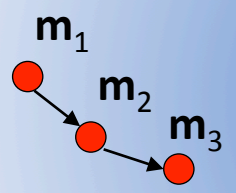
based on $\mathbf{m}_0$, $\mathbf{m}_1$, $\mathbf{m}_2$



- The BFGS method requires a lot of storage [matrix $\mathbf{A}_i$].
- Basic idea of the L-BFGS method: Just do not use all of the models along the path!

3rd approximation of $\mathbf{H}^{-1} = \mathbf{A}_3$

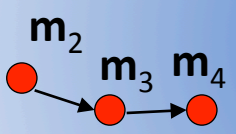
based on $\mathbf{m}_1$, $\mathbf{m}_2$, $\mathbf{m}_3$



- The BFGS method requires a lot of storage [matrix $\mathbf{A}_i$].
- Basic idea of the L-BFGS method: Just do not use all of the models along the path!

4^{th} approximation of $\mathbf{H}^{-1} = \mathbf{A}_4$

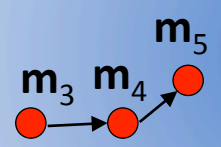
based on $\mathbf{m}_2$, $\mathbf{m}_3$, $\mathbf{m}_4$



- The BFGS method requires a lot of storage [matrix $\mathbf{A}_i$].
- Basic idea of the L-BFGS method: Just do not use all of the models along the path!

5^{th} approximation of $\mathbf{H}^{-1} = \mathbf{A}_5$

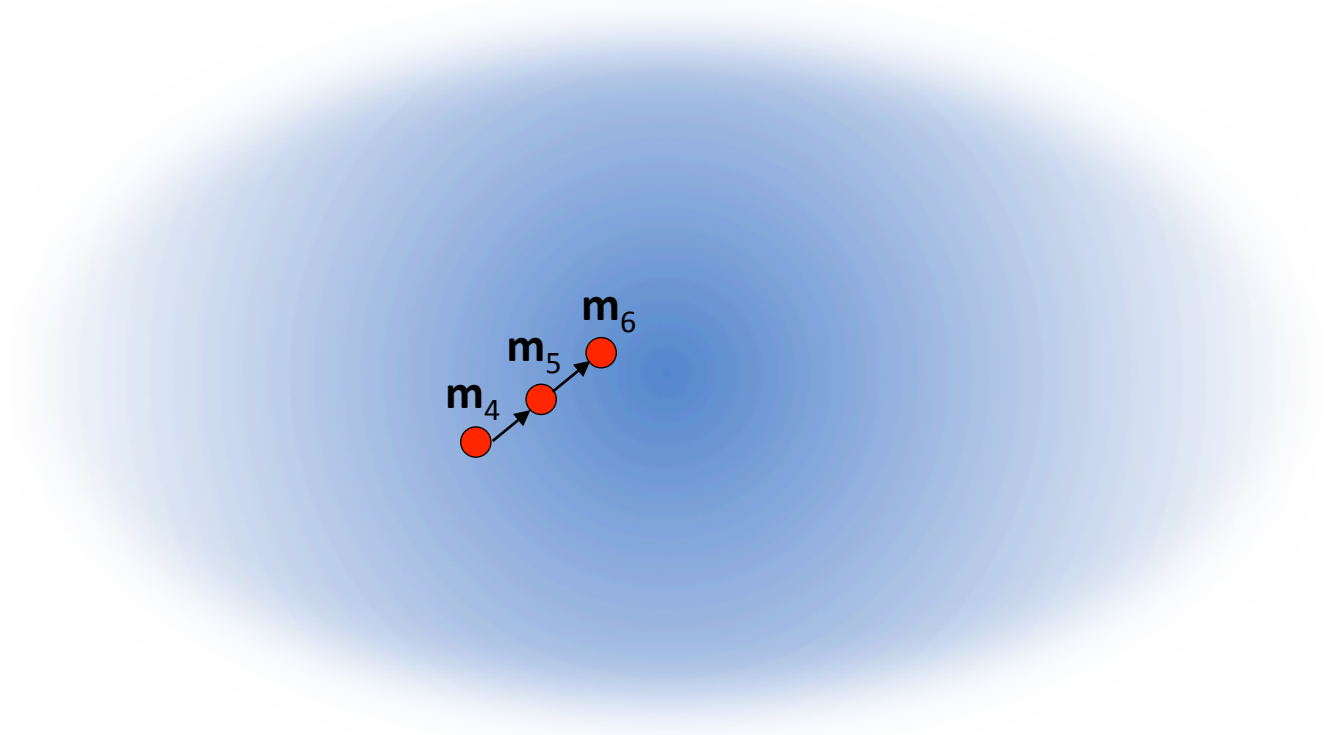
based on $\mathbf{m}_3$, $\mathbf{m}_4$, $\mathbf{m}_5$



- The BFGS method requires a lot of storage [matrix $\mathbf{A}_i$].
- Basic idea of the L-BFGS method: Just do not use all of the models along the path!

6^{th} approximation of $\mathbf{H}^{-1} = \mathbf{A}_6$

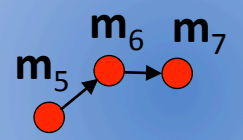
based on $\mathbf{m}_4$, $\mathbf{m}_5$, $\mathbf{m}_6$



- The BFGS method requires a lot of storage [matrix $\mathbf{A}_i$].
- Basic idea of the L-BFGS method: Just do not use all of the models along the path!

7^{th} approximation of $\mathbf{H}^{-1} = \mathbf{A}_7$

based on $\mathbf{m}_5$, $\mathbf{m}_6$, $\mathbf{m}_7$



L-BFGS Method

- The BFGS method requires a lot of storage [matrix $\mathbf{A}_i$].
- Basic idea of the L-BFGS method: Just do not use all of the models along the path!
- More difficult to implement than the other methods.
- Empirically hard to beat in the large-scale inverse problems that we deal with!
- Behavious for test functions: See Jupyter Notebook!

PART IV

Adjoint methods

1. Motivation

So, Where Is The Problem?

- The full gradient – with all its components - is needed in each iteration.
- The most straightforward approach: approximate the gradient by finite-differences:

$$\frac{\partial \chi(m)}{\partial m_k} \approx \frac{\chi(\ldots, m_k + \delta m, \ldots) - \chi(\ldots, m_k, \ldots)}{\delta m}$$

- Example with 500,000 model parameters:

	500,001 forward simulations
×	0.5 h per simulation
×	126 compute cores
×	50 sources (earthquakes)
×	50 conjugate gradient iterations
	$78e^9$ cpu hours ≈ 8,900,000 cpu years

2. Discrete adjoints

Regular wave equation

$$\underline{\underline{L}}\,\underline{u} = \underline{f}$$

Adjoint wave equation

$$\underline{\underline{L}}^{T}\,\underline{v} = -\nabla\chi$$

Gradient equation

$$\frac{\partial\chi}{\partial m_i} = \underline{v}^{T}\,\frac{\partial\underline{\underline{L}}}{\partial m_i}\,\underline{u}$$

Regular wave equation

$$\underline{\underline{L}}\,\underline{u} = \underline{f}$$

Adjoint wave equation

$$\underline{\underline{L}}^{T}\,\underline{v} = -\nabla\chi$$

Gradient equation

$$\frac{\partial\chi}{\partial m_i} = \underline{v}^{T}\,\frac{\partial\underline{\underline{L}}}{\partial m_i}\,\underline{u}$$

Adjoint recipe

1. Solve forward problem [regular wave equation] to obtain $\underline{u}$.
2. Evaluate misfit χ.
3. Compute adjoint source, $-\nabla\chi$.
4. Solve adjoint equation to obtain adjoint field $\underline{v}$.
5. Plug $\underline{u}$ and $\underline{v}$ into the gradient equation.

Discrete Adjoint Method Summary

Regular wave equation

$$\underline{\underline{L}}\,\underline{u} = \underline{f}$$

Adjoint wave equation

$$\underline{\underline{L}}^{T}\,\underline{v} = -\nabla\chi$$

Gradient equation

$$\frac{\partial\chi}{\partial m_i} = \underline{v}^{T}\,\frac{\partial\underline{\underline{L}}}{\partial m_i}\,\underline{u}$$

Adjoint recipe

1. Solve forward problem [regular wave equation] to obtain $\underline{u}$.
2. Evaluate misfit χ.
3. Compute adjoint source, $-\nabla\chi$.
4. Solve adjoint equation to obtain adjoint field $\underline{v}$.
5. Plug $\underline{u}$ and $\underline{v}$ into the gradient equation.

Comments

1. No need to explicitly compute the derivative of the wavefield $\underline{u}$ [by construction].
2. Gradient is entirely determined by the definition of the misfit [adjoint source is the only thing that explicitly depends on the misfit].
3. Computation of gradient requires storage of forward wavefield $\underline{u}$.

3. Continuous adjoints

Discrete case [frequency domain]

$$\underline{\underline{L}}\,\underline{u} = (-\omega^2 \underline{\underline{M}} + \underline{\underline{K}})\underline{u}$$

$$\nabla\chi = \underline{v}^T \nabla \underline{\underline{L}}\,\underline{u}$$

Continuous case [time domain]

$$L(\underline{u}) = \rho\ddot{\underline{u}} - \nabla\cdot(C:\nabla\underline{u}) = \underline{f}$$

$$\nabla\chi' = \int \underline{v}^T \nabla L(\underline{u})\,dt$$

- The same formal derivation from the discrete case can be used in the continuous case.
  - Matrix $\underline{\underline{L}}$ becomes operator L.
  - Scalar product $\underline{a}^T\underline{b}$ becomes integral $\int a(\mathbf{x})b(\mathbf{x})\,d\mathbf{x}$.
- In somewhat loose terms, $\nabla\chi$ is called a **sensitivity** or **Fréchet kernel** and symbolised by K.
- The only question: What is $\mathbf{L}^T$ in the continuous case?

Regular wave equation

momentum balance

$$\rho(\mathbf{x})\ddot{\mathbf{u}}(\mathbf{x},t) - \nabla \cdot \sigma(\mathbf{x},t) = \mathbf{f}(\mathbf{x},t)$$

stress-strain relation

$$\sigma(\mathbf{x},t) = \int_{\tau=t_0}^{\infty} \dot{\mathbf{C}}(\mathbf{x},t-\tau) : \nabla \mathbf{u}(\mathbf{x},\tau)\, d\tau$$

initial conditions

$$\mathbf{u}|_{t \le t_0} = \dot{\mathbf{u}}|_{t \le t_0} = \mathbf{0}$$

boundary conditions

$$\mathbf{n} \cdot \sigma|_{\mathbf{x} \in \partial G} = \mathbf{0}$$

Regular wave equation

momentum balance

$$\rho(\mathbf{x})\ddot{\mathbf{u}}(\mathbf{x},t) - \nabla\cdot\sigma(\mathbf{x},t) = \mathbf{f}(\mathbf{x},t)$$

stress-strain relation

$$\sigma(\mathbf{x},t) = \int_{\tau=t_0}^{\infty} \dot{\mathbf{C}}(\mathbf{x},t-\tau) : \nabla\mathbf{u}(\mathbf{x},\tau)\, d\tau$$

initial conditions

$$\mathbf{u}|_{t\le t_0} = \dot{\mathbf{u}}|_{t\le t_0} = \mathbf{0}$$

boundary conditions

$$\mathbf{n}\cdot\sigma|_{\mathbf{x}\in\partial G} = \mathbf{0}$$

Adjoint wave equation

adjoint momentum balance

$$\rho\ddot{\mathbf{u}}^{\dagger} - \nabla\cdot\sigma^{\dagger} = -\nabla_u\chi$$

adjoint stress-strain relation

$$\sigma^{\dagger}(t) = \int_{\tau=t}^{t_1} \dot{\mathbf{C}}(\tau-t) : \nabla\mathbf{u}^{\dagger}(\tau)\, d\tau$$

terminal conditions

$$\mathbf{u}^{\dagger}|_{t\ge t_1} = \dot{\mathbf{u}}^{\dagger}|_{t\ge t_1} = \mathbf{0}$$

boundary conditions

$$\mathbf{n}\cdot\sigma^{\dagger}|_{\mathbf{x}\in\partial G} = \mathbf{0}$$

notation

$$\mathbf{v} = \mathbf{u}^{\dagger}$$

Regular wave equation

momentum balance

$$\rho(\mathbf{x})\ddot{\mathbf{u}}(\mathbf{x},t) - \nabla\cdot\sigma(\mathbf{x},t) = \mathbf{f}(\mathbf{x},t)$$

stress-strain relation

$$\sigma(\mathbf{x},t) = \int_{\tau=t_0}^{\infty} \dot{\mathbf{C}}(\mathbf{x},t-\tau) : \nabla\mathbf{u}(\mathbf{x},\tau)\, d\tau$$

initial conditions

$$\mathbf{u}|_{t\le t_0} = \dot{\mathbf{u}}|_{t\le t_0} = \mathbf{0}$$

boundary conditions

$$\mathbf{n}\cdot\sigma|_{\mathbf{x}\in\partial G} = \mathbf{0}$$

Adjoint wave equation

adjoint momentum balance

$$\rho\ddot{\mathbf{u}}^{\dagger} - \nabla\cdot\sigma^{\dagger} = -\nabla_u\chi$$

adjoint stress-strain relation

$$\sigma^{\dagger}(t) = \int_{\tau=t}^{t_1} \dot{\mathbf{C}}(\tau-t) : \nabla\mathbf{u}^{\dagger}(\tau)\, d\tau$$

terminal conditions

$$\mathbf{u}^{\dagger}|_{t\ge t_1} = \dot{\mathbf{u}}^{\dagger}|_{t\ge t_1} = \mathbf{0}$$

boundary conditions

$$\mathbf{n}\cdot\sigma^{\dagger}|_{\mathbf{x}\in\partial G} = \mathbf{0}$$

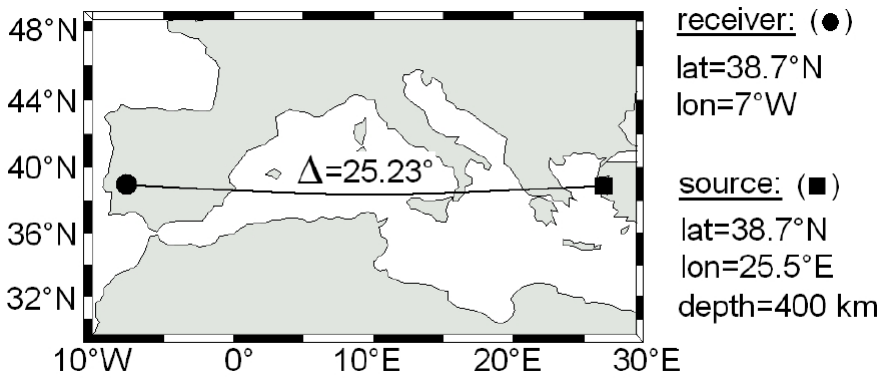
Comments

- Adjoint equation is a wave equation [same code can be used for its solution].
- Solving terminal conditions can be done by running code in reversed time.

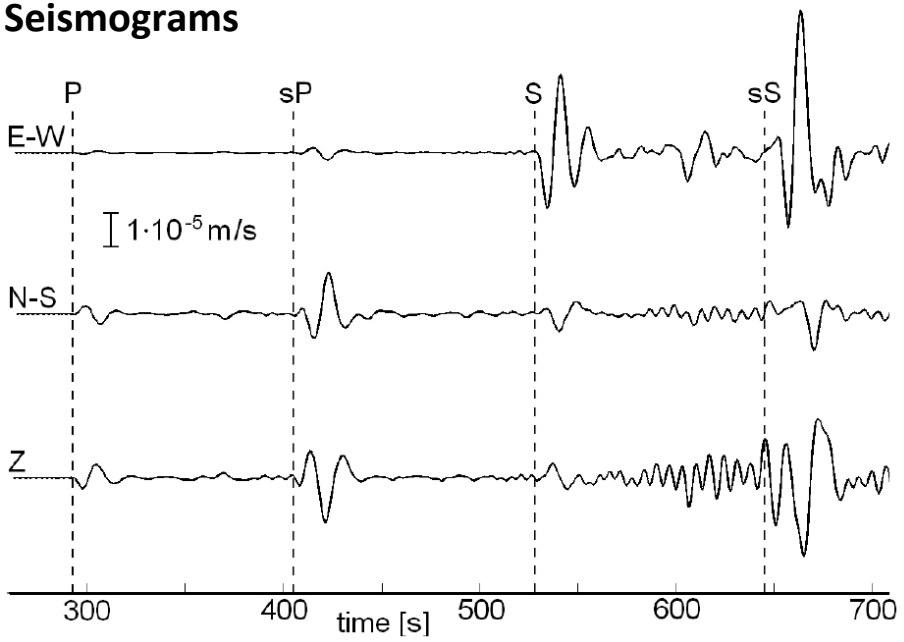
4. Sensitivity kernels and kernel gallery

Traveltime Measurement On Specific Phases

Source-receiver geometry

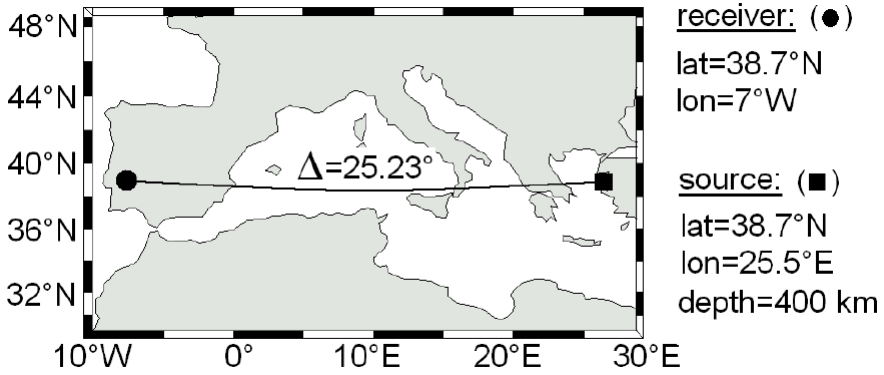


Seismograms

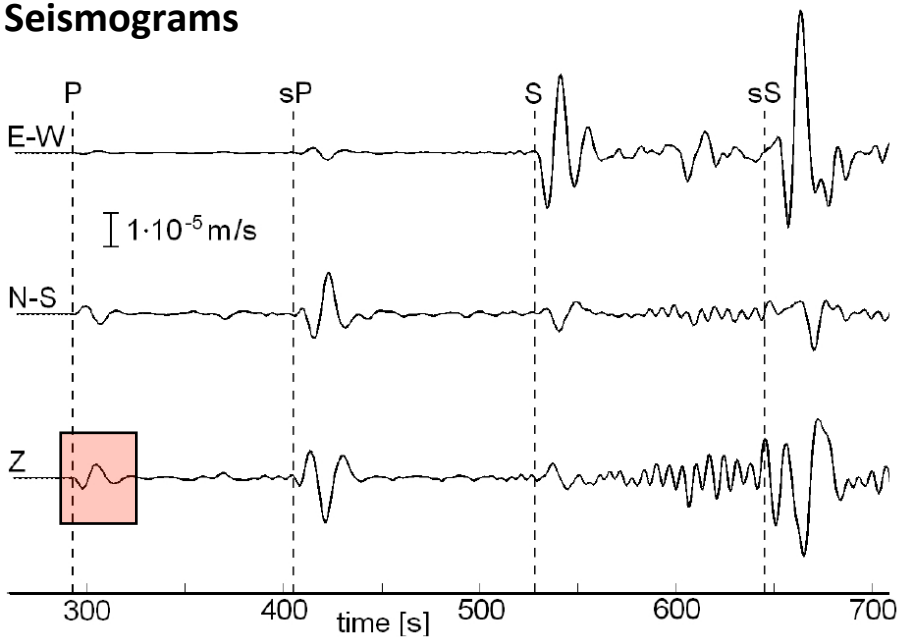


Traveltime Measurement On Specific Phases

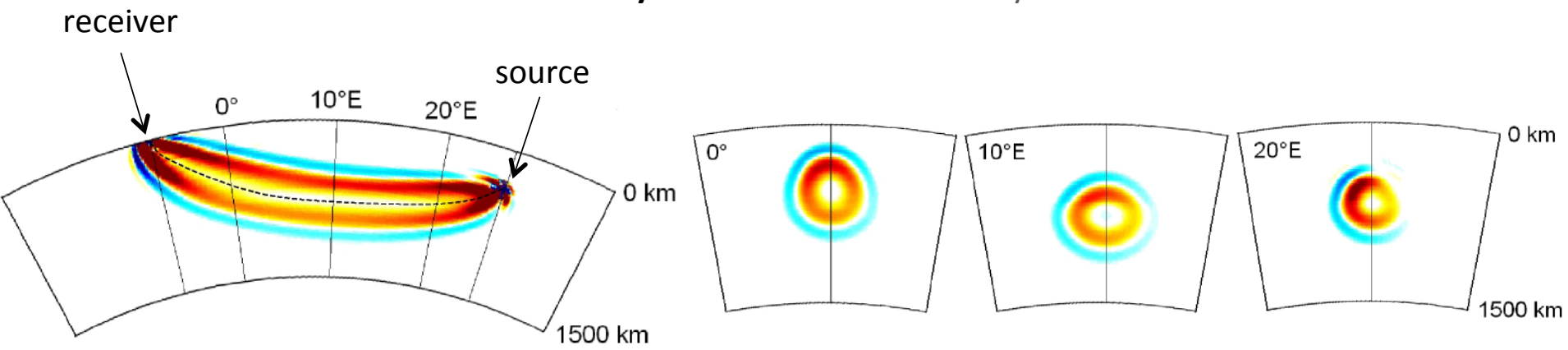
Source-receiver geometry



Seismograms

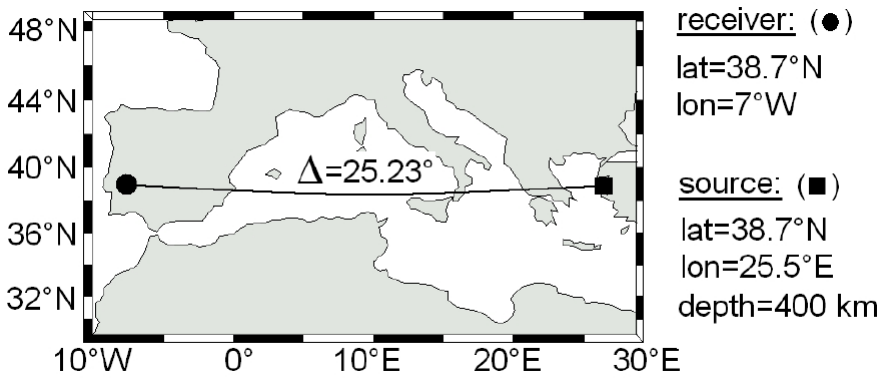


Sensitivity kernel for P wave velocity



Traveltime Measurement On Specific Phases

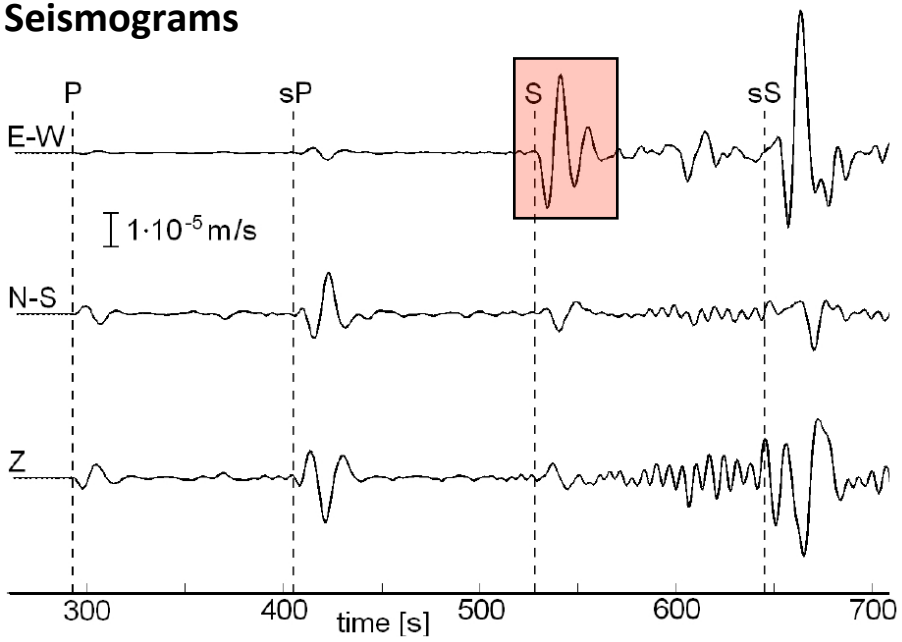
Source-receiver geometry



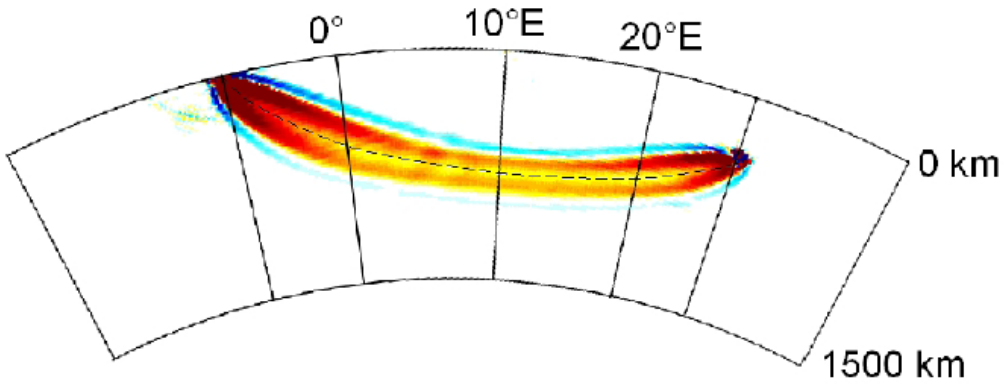
receiver: (●)
lat=38.7°N
lon=7°W

source: (■)
lat=38.7°N
lon=25.5°E
depth=400 km

Seismograms

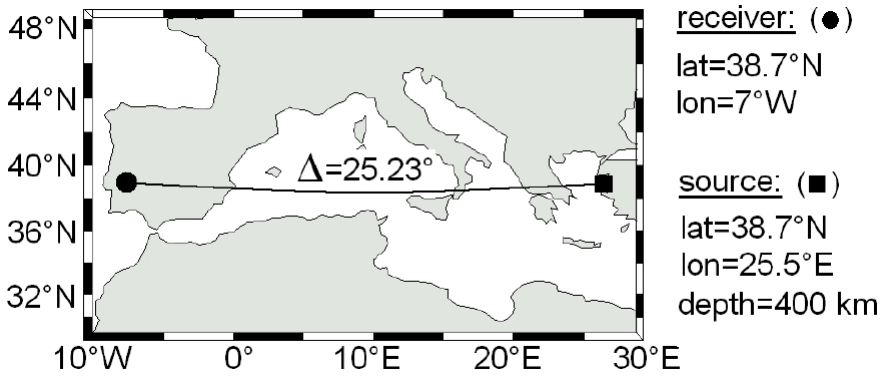


Sensitivity kernel for S wave velocity



Traveltime Measurement On Specific Phases

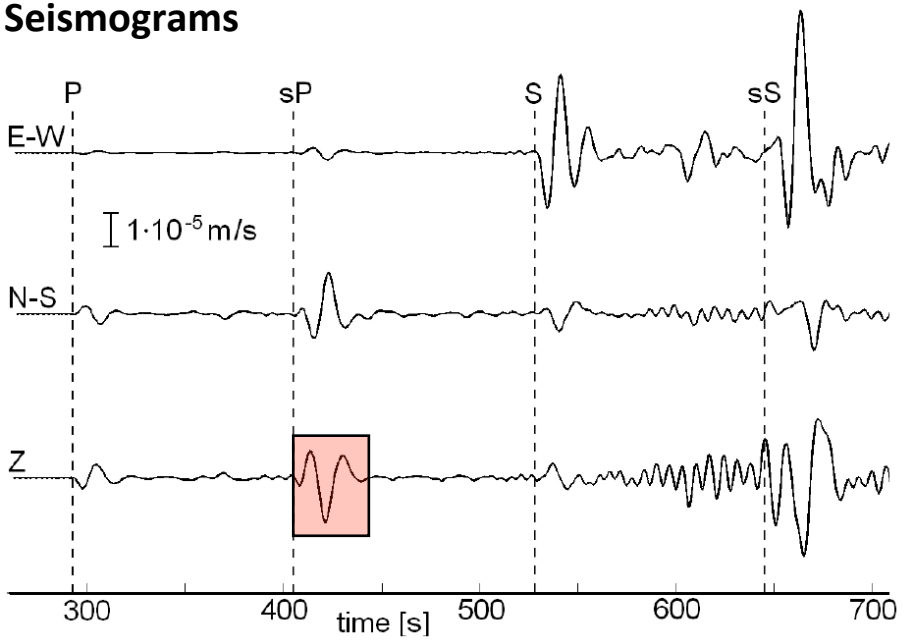
Source-receiver geometry



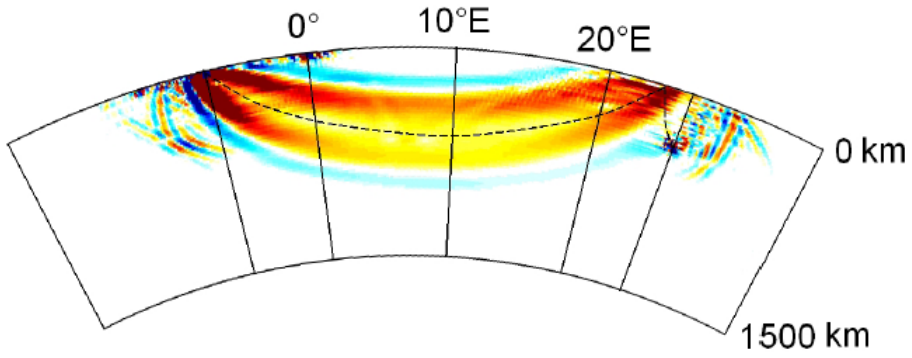
receiver: (●)
lat=38.7°N
lon=7°W

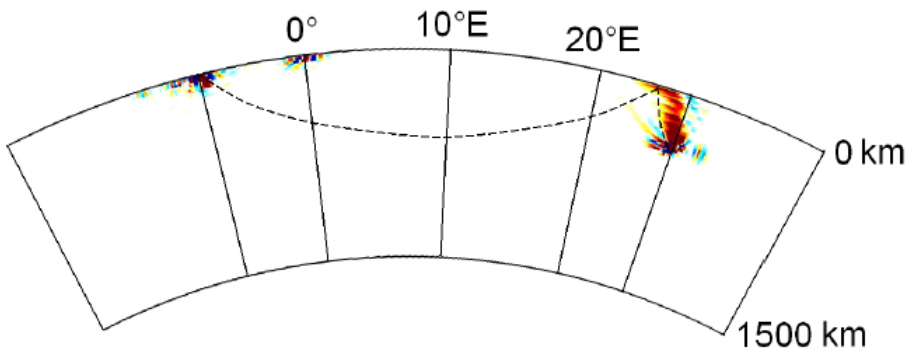
source: (■)
lat=38.7°N
lon=25.5°E
depth=400 km

Seismograms



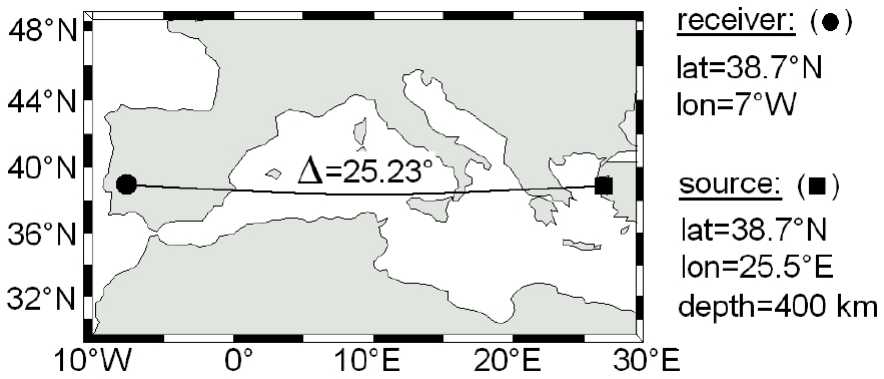
Sensitivity kernels for P wave velocity and S wave velocity





Traveltime Measurement On Specific Phases

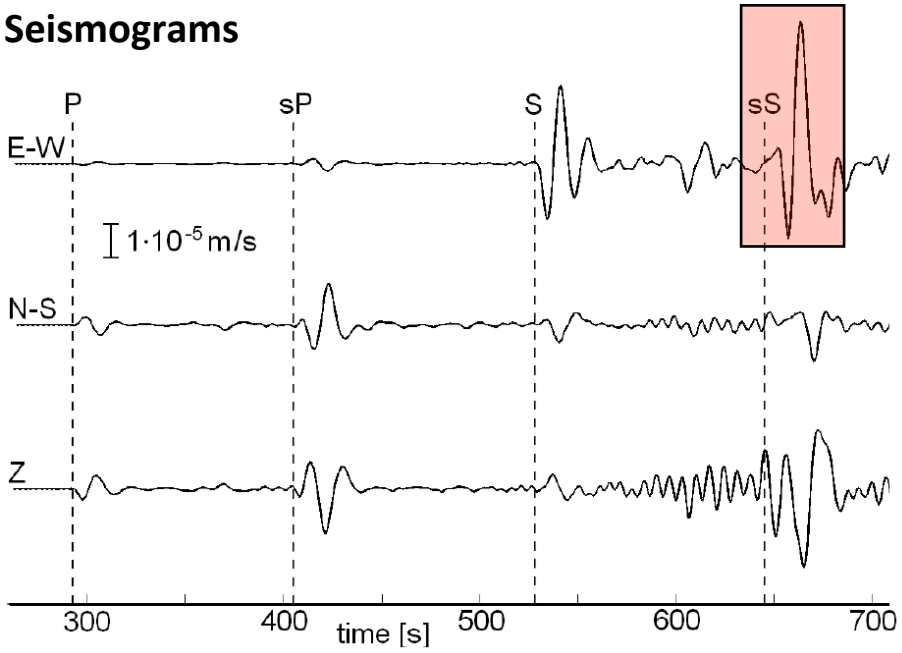
Source-receiver geometry



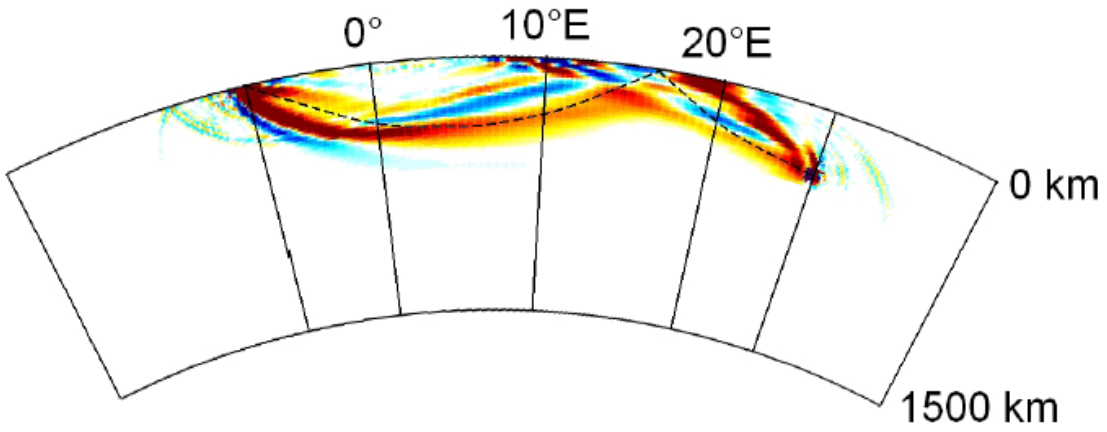
receiver: (●)
lat=38.7°N
lon=7°W

source: (■)
lat=38.7°N
lon=25.5°E
depth=400 km

Seismograms

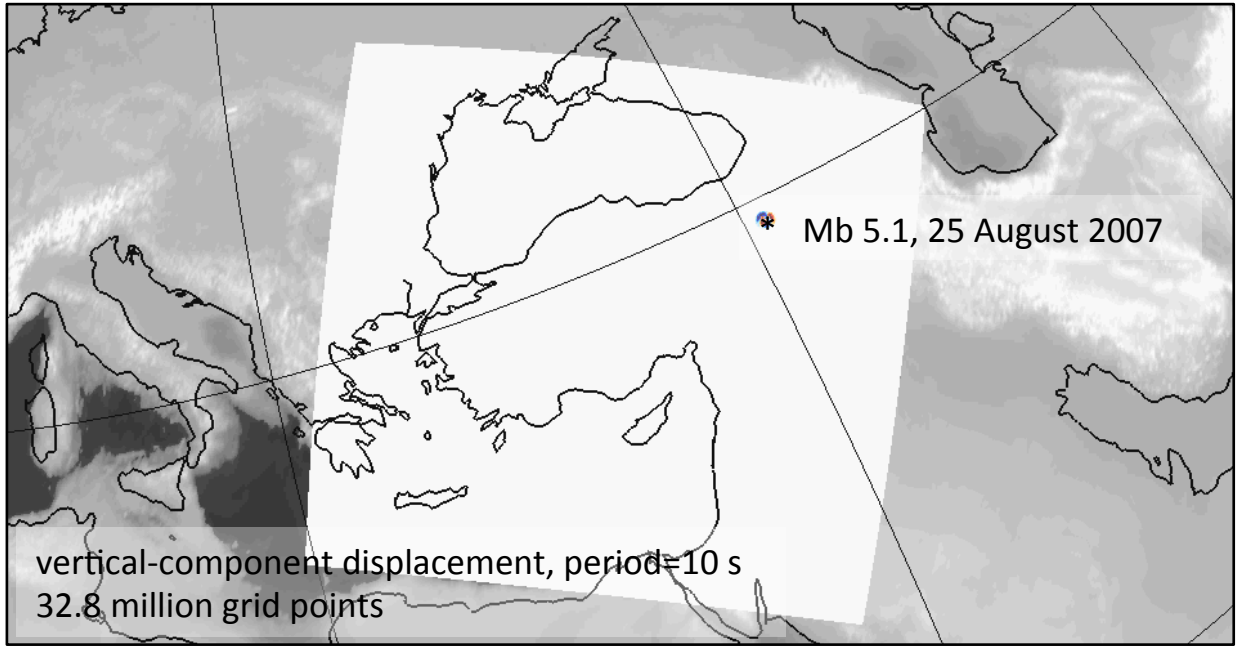


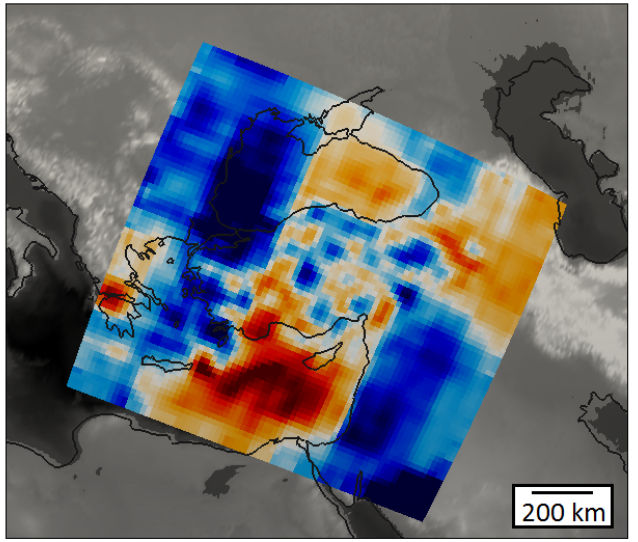
Sensitivity kernel for S wave velocity



Measuring Time-Frequency Phase Differences

Measuring Time-Frequency Phase Differences

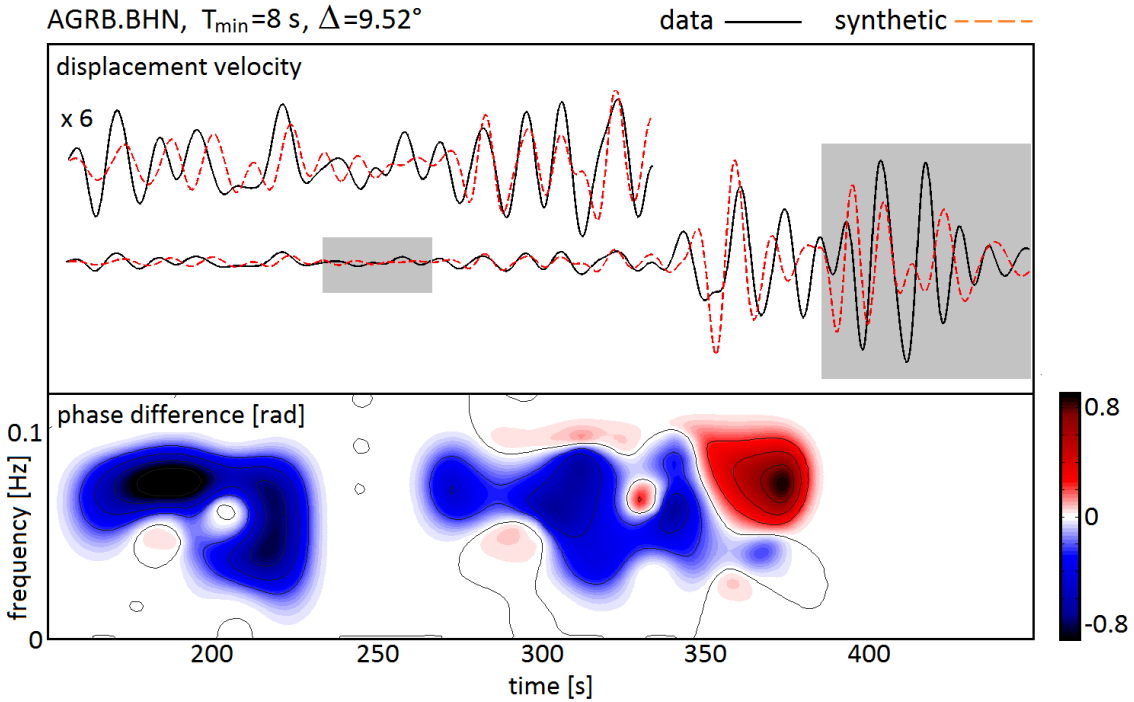


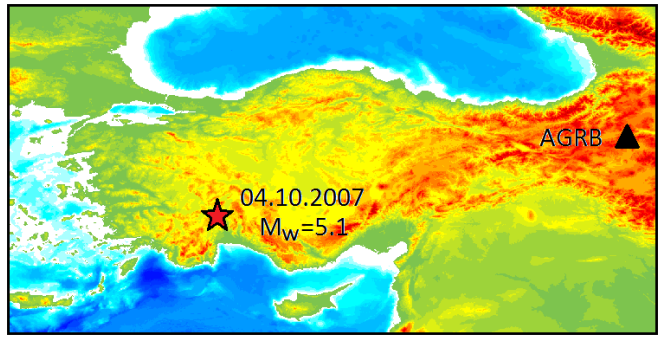


2.8 — 3.7

v_{sv}@ 20 km [km/s]

Measuring Time-Frequency Phase Differences





- Time- and frequency-dependent phase differences
- Based on selection of time windows where data and synthetics are similar
- Independent of absolute amplitudes

Sensitivity kernels